Package ‘rerf’

March 15, 2019

Type Package

Title Randomer Forest

Version 2.0.4

Date 2019-03-15

Description R-RerF (aka Randomer Forest (RerF) or Random Projection Forests) is an algorithm developed by Tomita (2016) <arXiv:1506.03410v2> which is similar to Random Forest - Random Combination (Forest-RC) developed by Breiman (2001) <doi:10.1023/A:1010933404324>. Random Forests create axis-parallel, or orthogonal trees. That is, the feature space is recursively split along directions parallel to the axes of the feature space. Thus, in cases in which the classes seem inseparable along any single dimension, Random Forests may be suboptimal. To address this, Breiman also proposed and characterized Forest-RC, which uses linear combinations of coordinates rather than individual coordinates, to split along. This package, ‘rerf’, implements RerF which is similar to Forest-RC. The difference between the two algorithms is where the random linear combinations occur: Forest-RC combines features at the per tree level whereas RerF takes linear combinations of coordinates at every node in the tree.

Depends R (>= 3.3.0), Rcpp (>= 1.0.0)

License Apache License 2.0 | file LICENSE

URL https://github.com/neurodata/R-RerF

BugReports https://github.com/neurodata/R-RerF/issues

Imports parallel, RcppZiggurat, utils, stats, dummies, mclust

Suggests roxygen2 (>= 5.0.0), testthat

LinkingTo Rcpp, RcppArmadillo

SystemRequirements GNU make

ByteCompile true

RoxygenNote 6.1.1

NeedsCompilation yes
Author Jesse Patsolic [ctb, cre],
    Benjamin Falk [ctb],
    Jaewon Chung [ctb],
    James Browne [aut],
    Tyler Tomita [aut],
    Joshua Vogelstein [ths]
Maintainer Jesse Patsolic <software@neurodata.io>
Repository CRAN
Date/Publication 2019-03-15 18:50:03 UTC

R topics documented:

BICCutFast ........................................... 3
BICCutMclust ......................................... 3
BuildTree ............................................. 4
checkInputMatrix ..................................... 5
ComputeSimilarity ..................................... 5
FeatureImportance ..................................... 6
flipWeights ........................................... 7
getFeatures .......................................... 8
getWeights ............................................ 8
GrowUnsupervisedForest ............................... 9
makeA ................................................ 10
makeAB ............................................... 10
mnist ............................................... 11
OOBPredict .......................................... 12
PackForest ........................................... 13
PackPredict ......................................... 13
Predict ............................................... 14
PrintTree ............................................. 15
RandMatBinary ........................................ 16
RandMatContinuous .................................. 17
RandMatCustom ....................................... 17
RandMatFRC ......................................... 18
RandMatFRCN ........................................ 19
RandMatImageControl ................................. 20
RandMatImagePatch ................................... 21
RandMatPoisson ...................................... 22
RandMatRF ........................................... 22
RandMatTSpatch ....................................... 23
RerF ................................................ 24
RunFeatureImportance ................................ 27
RunFeatureImportanceBinary ......................... 27
RunFeatureImportanceCounts ........................ 28
RunOOB .............................................. 29
RunPredict .......................................... 29
RunPredictLeaf ...................................... 30
**BICCutFast**

*Find minimizing BIC Cut for Vector*

**Description**

Find minimizing BIC Cut for Vector

**Usage**

```r
BICCutFast(x)
```

**Arguments**

- `x` a one dimensional vector

**Value**

list containing minimizing cut point and corresponding BIC score.

---

**BICCutMclust**

*Find minimizing BIC Cut for Vector*

**Description**

Find minimizing BIC Cut for Vector

**Usage**

```r
BICCutMclust(x)
```

**Arguments**

- `x` a one dimensional vector

**Value**

list containing minimizing cut point and corresponding BIC score.
BuildTree

**RerF Tree Generator**

**Description**

Creates a single decision tree based on an input matrix and class vector. This is the function used by rerf to generate trees.

**Usage**

```r
BuildTree(X, Y, FUN, paramList, min.parent, max.depth, bagging, replacement, stratify, class.ind, class.ct, store.oob, store.impurity, progress, rotate)
```

**Arguments**

- **X**: an n by d numeric matrix (preferable) or data frame. The rows correspond to observations and columns correspond to features.
- **Y**: an n length vector of class labels. Class labels must be integer or numeric and be within the range 1 to the number of classes.
- **FUN**: a function that creates the random projection matrix.
- **paramList**: parameters in a named list to be used by FUN. If left unchanged, default values will be populated, see `defaults` for details.
- **min.parent**: the minimum splittable node size. A node size < min.parent will be a leaf node. (min.parent = 6)
- **max.depth**: the longest allowable distance from the root of a tree to a leaf node (i.e. the maximum allowed height for a tree). If max.depth=0, the tree will be allowed to grow without bound.
- **bagging**: a non-zero value means a random sample of X will be used during tree creation. If replacement = FALSE the bagging value determines the percentage of samples to leave out-of-bag. If replacement = TRUE the non-zero bagging value is ignored.
- **replacement**: if TRUE then n samples are chosen, with replacement, from X.
- **stratify**: if TRUE then class sample proportions are maintained during the random sampling. Ignored if replacement = FALSE.
- **class.ind**: a vector of lists. Each list holds the indexes of its respective class (e.g. list 1 contains the index of each class 1 sample).
- **class.ct**: a cumulative sum of class counts.
- **store.oob**: if TRUE then the samples omitted during the creation of a tree are stored as part of the tree. This is required to run OOBPredict().
- **store.impurity**: if TRUE then the reduction in Gini impurity is stored for every split. This is required to run FeatureImportance().
- **progress**: if true a pipe is printed after each tree is created. This is useful for large datasets.
- **rotate**: if TRUE then the data matrix X is uniformly randomly rotated.
checkInputMatrix

Value

Tree

Examples

x <- iris[, 1:5]
y <- as.numeric(iris[, 5])
# BuildTree(x, y, RandMatBinary, p = 4, d = 4, rho = 0.25, prob = 0.5)

checkInputMatrix  Determine if given input can be processed by Urerf.

Description

Determine if given input can be processed by Urerf.

Usage

checkInputMatrix(X)

Arguments

X  an Nxd matrix or Data frame of numeric values.

Value

stops function execution and outputs error if invalid input is detected.

computesimilarity  Compute Similarities

Description

Computes pairwise similarities between observations. The similarity between two points is defined as the fraction of trees such that two points fall into the same leaf node.

Usage

computeSimilarity(X, forest, num.cores = 0L, Xtrain = NULL)
**FeatureImportance**

### Arguments

- **X**: an n sample by d feature matrix (preferable) or data frame which was used to train the provided forest.
- **forest**: a forest trained using the `rerf` function, with COOB=TRUE.
- **num.cores**: the number of cores to use while training. If num.cores=0 then 1 less than the number of cores reported by the OS are used. (num.cores=0)
- **Xtrain**: an n by d numeric matrix (preferable) or data frame. This should be the same data matrix/frame used to train the forest, and is only required if RerF was called with rank.transform = TRUE. (Xtrain=NULL)

### Value

- **similarity**: a normalized n by n matrix of pairwise similarities

### Examples

```r
library(rerf)
X <- as.matrix(iris[, 1:4])
Y <- iris[[5L]]
forest <- RerF(X, Y, num.cores = 1L)
sim.matrix <- ComputeSimilarity(X, forest, num.cores = 1L)
```

---

**FeatureImportance**  
*Compute Feature Importance of a RerF model*

### Description

Computes feature importance of every unique feature used to make a split in the RerF model.

### Usage

```r
FeatureImportance(forest, num.cores = 0L, type = NULL)
```

### Arguments

- **forest**: a forest trained using the `RerF` function with argument store.impurity = TRUE
- **num.cores**: number of cores to use. If num.cores = 0, then 1 less than the number of cores reported by the OS are used. (num.cores = 0)
- **type**: character string specifying which method to use in calculating feature importance.
  - 'C' specifies that unique combinations of features should be counted across trees.
  - 'R' feature importance will be calculated as in *R*andomForest.
  - 'E' calculates the unique projections up to equivalence if the vector of projection weights parametrizes the same line in \( R^p \).
**flipWeights**

Value

a list with 3 elements,

imp The vector of scores/counts, corresponding to each feature.
features The features/projections used.
type The code for the method used.

Examples

```r
library(rerf)
num.cores <- 1L
forest <- RerF(as.matrix(iris[, 1:4]), iris[[5L]], num.cores = 1L, store.impurity = TRUE)

imp.C <- FeatureImportance(forest, num.cores, "C")
imp.R <- FeatureImportance(forest, num.cores, "R")
imp.E <- FeatureImportance(forest, num.cores, "E")

fRF <- RerF(as.matrix(iris[, 1:4]), iris[[5L]],
FUN = RandMatRF, num.cores = 1L, store.impurity = TRUE)

fRF.imp <- FeatureImportance(forest = fRF, num.cores = num.cores)
```

**Description**

A helper function to extract the feature weights from the projection vector stored in a tree object. Used in RunFeatureImportanceBinary.

**Usage**

```r
flipWeights(x)
```

**Arguments**

- **x**
  a list of unique.projections from the intermediate steps of the FeatureImportance function.

**Value**

x with sign of weights flipped.
**getFeatures**

*Extract feature indices from the sparse projection vector.*

**Description**

A helper function to extract the feature indices from the projection vector stored in a tree object.

**Usage**

```
getFeatures(x)
```

**Arguments**

- `x` : a list of unique.projections from the intermediate steps of the FeatureImportance function.

**Value**

list of unique feature combinations

---

**getWeights**

*Extract feature weights from the sparse projection vector.*

**Description**

A helper function to extract the feature weights from the projection vector stored in a tree object.

**Usage**

```
getWeights(x)
```

**Arguments**

- `x` : a list of unique.projections from the intermediate steps of the FeatureImportance function.

**Value**

list of unique feature weights
**GrowUnsupervisedForest**

*Description*

Creates UreF Tree.

**Usage**

```r
growunsupervisedforest(xL minparent = 1L, trees = 100L, maxdepth = Inf, bagging = 0.2L, replacement = TRUEL, FUN = makeAB, options = list(p = ncol(X), d = ceiling(ncol(X)^0.5L), sparsity = 1L/ncol(X)), Progress = TRUEL, splitCrit = "twomeans", LinearCombo = TRUE)
```

**Arguments**

- `x` an Nxd matrix or Data frame of numeric values.
- `minparent` the minimum splittable node size (MinParent=1).
- `trees` the number of trees to grow in a forest (trees=100).
- `maxdepth` the maximum depth allowed in a forest (MaxDepth=Inf).
- `bagging` only used experimentally. Determines the hold out size if replacement=FALSE (bagging=.2).
- `replacement` method used to determine boot strap samples (replacement=TRUE).
- `FUN` the function to create the rotation matrix used to determine mtry features.
- `options` options provided to FUN.
- `Progress` logical that determines whether to show tree creation status (Progress=TRUE).
- `splitCrit` split based on twomeans(splitCrit="twomeans") or BIC test(splitCrit="bicfast")
- `LinearCombo` logical that determines whether to use linear combination of features. (LinearCombo=TRUE).

**Value**

- `tree`
Description

This function is the default option to make the projection matrix for unsupervised random forest. The sparseM matrix is the projection matrix. The creation of this matrix can be changed, but the nrow of sparseM should remain p. The ncol of the sparseM matrix is currently set to mtry but this can actually be any integer > 1; can even be greater than p. The matrix returned by this function creates a sparse matrix with one feature per column.

Usage

makeA(p, d, sparsity, ...)

Arguments

p the number of dimensions.
d the number of desired columns in the projection matrix.
sparsity a real number in (0, 1) that specifies the distribution of non-zero elements in the random matrix.
... used to handle superfluous arguments passed in using paramList.

Value

rotationMatrix the matrix used to determine which mtry features or combination of features will be used to split a node.

Description

This function is the default option to make the projection matrix for unsupervised random forest. The sparseM matrix is the projection matrix. The creation of this matrix can be changed, but the nrow of sparseM should remain p. The ncol of the sparseM matrix is currently set to mtry but this can actually be any integer > 1; can even be greater than p. The matrix returned by this function creates a sparse matrix with multiple features per column.

Usage

makeAB(p, d, sparsity, ...)

Create rotation matrix used to determine mtry features.

Create rotation matrix used to determine linear combination of mtry features.
**Arguments**

- **p** the number of dimensions.
- **d** the number of desired columns in the projection matrix.
- **sparsity** a real number in (0, 1) that specifies the distribution of non-zero elements in the random matrix.
- ... used to handle superfluous arguments passed in using paramList.

**Value**

- **rotationMatrix** the matrix used to determine which mtry features or combination of features will be used to split a node.

---

**Description**

A dataset consisting of 10 percent of the MNIST training set and the full test set.

**Usage**

data(mnist)

**Format**

A list with four items: Xtrain is a training set matrix with 6000 rows (samples) and 784 columns (features), Xtrain is an integer array of corresponding training class labels, Xtest is a test set matrix of 10000 rows and 784 columns, and Ytest is the corresponding class labels. Rows in Xtrain and Xtest correspond to different images of digits, and columns correspond to the pixel intensities in each image, obtained by flattening the image pixels in column-major ordering.

**Source**

MNIST

**References**


**Examples**

data(mnist)
OOBPredict

*Compute out-of-bag predictions*

**Description**

Computes out-of-bag class predictions for a forest trained with store.oob=TRUE.

**Usage**

```r
OOBPredict(X, forest, num.cores = 0L, Xtrain = NULL, output.scores = FALSE)
```

**Arguments**

- **X**: an n sample by d feature matrix (preferable) or data frame which was used to train the provided forest.
- **forest**: a forest trained using the RerF function, with store.oob=TRUE.
- **num.cores**: the number of cores to use while training. If num.cores=0 then 1 less than the number of cores reported by the OS are used. (num.cores=0)
- **Xtrain**: an n by d numeric matrix (preferable) or data frame. This should be the same data matrix/frame used to train the forest, and is only required if RerF was called with rank.transform = TRUE. (Xtrain=NULL)
- **output.scores**: if TRUE then predicted class scores (probabilities) for each observation are returned rather than class labels. (output.scores = FALSE)

**Value**

predictions a length n vector of predictions in a format similar to the Y vector used to train the forest

**Examples**

```r
library(rerf)
X <- as.matrix(iris[, 1:4])
Y <- iris[[5L]]
forest <- RerF(X, Y, store.oob = TRUE, num.cores = 1L)
predictions <- OOBPredict(X, forest, num.cores = 1L)
oob.error <- mean(predictions != Y)
```
PackForest

Packs a forest and saves modified forest to disk for use by PackPredict function

Description

Efficiently packs a forest trained with the RF option. Two intermediate data structures are written to disk, forestPackTempFile.csv and traversalPackTempFile.csv. The size of these data structures is proportional to a trained forest and training data respectively. Both data structures are removed at the end of the operation. The resulting forest is saved as forest.out. The size of this file is similar to the size of the trained forest.

Usage

PackForest(X, Y, forest)

Arguments

X: an n by d numeric matrix (preferable) or data frame used to train the forest.
Y: a numeric vector of size n. If the Y vector used to train the forest was not of type numeric then a simple call to as.numeric(Y) will suffice as input.
forest: a forest trained using the RerF function using the RF option.

PackPredict

Compute class predictions for each observation in X

Description

Predicts the classification of samples using a trained forest.

Usage

PackPredict(X, num.cores = 1)

Arguments

X: an n by d numeric matrix (preferable) or data frame. The rows correspond to observations and columns correspond to features of a test set, which should be different from the training set.
num.cores: the number of cores to use while predicting. (num.cores=0)

Value

predictions an n length vector of prediction class numbers
Examples

```r
library(rerf)
trainIdx <- c(1:40, 51:90, 101:140)
X <- as.matrix(iris[, 1:4])
Y <- as.numeric(iris[, 5])

paramList <- list(p = ncol(X), d = ceiling(sqrt(ncol(X))))

forest <- RerF(X, Y, FUN = RandMatRF, paramList = paramList, rfPack = TRUE, num.cores = 1)
predictions <- PackPredict(X)
```

<table>
<thead>
<tr>
<th>Predict</th>
<th>Compute class predictions for each observation in X</th>
</tr>
</thead>
</table>

Description

Predicts the classification of samples using a trained forest.

Usage

```r
predict(X, forest, OOB = FALSE, num.cores = 0L, Xtrain = NULL,
aggregate.output = TRUE, output.scores = FALSE)
```

Arguments

- **X**: an n by d numeric matrix (preferable) or data frame. The rows correspond to observations and columns correspond to features of a test set, which should be different from the training set.
- **forest**: a forest trained using the RerF function.
- **OOB**: if TRUE then run predictions using out-of-bag samples.
- **num.cores**: the number of cores to use while training. If NumCores=0 then 1 less than the number of cores reported by the OS are used. (NumCores=0)
- **Xtrain**: an n by d numeric matrix (preferable) or data frame. This should be the same data matrix/frame used to train the forest, and is only required if RerF was called with rank.transform = TRUE. (Xtrain=NULL)
- **aggregate.output**: if TRUE then the tree predictions are aggregated weighted by their probability estimates. Otherwise, the individual tree probabilities are returned. (aggregate.output=TRUE)
- **output.scores**: if TRUE then predicted class scores (probabilities) for each observation are returned rather than class labels. (output.scores = FALSE)
Value
predictions an n length vector of predictions

Examples
library(rerf)
trainIdx <- c(1:40, 51:90, 101:140)
X <- as.matrix(iris[,1:4])
Y <- as.numeric(iris[,5])
forest <- RerF(X[trainIdx,], Y[trainIdx], num.cores = 1L, rank.transform = TRUE)
# Using a set of samples with unknown classification
predictions <- Predict(X[-trainIdx,], forest, num.cores = 1L, Xtrain = X[trainIdx,])
errorRate <- mean(predictions != Y[-trainIdx])

PrintTree

RerF Tree Printer

Description
Prints the layout of a specified tree.

Usage
PrintTree(forest, numTree = 1, pretty = FALSE)

Arguments
forest a rerf forest structure.
numTree the tree number to print. (numTree=1)
pretty boolean if TRUE the column of cut features are formatted nicely for viewing. (FALSE)

Value
a data.frame with the following information about the tree:

- `nodeNum` The node number
- `LC` The id of the left child of the node
- `RC` The id of the right child of the node
- `CutValue` The cut value of non-terminal nodes, otherwise NA.
- `nodeClass` The class vote of a terminal node when used for classification/prediction.
- `CutFeatures` a list of ordered pairs $(d, w)$, where $d$ is the original feature and $w$ is the corresponding weight.
Examples

```r
### Train RerF on numeric data ###
library(rerf)
numTree <- 1
forest <- RerF(as.matrix(iris[, 1:4]), iris[, 5], num.core = 1L)
forest.rmc <- RerF(as.matrix(iris[, 1:4]), iris[, 5], num.core = 1L, RandMatContinuous)
(out <- PrintTree(forest, numTree))
(out.rmc <- PrintTree(forest.rmc, numTree))
```

---

**RandMatBinary**

Create a Random Matrix: Binary

**Description**

Create a Random Matrix: Binary

**Usage**

```r
RandMatBinary(p, d, sparsity, prob, catMap = NULL, ...)
```

**Arguments**

- `p`: the number of dimensions.
- `d`: the number of desired columns in the projection matrix.
- `sparsity`: a real number in $(0, 1)$ that specifies the distribution of non-zero elements in the random matrix.
- `prob`: a probability $\in (0, 1)$ used for sampling from $-1, 1$ where prob = 0 will only sample -1 and prob = 1 will only sample 1.
- `catMap`: a list specifying specifies which one-of-K encoded columns in X correspond to the same categorical feature.
- `...`: used to handle superfluous arguments passed in using paramList.

**Value**

A random matrix to use in running RerF.

**Examples**

```r
p <- 8
d <- 3
sparsity <- 0.25
prob <- 0.5
set.seed(4)
(a <- RandMatBinary(p, d, sparsity, prob))
```
RandMatContinuous  

Create a Random Matrix: Continuous

**Description**
Create a Random Matrix: Continuous

**Usage**
```
RandMatContinuous(p, d, sparsity, catMap = NULL, ...)
```

**Arguments**
- `p`: the number of dimensions.
- `d`: the number of desired columns in the projection matrix.
- `sparsity`: a real number in \((0, 1)\) that specifies the distribution of non-zero elements in the random matrix.
- `catMap`: a list specifying which one-of-
K encoded columns in X correspond to the same categorical feature.
- `...`: used to handle superfluous arguments passed in using paramList.

**Value**
A random matrix to use in running `RerF`.

**Examples**
```
p <- 8
d <- 3
sparsity <- 0.25
set.seed(4)
(a <- RandMatContinuous(p, d, sparsity))
```

---

RandMatCustom  

Create a Random Matrix: custom

**Description**
Create a Random Matrix: custom

**Usage**
```
RandMatCustom(p, d, nnzSample, nnzProb, ...)
```
Arguments

\begin{itemize}
  \item \texttt{p} \hspace{1cm} the number of dimensions.
  \item \texttt{d} \hspace{1cm} the number of desired columns in the projection matrix.
  \item \texttt{nnzSample} \hspace{1cm} a vector specifying the number of non-zeros to sample at each \texttt{d}. Each entry should be less than \texttt{p}.
  \item \texttt{nnzProb} \hspace{1cm} a vector specifying probabilities in one-to-one correspondance with \texttt{nnzSample}.
  \texttt{...} \hspace{1cm} used to handle superfluous arguments passed in using \texttt{paramList}.
\end{itemize}

Value

A random matrix to use in running \texttt{RerF}.

Examples

```
\begin{verbatim}
p <- 28
d <- 8
nnzSample <- 1:8
nnzProb <- 1 / 36 * 1:8
paramList <- list(p = p, d = d, nnzSample, nnzProb)
set.seed(8)
(a <- do.call(RandMatCustom, paramList))
\end{verbatim}
```

Description

Create a Random Matrix: FRC

Usage

\texttt{RandMatFRC(p, d, nmix, catMap = \texttt{NULL}, ...)}

Arguments

\begin{itemize}
  \item \texttt{p} \hspace{1cm} integer the number of dimensions.
  \item \texttt{d} \hspace{1cm} integer the number of desired columns in the projection matrix.
  \item \texttt{nmix} \hspace{1cm} integer multiplier to \texttt{d} to specify the number of non-zeros.
  \item \texttt{catMap} \hspace{1cm} a list specifying specifies which one-of-K encoded columns in \texttt{X} correspond to the same categorical feature.
  \texttt{...} \hspace{1cm} used to handle superfluous arguments passed in using \texttt{paramList}.
\end{itemize}

Value

A random matrix to use in running \texttt{RerF}.
RandMatFRCN

Examples

```r
p <- 8
d <- 2
nmix <- 5
paramList <- list(p = p, d = d, nmix = nmix)
set.seed(4)
(a <- do.call(RandMatFRCN, paramList))
```

Description

Create a Random Matrix: FRCN

Usage

```r
RandMatFRCN(p, d, nmix, catMap = NULL, ...)
```

Arguments

- `p`: the number of dimensions.
- `d`: the number of desired columns in the projection matrix.
- `nmix`: multiplier to `d` to specify the number of non-zeros.
- `catMap`: a list specifying which one-of-K encoded columns in `X` correspond to the same categorical feature.
- `...`: used to handle superfluous arguments passed in using paramList.

Value

A random matrix to use in running `RerF`.

Examples

```r
p <- 8
d <- 8
nmix <- 5
paramList <- list(p = p, d = d, nmix = nmix)
set.seed(8)
(a <- do.call(RandMatFRCN, paramList))
```
RandMatImageControl  Create a Random Matrix: image-control

Description

Create a Random Matrix: image-control

Usage

RandMatImageControl(p, d, ih, iw, pwMin, pwMax, ...)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>the number of dimensions.</td>
</tr>
<tr>
<td>d</td>
<td>the number of desired columns in the projection matrix.</td>
</tr>
<tr>
<td>ih</td>
<td>the height (px) of the image.</td>
</tr>
<tr>
<td>iw</td>
<td>the width (px) of the image.</td>
</tr>
<tr>
<td>pwMin</td>
<td>the minimum patch size to sample.</td>
</tr>
<tr>
<td>pwMax</td>
<td>the maximum patch size to sample.</td>
</tr>
<tr>
<td>...</td>
<td>used to handle superfluous arguments passed in using paramList.</td>
</tr>
</tbody>
</table>

Value

A random matrix to use in running RerF.

Examples

```r
p <- 28^2
d <- 8
ih <- iw <- 28
pwMin <- 3
pwMax <- 6
paramList <- list(p = p, d = d, ih = ih, iw = iw, pwMin = pwMin, pwMax = pwMax)
set.seed(8)
(a <- do.call(RandMatImageControl, paramList))
```
RandMatImagePatch

Create a Random Matrix: image-patch

Description

Create a Random Matrix: image-patch

Usage

RandMatImagePatch(p, d, ih, iw, pwMin, pwMax, ...)

Arguments

p the number of dimensions.
d the number of desired columns in the projection matrix.
ih the height (px) of the image.
iw the width (px) of the image.
pwMin the minimum patch size to sample.
pwMax the maximum patch size to sample.
... used to handle superfluous arguments passed in using paramList.

Value

A random matrix to use in running RerF.

Examples

p <- 28^2
d <- 8
ih <- iw <- 28
pwMin <- 3
pwMax <- 6
paramList <- list(p = p, d = d, ih = ih, iw = iw, pwMin = pwMin, pwMax = pwMax)
set.seed(8)
(a <- do.call(RandMatImagePatch, paramList))
RandMatPoisson  

Create a Random Matrix: Poisson

Description
Samples a binary projection matrix where sparsity is distributed $\text{Poisson}(\lambda)$.

Usage
RandMatPoisson(p, d, lambda, catMap = NULL, ...)

Arguments
- p: the number of dimensions.
- d: the number of desired columns in the projection matrix.
- lambda: passed to the \texttt{rpois} function for generation of non-zero elements in the random matrix.
- catMap: a list specifying specifies which one-of-K encoded columns in X correspond to the same categorical feature.
- ...: used to handle superfluous arguments passed in using paramList.

Value
A random matrix to use in running \texttt{RerF}.

Examples
```
p <- 8
d <- 8
lambda <- 0.5
paramList <- list(p = p, d = d, lambda = lambda)
set.seed(8)
(a <- do.call(RandMatPoisson, paramList))
```

RandMatRF  

Create a Random Matrix: Random Forest (RF)

Description
Create a Random Matrix: Random Forest (RF)

Usage
RandMatRF(p, d, catMap = NULL, ...)
Arguments

- **p**: the number of dimensions.
- **d**: the number of desired columns in the projection matrix.
- **catMap**: a list specifying which one-of-K encoded columns in X correspond to the same categorical feature.
- ...: used to handle superfluous arguments passed in using paramList.

Value

A random matrix to use in running `RerF`.

Examples

```r
p <- 8
d <- 3
paramList <- list(p = p, d = d)
set.seed(4)
(a <- do.call(RandMatRF, paramList))
```

---

RandMatTSpach

Description

Create a Random Matrix: ts-patch

Usage

```
RandMatTSpach(p, d, pwMin, pwMax, ...)
```

Arguments

- **p**: the number of dimensions.
- **d**: the number of desired columns in the projection matrix.
- **pwMin**: the minimum patch size to sample.
- **pwMax**: the maximum patch size to sample.
- ...: used to handle superfluous arguments passed in using paramList.

Value

A random matrix to use in running `RerF`.
Examples

```r
p <- 8
d <- 8
pwMin <- 3
pwMax <- 6
paramList <- list(p = p, d = d, pwMin = pwMin, pwMax = pwMax)
set.seed(8)
(a <- do.call(RandMatTSpitch, paramList))
```

---

**RerF forest Generator**

**Description**

Creates a decision forest based on an input matrix and class vector. This is the main function in the rerf package.

**Usage**

```r
RerF(X, Y, FUN = RandMatBinary, paramList = list(p = NA, d = NA,
              sparsity = NA, prob = NA), min.parent = 1L, trees = 500L,
              max.depth = 0, bagging = 0.2, replacement = TRUE,
              stratify = TRUE, rank.transform = FALSE, store.oob = FALSE,
              store.impurity = FALSE, progress = FALSE, rotate = FALSE,
              num.cores = 0L, seed = sample(0:1e+08, 1), cat.map = NULL,
              rfPack = FALSE)
```

**Arguments**

- **X**: an n by d numeric matrix (preferable) or data frame. The rows correspond to observations and columns correspond to features.
- **Y**: an n length vector of class labels. Class labels must be integer or numeric and be within the range 1 to the number of classes.
- **FUN**: a function that creates the random projection matrix. If NULL and cat.map is NULL, then RandMat is used. If NULL and cat.map is not NULL, then RandMatCat is used, which adjusts the sampling of features when categorical features have been one-of-K encoded. If a custom function is to be used, then it must return a matrix in sparse representation, in which each nonzero is an array of the form (row.index, column.index, value). See RandMat or RandMatCat for details.
- **paramList**: parameters in a named list to be used by FUN. If left unchanged, default values will be populated, see `defaults` for details.
- **min.parent**: the minimum splittable node size. A node size < min.parent will be a leaf node. (min.parent = 1)
- **trees**: the number of trees in the forest. (trees=500)
max.depth: the longest allowable distance from the root of a tree to a leaf node (i.e. the maximum allowed height for a tree). If max.depth=0, the tree will be allowed to grow without bound. (max.depth=0)

bagging: a non-zero value means a random sample of X will be used during tree creation. If replacement = FALSE the bagging value determines the percentage of samples to leave out-of-bag. If replacement = TRUE the non-zero bagging value is ignored. (bagging=.2)

replacement: if TRUE then n samples are chosen, with replacement, from X. (replacement=TRUE)

stratify: if TRUE then class sample proportions are maintained during the random sampling. Ignored if replacement = FALSE. (stratify = FALSE).

rank.transform: if TRUE then each feature is rank-transformed (i.e. smallest value becomes 1 and largest value becomes n) (rank.transform=FALSE)

store.oob: if TRUE then the samples omitted during the creation of a tree are stored as part of the tree. This is required to run OOBPredict(). (store.oob=FALSE)

store.impurity: if TRUE then the decrease in impurity is stored for each split. This is required to run FeatureImportance() (store.impurity=FALSE)

progress: if TRUE then a pipe is printed after each tree is created. This is useful for large datasets. (progress=FALSE)

rotate: if TRUE then the data matrix X is uniformly randomly rotated for each tree. (rotate=FALSE)

num.cores: the number of cores to use while training. If num.cores=0 then 1 less than the number of cores reported by the OS are used. (num.cores=0)

seed: the seed to use for training the forest. For two runs to match you must use the same seed for each run AND you must also use the same number of cores for each run. (seed=sample((0:100000000,1)))

cat.map: a list specifying which columns in X correspond to the same one-of-K encoded feature. Each element of cat.map is a numeric vector specifying the K column indices of X corresponding to the same categorical feature after one-of-K encoding. All one-of-K encoded features in X must come after the numeric features. The K encoded columns corresponding to the same categorical feature must be placed contiguously within X. The reason for specifying cat.map is to adjust for the fact that one-of-K encoding categorical features results in a dilution of numeric features, since a single categorical feature is expanded to K binary features. If cat.map = NULL, then RerF assumes all features are numeric (i.e. none of the features have been one-of-K encoded).

rFpack: boolean flag to determine whether to pack a random forest in order to improve prediction speed. This flag is only applicable when training a forest with the "rf" option. (rfPack = FALSE)

Value

forest
Examples

```r
### Train RefF on numeric data ###
library(ref)
forest <- RefF(as.matrix(iris[, 1:4]), iris[[5L]], num.cores = 1L)

### Train RefF on one-of-K encoded categorical data ###
df1 <- as.data.frame(Titanic)
nc <- ncol(df1)
df2 <- df1[NULL, -nc]
for (i in which(df1$Freq != 0L)) {
  df2 <- rbind(df2, df1[rep(i, df1$Freq[i]), -nc])
}

n <- nrow(df2) # number of observations
p <- ncol(df2) - 1L # number of features
num.categories <- apply(df2[, 1:p], 2, function(x) length(unique(x)))
p.enc <- sum(num.categories) # number of features after one-of-K encoding
X <- matrix(0, nrow = n, ncol = p.enc) # initialize training data matrix X
cat.map <- vector("list", p)
col.idx <- 0L
# one-of-K encode each categorical feature and store in X
for (j in 1:p) {
  cat.map[[j]] <- (col.idx + 1L):(col.idx + num.categories[j])
  # convert categorical feature to K dummy variables
  X[, cat.map[[j]]] <- dummies::dummy(df2[[j]])
  col.idx <- col.idx + num.categories[j]
}
Y <- df2$Survived

# specifying the cat.map in RefF allows training to
# be aware of which dummy variables correspond
# to the same categorical feature
forest <- RefF(X, Y, num.cores = 1L, cat.map = cat.map)
## Not run:
# takes longer than 5s to run.
# adding a continuous feature along with the categorical features
# must be prepended to the categorical features.
set.seed(1234)
xp <- runif(nrow(X))
Xp <- cbind(xp, X)
cat.map1 <- lapply(cat.map, function(x) x + 1)
forestW <- RefF(Xp, Y, num.cores = 1L, cat.map = cat.map1)
## End(Not run)

### Train a random rotation ensemble of CART decision trees (see Blaser and Fryzlewicz 2016) ###
forest <- RefF(as.matrix(iris[, 1:4]), iris[[5L]],
              num.cores = 1L,
              FUN = RandMatRF, paramList = list(p = 4, d = 2), rotate = TRUE)
```

```
RunFeatureImportance

Compute Feature Importance of a single RerF tree

Description
Computes feature importance of every unique feature used to make a split in a single tree.

Usage
RunFeatureImportance(tree, unique.projections)

Arguments
- tree: a single tree from a trained RerF model with argument store.impurity = TRUE.
- unique.projections: a list of all of the unique split projections used in the RerF model.

Value
feature.imp

RunFeatureImportanceBinary

Compute Feature Importance of a single RerF tree

Description
Computes feature importance of every unique feature used to make a split in a single tree.

Usage
RunFeatureImportanceBinary(tree, unique.projections)

Arguments
- tree: a single tree from a trained RerF model with argument store.impurity = TRUE.
- unique.projections: a list of all of the unique split projections used in the RerF model.

Value
feature.imp
RunFeatureImportanceCounts

Tabulate the unique feature combinations used in a single RerF tree

Description

Computes feature importance of every unique feature used to make a split in a single tree.

Usage

RunFeatureImportanceCounts(tree, unique.projections)

Arguments

tree
unique.projections

Value

feature.counts

Examples

library(rerf)
X <- iris[, -5]
Y <- iris[,5]
store.impurity <- TRUE
FUN <- RandMatBinary
forest <- RerF(X, Y, FUN = FUN, num.cores = 1L, store.impurity = store.impurity)
FeatureImportance(forest, num.cores = 1L)
RunOOB

Predict class labels on out-of-bag observations using a single tree.

Description

This is the base function called by OOBPredict.

Usage

RunOOB(X, tree)

Arguments

X           an n sample by d feature matrix (preferable) or data frame which was used to train the provided forest.

Tree       a tree from a forest returned by RerF.

Value

out prediction matrix used by OOBPredict

RunPredict

Predict class labels on a test set using a single tree.

Description

This is the base function called by Predict.

Usage

RunPredict(X, tree)

Arguments

X           an n sample by d feature matrix (preferable) or data frame which was used to train the provided forest.

Tree       a tree from a forest returned by RerF.

Value

predictions an n length vector of prediction based on the tree provided to this function
**RunPredictLeaf**

Calculate similarity using a single tree.

**Description**

This is the base function called by ComputeSimilarity.

**Usage**

RunPredictLeaf(X, tree)

**Arguments**

- **X**: an n sample by d feature matrix (preferable) or data frame which was used to train the provided forest.
- **tree**: a tree from a forest returned by RerF.

**Value**

similarity based on one tree

---

**StrCorr**

Compute tree strength and correlation

**Description**

Computes estimates of tree strength and correlation according to the definitions in Breiman’s 2001 Random Forests paper.

**Usage**

StrCorr(Yhats, Y)

**Arguments**

- **Yhats**: predicted class labels for each tree in a forest.
- **Y**: true class labels.

**Value**

scor
Examples

```r
library(reer)
trainIdx <- c(1:40, 51:90, 101:140)
X <- as.matrix(iris[, 1:4])
Y <- iris[[5]]
forest <- Reer(X[trainIdx, ], Y[trainIdx], num.cores = 1L)
predictions <- Predict(X[-trainIdx, ], forest, num.cores = 1L, aggregate.output = FALSE)
scor <- StrCorr(predictions, Y[-trainIdx])
```

---

**TwoMeansCut**

*Find minimizing Two Means Cut for Vector*

---

**Description**

Find minimizing Two Means Cut for Vector

**Usage**

`TwoMeansCut(X)`

**Arguments**

- `X` a one dimensional vector

**Value**

list containing minimizing cut point and corresponding sum of left and right variances.

---

**uniqueByEquivalenceClass**

*Remove unique projections that are equivalent due to a rotation of 180 degrees.*

---

**Description**

This function finds the projections that are equivalent via a 180 degree rotation and removes the duplicates.

**Usage**

`uniqueByEquivalenceClass(p, unique.projections)`
Arguments

\( p \)  
the number of features in the original data. This can be obtained from a forest object via `forest$params$paramList$p`.

unique.projections
a list of projections from intermediate steps of the `FeatureImportance` function.

Value

unique.projections a list which is a subset of the input.

See Also

`FeatureImportance`

Rerf  
**Unsupervised RerF forest Generator**

Description

Creates a decision forest based on an input matrix.

Usage

```
Rerf(X, trees = 100, min.parent = round(nrow(X)^0.5),
     max.depth = NA, mtry = ceiling(ncol(X)^0.5), sparsity = 1/nrow(X),
     normalizeData = TRUE, Progress = TRUE, splitCrit = "twomeans",
     LinearCombo = TRUE)
```

Arguments

\( X \)  
an n by d numeric matrix. The rows correspond to observations and columns correspond to features.

\( \text{trees} \)  
the number of trees in the forest. (trees=100)

\( \text{min.parent} \)  
the minimum splittable node size. A node size < min.parent will be a leaf node. (min.parent = round(nrow(X)^0.5))

\( \text{max.depth} \)  
the longest allowable distance from the root of a tree to a leaf node (i.e. the maximum allowed height for a tree). If max.depth=NA, the tree will be allowed to grow without bound. (max.depth=NA)

\( \text{mtry} \)  
the number of features to test at each node. (mtry=ceiling(ncol(X)^0.5))

\( \text{sparsity} \)  
a real number in \((0, 1)\) that specifies the distribution of non-zero elements in the random matrix. (sparsity=1/nrow(X))

\( \text{normalizeData} \)  
a logical value that determines if input data is normalized to values ranging from 0 to 1 prior to processing. (normalizeData=TRUE)

\( \text{Progress} \)  
boolean for printing progress.
splitCrit  split based on twomeans(splitCrit="twomeans") or BIC test(splitCrit="bicfast")
LinearCombo    logical that determines whether to use linear combination of features. (LinearCombo=TRUE).

Value
urerfStructure

Examples

### Train RerF on numeric data ###
library(rerf)
urerfStructure <- Urerf(as.matrix(iris[, 1:4]))
urerfStructure.bic <- Urerf(as.matrix(iris[, 1:4]), splitCrit = 'bicfast')

dissimilarityMatrix <- hclust(as.dist(1 - urerfStructure$similarityMatrix), method = "mcquitty")
clusters <- cutree(dissimilarityMatrix, k = 3)
Index

*Topic **datasets**
  mnist, 11

BICCutFast, 3
BICCutMclust, 3
BuildTree, 4

checkInputMatrix, 5
ComputeSimilarity, 5

defaults, 4, 24

FeatureImportance, 6, 32
flipWeights, 7

getFeatures, 8
getWeights, 8
GrowUnsupervisedForest, 9

makeA, 10
makeAB, 10

mnist, 11

OoBPredict, 12

PackForest, 13
PackPredict, 13
Predict, 14
PrintTree, 15

RandMatBinary, 16
RandMatContinuous, 17
RandMatCustom, 17
RandMatFRC, 18
RandMatFRCN, 19
RandMatImageControl, 20
RandMatImagePatch, 21
RandMatPoisson, 22
RandMatRF, 22
RandMatTSPatch, 23
RerF, 16–23, 24

rpois, 22
RunFeatureImportance, 27
RunFeatureImportanceBinary, 27
RunFeatureImportanceCounts, 28
RunOOB, 29
RunPredict, 29
RunPredictLeaf, 30

StrCorr, 30

TwoMeansCut, 31

uniqueByEquivalenceClass, 31

Uerf, 32