# Package 'sortinghat'

August 29, 2016

**Title** sortinghat

**Version** 0.1

**Date** 2013-12-07

**Author** John A. Ramey

**Maintainer** John A. Ramey <johnramey@gmail.com>

**Description** sortinghat is a classification framework to streamline the evaluation of classifiers (classification models and algorithms) and seeks to determine the best classifiers on a variety of simulated and benchmark data sets. Several error-rate estimators are included to evaluate the performance of a classifier. This package is intended to complement the well-known 'caret' package.

**Depends** R (>= 3.0.1)

**Imports** MASS, bdsmatrix, mvtnorm

**Suggests** testthat

**License** MIT + file LICENSE

**LazyData** true

**URL** http://github.com/ramhiser/sortinghat

**Collate** 'check-arguments.r' 'sortinghat-package.r' 'helper-partition-data.r' 'helper-which-min.r' 'simdata-normal.r' 'simdata-uniform.r' 'covariance-matrices.r' 'helpers.r' 'simdata-t.r' 'simdata-contaminated-normal.r' 'simdata-guo.r' 'simdata-friedman.r' 'simdata-wrapper.r' 'errorest-632.r' 'errorest-632plus.r' 'errorest-apparent.r' 'errorest-bcv.r' 'errorest-boot.r' 'errorest-cv.r' 'errorest-loo-boot.r' 'errorest-wrapper.r'

**NeedsCompilation** no

**Repository** CRAN

**Date/Publication** 2013-12-07 10:31:18

# R topics documented:



---

`all_equal` *Function to check whether all elements in a numeric vector are equal within some tolerance*

---

## Description

Function to check whether all elements in a numeric vector are equal within some tolerance

## Usage

```
all_equal(x, tol = .Machine$double.eps^0.5)
```

## Arguments

| | |
|---|---|
| `x` | numeric vector |
| `tol` | tolerance value |

## Value

logical value

## Examples

```
# Returns TRUE
all_equal(c(3, 3, 3))
# Returns FALSE
all_equal(c(3, 3, 2))
```

---

`check_arguments` *Checks the arguments passed to the error rate estimator functions.*

---

## Description

This function is a helper function that checks the arguments passed to make sure the input is valid and consistent across all error rate estimators.

## Usage

```
check_arguments(x, y, train, classify)
```

## Arguments

| | |
|---|---|
| `x` | a matrix of n observations and p features |
| `y` | a vector of n class labels |
| `train` | a function that builds the classifier (See details) |
| `classify` | a function that classified observations from the constructed classifier from `train`. (See details.) |

## Details

We expect that the first two arguments of the classifier function given in `train` are `x` and `y`, corresponding to the data matrix and the vector of their labels. Additional arguments can be passed to the `train` function. The returned object should be a classifier that will be passed to the function given in the `classify` argument.

## Value

`TRUE` invisibly if no errors are encountered.

`cov_autocorrelation` *Constructs a p-dimensional covariance matrix with an autocorrelation (autoregressive) structure.*

## Description

This function generates a $p \times p$ autocorrelated covariance matrix with autocorrelation parameter `rho`. The variance `sigma2` is constant for each feature and defaulted to 1.

## Usage

```
cov_autocorrelation(p = 100, rho = 0.9, sigma2 = 1)
```

## Arguments

| | |
|---|---|
| `p` | the size of the covariance matrix |
| `rho` | the autocorrelation value |
| `sigma2` | the variance of each feature |

## Details

The autocorrelated covariance matrix is defined as: The $(i, j)$th entry of the autocorrelated covariance matrix is defined as: $\rho^{|i-j|}$.

The value of `rho` must be such that $|\rho| < 1$ to ensure that the covariance matrix is positive definite.

## Value

autocorrelated covariance matrix

`cov_block_autocorrelation` *Generates a p-dimensional block-diagonal covariance matrix with autocorrelated blocks.*

## Description

This function generates a $p \times p$ covariance matrix with autocorrelated blocks. The autocorrelation parameter is `rho`. There are `num_blocks` blocks each with size, `block_size`. The variance, `sigma2`, is constant for each feature and defaulted to 1.

## Usage

```
cov_block_autocorrelation(num_blocks, block_size, rho,
  sigma2 = 1)
```

## Arguments

| | |
|---|---|
| num_blocks | the number of blocks in the covariance matrix |
| block_size | the size of each square block within the covariance matrix |
| rho | the autocorrelation parameter. Must be less than 1 in absolute value. |
| sigma2 | the variance of each feature |

## Details

The autocorrelated covariance matrix is defined as:

$$\Sigma = \Sigma^{(\rho)} \oplus \Sigma^{(-\rho)} \oplus \ldots \oplus \Sigma^{(\rho)},$$

where $\oplus$ denotes the direct sum and the $(i, j)$th entry of $\Sigma^{(\rho)}$ is

$$\Sigma_{ij}^{(\rho)} = \{\rho^{|i-j|}\}.$$

The matrix $\Sigma^{(\rho)}$ is the autocorrelated block discussed above.

The value of `rho` must be such that $|\rho| < 1$ to ensure that the covariance matrix is positive definite.

The size of the resulting matrix is $p \times p$, where `p = num_blocks * block_size`.

The block-diagonal covariance matrix with autocorrelated blocks was popularized by Guo et al. (2007) for studying classification of high-dimensional data.

## Value

autocorrelated covariance matrix

## References

Guo, Y., Hastie, T., & Tibshirani, R. (2007). "Regularized linear discriminant analysis and its application in microarrays," Biostatistics, 8, 1, 86-100.

---

`cov_intraclass` *Constructs a p-dimensional intraclass covariance matrix.*

---

## Description

We define a $p \times p$ intraclass covariance (correlation) matrix to be $\Sigma = \sigma^2(1-\rho)J_p + \rho I_p$, where $-(p-1)^{-1} < \rho < 1$, $I_p$ is the $p \times p$ identity matrix, and $J_p$ denotes the $p \times p$ matrix of ones.

## Usage

```
cov_intraclass(p, rho, sigma2 = 1)
```

### Arguments

| | |
|---|---|
| p | the size of the covariance matrix |
| rho | the intraclass covariance (correlation) constant |
| sigma2 | the coefficient of the intraclass covariance matrix |

### Value

an intraclass covariance matrix of size $p \times p$

---

## cv_partition *Partitions data for cross-validation.*

---

### Description

For a vector of training labels, we return a list of cross-validation folds, where each fold has the indices of the observations to leave out in the fold. In terms of classification error rate estimation, one can think of a fold as a the observations to hold out as a test sample set.

### Usage

```
cv_partition(y, num_folds = 10, hold_out = NULL,
  seed = NULL)
```

### Arguments

| | |
|---|---|
| y | a vector of class labels to partition |
| num_folds | the number of cross-validation folds. Ignored if hold_out is not NULL. See Details. |
| hold_out | the hold-out size for cross-validation. See Details. |
| seed | optional random number seed for splitting the data for cross-validation |

### Details

Either the hold_out size or num_folds can be specified. The number of folds defaults to 10, but if the hold_out size is specified, then num_folds is ignored.

We partition the vector y based on its length, which we treat as the sample size, n. If an object other than a vector is used in y, its length can yield unexpected results. For example, the output of length(diag(3)) is 9.

### Value

list the indices of the training and test observations for each fold.

## Examples

```
library(MASS)
# The following three calls to \code{cv_partition} yield the same partitions.
set.seed(42)
cv_partition(iris$Species)
cv_partition(iris$Species, num_folds = 10, seed = 42)
cv_partition(iris$Species, hold_out = 15, seed = 42)
```

---

`errorest` *Wrapper function to estimate the error rate of a classifier*

---

## Description

We provide a wrapper function to estimate the error rate of a classifier using any of the following estimators:#'

`errorest_cv`: Cross-validation Error Rate

`errorest_boot`: Bootstrap Error Rate

`errorest_632`: .632 Estimator from Efron (1983)

`errorest_632plus`: .632+ Estimator from Efron and Tibshirani (1997)

`errorest_bcv`: Bootstrap Cross-validation from Fu et al. (2005)

`errorest_loo_boot`: Leave-One-Out Bootstrap Error Rate

`errorest_apparent`: Apparent Error Rate

## Usage

```
errorest(x, y,
  estimator = c("cv", "boot", "632", "632+", "bcv", "loo-boot", "apparent"),
  train, classify, ...)
```

## Arguments

| | |
|---|---|
| `x` | a matrix of n observations and p features |
| `y` | a vector of n class labels. (Must to be a 'factor'.) |
| `estimator` | the estimator used to compute the error rate |
| `train` | a function that builds the classifier. (See details.) |
| `classify` | a function that classifies observations from the constructed classifier from `train`. (See Details.) |
| `...` | additional arguments passed to the error-rate estimation code |

## Details

This wrapper function provides a common means to estimate classification error rates and is useful for simulation studies where multiple error-rate estimators are being considered.

For details about an individual error-rate estimator, please see its respective documentation.

## Value

an estimate of the classifier's error rate

## Examples

```
require('MASS')
iris_x <- data.matrix(iris[, -5])
iris_y <- iris[, 5]

# Because the \code{classify} function returns multiples objects in a list,
# we provide a wrapper function that returns only the class labels.
lda_wrapper <- function(object, newdata) { predict(object, newdata)$class }

# Cross-Validation (default)
errorest(x = iris_x, y = iris_y, train = MASS:::lda, classify = lda_wrapper)

# .632
errorest(x = iris_x, y = iris_y, estimator = "632", train = MASS:::lda,
         classify = lda_wrapper)

# Bootstrap Error Rate
# The argument 'num_bootstraps' is passed on to 'errorest_boot'
errorest(x = iris_x, y = iris_y, estimator = "boot", train = MASS:::lda,
         classify = lda_wrapper, num_bootstraps = 42)
```

---

`errorest_632` *Calculates the .632 Error Rate for a specified classifier given a data set.*

---

## Description

For a given data matrix and its corresponding vector of labels, we calculate the .632 error rate from Efron (1983) for a given classifier.

## Usage

```
errorest_632(x, y, train, classify, num_bootstraps = 50,
  apparent = NULL, loo_boot = NULL, ...)
```

## Arguments

| | |
|---|---|
| `x` | a matrix of n observations (rows) and p features (columns) |
| `y` | a vector of n class labels |
| `train` | a function that builds the classifier. (See details.) |
| `classify` | a function that classifies observations from the constructed classifier from `train`. (See details.) |
| `num_bootstraps` | the number of bootstrap replications |

| | |
|---|---|
| apparent | the apparent error rate for the given classifier. If NULL, this argument is ignored. See Details. |
| loo_boot | the leave-one-out bootstrap error rate for the given classifier. If NULL, this argument is ignored. See Details. |
| ... | additional arguments passed to the function specified in train. |

## Details

To calculate the .632 error rate, we compute the leave-one-out bootstrap (LOO-Boot) error rate and the apparent error rate (AER). Then, we compute a convex combination of these two error rates estimators.

To calculate the AER, we use the errorest_apparent function. Similarly, we use the errorest_loo_boot function to calculate the (LOO-Boot error rate. In some cases (e.g., simulation study) one, if not both, of these error rate estimators might already be computed. Hence, we allow the user to provide these values if they are already computed; by default, the arguments are NULL to indicate that they are ignored.

For the given classifier, two functions must be provided 1. to train the classifier and 2. to classify unlabeled observations. The training function is provided as train and the classification function as classify.

We expect that the first two arguments of the train function are x and y, corresponding to the data matrix and the vector of their labels, respectively. Additional arguments can be passed to the train function.

We stay with the usual R convention for the classify function. We expect that this function takes two arguments: 1. an object argument which contains the trained classifier returned from the function specified in train; and 2. a newdata argument which contains a matrix of observations to be classified – the matrix should have rows corresponding to the individual observations and columns corresponding to the features (covariates). For an example, see lda.

## Value

the 632 error rate estimate

## References

Efron, Bradley (1983), "Estimating the Error Rate of a Prediction Rule: Improvement on Cross-Validation," Journal of American Statistical Association, 78, 382, 316-331.

## Examples

```
require('MASS')
iris_x <- data.matrix(iris[, -5])
iris_y <- iris[, 5]

# Because the \code{classify} function returns multiples objects in a list,
# we provide a wrapper function that returns only the class labels.
lda_wrapper <- function(object, newdata) { predict(object, newdata)$class }

# We compute the apparent and LOO-Boot error rates up front to demonstrate
```

```
# that they can be computed before the \code{errorest_632} function is called.

set.seed(42)
apparent <- errorest_apparent(x = iris_x, y = iris_y, train = MASS:::lda,
                              classify = lda_wrapper)
set.seed(42)
loo_boot <- errorest_loo_boot(x = iris_x, y = iris_y, train = MASS:::lda,
                              classify = lda_wrapper)

# Each of the following 3 calls should result in the same error rate.
# 1. The apparent error rate is provided, while the LOO-Boot must be computed.
set.seed(42)
errorest_632(x = iris_x, y = iris_y, train = MASS:::lda, classify = lda_wrapper,
             apparent = apparent)
# 2. The LOO-Boot error rate is provided, while the apparent must be computed.
set.seed(42)
errorest_632(x = iris_x, y = iris_y, train = MASS:::lda, classify = lda_wrapper,
             loo_boot = loo_boot)
# 3. Both error rates are provided, so the calculation is quick.
errorest_632(x = iris_x, y = iris_y, train = MASS:::lda, classify = lda_wrapper,
             apparent = apparent, loo_boot = loo_boot)

# In each case the output is: 0.02194132
```

---

`errorest_632plus` *Calculates the .632+ Error Rate for a specified classifier given a data set.*

---

## Description

For a given data matrix and its corresponding vector of labels, we calculate the .632+ error rate from Efron and Tibshirani (1997) for a given classifier.

## Usage

```
errorest_632plus(x, y, train, classify,
  num_bootstraps = 50, apparent = NULL, loo_boot = NULL,
  ...)
```

## Arguments

| | |
|---|---|
| `x` | a matrix of n observations (rows) and p features (columns) |
| `y` | a vector of n class labels |
| `train` | a function that builds the classifier. (See details.) |
| `classify` | a function that classifies observations from the constructed classifier from `train`. (See details.) |
| `num_bootstraps` | the number of bootstrap replications |

| | |
|---|---|
| apparent | the apparent error rate for the given classifier. If NULL, this argument is ignored. See Details. |
| loo_boot | the leave-one-out bootstrap error rate for the given classifier. If NULL, this argument is ignored. See Details. |
| ... | additional arguments passed to the function specified in train. |

## Details

To calculate the .632+ error rate, we compute the leave-one-out (LOO) bootstrap error rate and the apparent error rate. Then, we compute the 'relative overfitting rate' based on these values. Next, we compute the 'no-information error rate'. Finally, we compute the .632+ error rate estimator from these values.

The 'no-information error rate', $\gamma$, is the error rate of the classifier if the error rate if the feature vectors and the class labels were independent. For $K$ classes, we can estimate $\gamma$ by

$$\hat{\gamma} = \sum_{k=1}^{K} p_k * (1 - q_k)$$

, where $p_k$ is the observed proportion of responses for class $k$ and $q_k$ is the proportion of observations classified as class $k$.

To calculate the apparent error rate, we use the errorest_apparent function. Similarly, to calculate the LOO bootstrap (LOO-Boot) error rate, we use the errorest_loo_boot function. In some cases (e.g. simulation study) one, if not both, of these error rate estimators might already be computed. Hence, we allow the user to provide these values if they are already computed; by default, the arguments are NULL to indicate that they are unavailable.

We expect that the first two arguments of the classifier function given in train are x and y, corresponding to the data matrix and the vector of their labels. Additional arguments can be passed to the train function. The returned object should be a classifier that will be passed to the function given in the classify argument.

We stay with the usual R convention for the classify function. We expect that this function takes two arguments: 1. an object argument which contains the trained classifier returned from the function specified in train; and 2. a newdata argument which contains a matrix of observations to be classified – the matrix should have rows corresponding to the individual observations and columns corresponding to the features (covariates).

## Value

the 632+ error rate estimate

## References

Efron, Bradley and Tibshirani, Robert (1997), "Improvements on Cross-Validation: The .632+ Bootstrap Method," Journal of American Statistical Association, 92, 438, 548-560.

## Examples

```
require('MASS')
iris_x <- data.matrix(iris[, -5])
iris_y <- iris[, 5]

# Because the \code{classify} function returns multiples objects in a list,
# we provide a wrapper function that returns only the class labels.
lda_wrapper <- function(object, newdata) { predict(object, newdata)$class }
set.seed(42)

# We compute the apparent and LOO-Boot error rates up front to demonstrate
# that they can be computed before the \code{errorest_632plus} function is called.

set.seed(42)
apparent <- errorest_apparent(x = iris_x, y = iris_y, train = MASS:::lda,
                              classify = lda_wrapper)
set.seed(42)
loo_boot <- errorest_loo_boot(x = iris_x, y = iris_y, train = MASS:::lda,
                              classify = lda_wrapper)

# Each of the following 3 calls should result in the same error rate.
# 1. The apparent error rate is provided, while the LOO-Boot must be computed.
set.seed(42)
errorest_632plus(x = iris_x, y = iris_y, train = MASS:::lda,
                 classify = lda_wrapper, apparent = apparent)
# 2. The LOO-Boot error rate is provided, while the apparent must be computed.
set.seed(42)
errorest_632plus(x = iris_x, y = iris_y, train = MASS:::lda,
                 classify = lda_wrapper, loo_boot = loo_boot)
# 3. Both error rates are provided, so the calculation is quick.
errorest_632plus(x = iris_x, y = iris_y, train = MASS:::lda,
                 classify = lda_wrapper, apparent = apparent,
                 loo_boot = loo_boot)

# In each case the output is: 0.02194472
```

---

`errorest_apparent` *Calculates the Apparent Error Rate for a specified classifier given a data set.*

---

## Description

For a given data matrix and its corresponding vector of labels, we calculate the apparent error rate (AER) for a given classifier.

## Usage

```
errorest_apparent(x, y, train, classify, ...)
```

## Arguments

| | |
|---|---|
| x | a matrix of n observations (rows) and p features (columns) |
| y | a vector of n class labels |
| train | a function that builds the classifier. (See details.) |
| classify | a function that classifies observations from the classifier constructed by train. (See details.) |
| ... | additional arguments passed to the function specified in train. |

## Details

The AER simply uses the data set as both the training and test data sets. The AER is well known to be biased downward in that it underestimates the true error rate of the classifier.

For the given classifier, two functions must be provided 1. to train the classifier and 2. to classify unlabeled observations. The training function is provided as train and the classification function as classify.

We expect that the first two arguments of the train function are x and y, corresponding to the data matrix and the vector of their labels, respectively. Additional arguments can be passed to the train function.

We stay with the usual R convention for the classify function. We expect that this function takes two arguments: 1. an object argument which contains the trained classifier returned from the function specified in train; and 2. a newdata argument which contains a matrix of observations to be classified – the matrix should have rows corresponding to the individual observations and columns corresponding to the features (covariates). For an example, see lda.

## Value

object of class errorest. The object is a named list that contains the following elements:

the calculated apparent error-rate estimate

## Examples

```
require('MASS')
iris_x <- data.matrix(iris[, -5])
iris_y <- iris[, 5]

# Because the \code{classify} function returns multiples objects in a list,
# we provide a wrapper function that returns only the class labels.
lda_wrapper <- function(object, newdata) { predict(object, newdata)$class }
errorest_apparent(x = iris_x, y = iris_y, train = MASS:::lda, classify = lda_wrapper)
# Output: 0.02

# The following code is equivalent for this example:
lda_out <- MASS:::lda(x = iris_x, grouping = iris_y)
lda_classifications <- predict(lda_out, newdata = iris_x)$class
mean(lda_classifications != iris_y)
# Output: 0.02
```

errorest_bcv *Calculates the Bootstrap Cross-Validation (BCV) Error Rate Estimator for a specified classifier given a data set.*

## Description

For a given data matrix and its corresponding vector of labels, we calculate the bootstrap cross-validation (BCV) error rate from Fu, Carroll, and Wang (2005) for a given classifier.

## Usage

```
errorest_bcv(x, y, train, classify, num_bootstraps = 50,
  num_folds = 10, hold_out = NULL, ...)
```

## Arguments

| | |
|---|---|
| x | a matrix of n observations (rows) and p features (columns) |
| y | a vector of n class labels |
| train | a function that builds the classifier. (See details.) |
| classify | a function that classifies observations from the constructed classifier from train. (See details.) |
| num_bootstraps | the number of bootstrap replications |
| num_folds | the number of cross-validation folds. Ignored if hold_out is not NULL. See Details. |
| hold_out | the hold-out size for cross-validation. See Details. |
| ... | additional arguments passed to the function specified in train. |

## Details

To calculate the BCV error rate, we sample from the data with replacement to obtain a bootstrapped training data set. We then compute a cross-validation error rate with the given classifier (given in train) on the bootstrapped training data set. We repeat this process num_bootstraps times to obtain a set of bootstrapped cross-validation error rates. We report the average of these error rates. The errorest_cv function is used to compute the cross-validation (CV) error rate estimator for each bootstrap iteration.

Fu et al. (2005) note that the BCV method works well because it is a bagging classification error. Furthermore, consider the leave-one-out (LOO) error rate estimator. For small sample sizes, the data are sparse, so that the left out observation has a high probability of being far in distance from the remaining training data set. Hence, the LOO error rate estimator yields a large variance for small data sets.

Rather than partitioning the observations into folds, an alternative convention is to specify the 'hold-out' size for each test data set. Note that this convention is equivalent to the notion of folds. We allow the user to specify either option with the hold_out and num_folds arguments. The num_folds argument is the default option but is ignored if the hold_out argument is specified (i.e. is not NULL).

We expect that the first two arguments of the classifier function given in `train` are `x` and `y`, corresponding to the data matrix and the vector of their labels. Additional arguments can be passed to the `train` function. The returned object should be a classifier that will be passed to the function given in the `classify` argument.

We stay with the usual R convention for the `classify` function. We expect that this function takes two arguments: 1. an `object` argument which contains the trained classifier returned from the function specified in `train`; and 2. a `newdata` argument which contains a matrix of observations to be classified – the matrix should have rows corresponding to the individual observations and columns corresponding to the features (covariates).

## Value

the BCV error rate estimate

## References

Fu, W.J., Carroll, R.J., and Wang, S. (2005), "Estimating misclassification error with small samples via bootstrap cross-validation," Bioinformatics, vol. 21, no. 9, pp. 1979-1986.

## Examples

```
require('MASS')
iris_x <- data.matrix(iris[, -5])
iris_y <- iris[, 5]

# Because the \code{classify} function returns multiples objects in a list,
# we provide a wrapper function that returns only the class labels.
lda_wrapper <- function(object, newdata) { predict(object, newdata)$class }
set.seed(42)
errorest_bcv(x = iris_x, y = iris_y, train = MASS:::lda,
             classify = lda_wrapper)
# Output: 0.02213333
```

---

# errorest_boot *Calculates the Bootstrap Error Rate for a specified classifier given a data set.*

---

## Description

For a given data matrix and its corresponding vector of labels, we calculate the bootstrap error rate for a given classifier.

## Usage

```
errorest_boot(x, y, train, classify, num_bootstraps = 50,
  ...)
```

## Arguments

| | |
|---|---|
| x | a matrix of n observations (rows) and p features (columns) |
| y | a vector of n class labels |
| train | a function that builds the classifier. (See details.) |
| classify | a function that classifies observations from the constructed classifier from train. (See details.) |
| num_bootstraps | the number of bootstrap replications |
| ... | additional arguments passed to the function specified in train. |

## Details

To calculate the bootstrap error rate, we sample from the data with replacement to obtain a bootstrapped training data set. We then train the given classifier (given in train) on the bootstrapped training data set and classify the original data set given in the matrix x. Then we calculate the proportion of misclassified observations, based on the true labels given in y, to obtain a single bootstrap error rate. We repeat this process num_bootstraps times and report the average of the bootstrap error rates.

For the given classifier, two functions must be provided 1. to train the classifier and 2. to classify unlabeled observations. The training function is provided as train and the classification function as classify.

We expect that the first two arguments of the train function are x and y, corresponding to the data matrix and the vector of their labels, respectively. Additional arguments can be passed to the train function.

We stay with the usual R convention for the classify function. We expect that this function takes two arguments: 1. an object argument which contains the trained classifier returned from the function specified in train; and 2. a newdata argument which contains a matrix of observations to be classified – the matrix should have rows corresponding to the individual observations and columns corresponding to the features (covariates). For an example, see lda.

## Value

the bootstrapped error rate estimate

## Examples

```
require('MASS')
iris_x <- data.matrix(iris[, -5])
iris_y <- iris[, 5]

# Because the \code{classify} function returns multiples objects in a list,
# we provide a wrapper function that returns only the class labels.
lda_wrapper <- function(object, newdata) { predict(object, newdata)$class }
set.seed(42)
errorest_boot(x = iris_x, y = iris_y, train = MASS:::lda, classify = lda_wrapper)
# Output: 0.0228
```

errorest_cv *Calculates the Cross-Validation Error Rate for a specified classifier given a data set.*

## Description

For a given data matrix and its corresponding vector of labels, we calculate the cross-validation (CV) error rate for a given classifier.

## Usage

```
errorest_cv(x, y, train, classify, num_folds = 10,
  hold_out = NULL, ...)
```

## Arguments

| | |
|---|---|
| x | a matrix of n observations (rows) and p features (columns) |
| y | a vector of n class labels |
| train | a function that builds the classifier. (See details.) |
| classify | a function that classifies observations from the constructed classifier from train. (See details.) |
| num_folds | the number of cross-validation folds. Ignored if hold_out is not NULL. See Details. |
| hold_out | the hold-out size for cross-validation. See Details. |
| ... | additional arguments passed to the function specified in train. |

## Details

To calculate the CV error rate, we partition the data set into 'folds'. For each fold, we consider the observations within the fold as a test data set, while the remaining observations are considered as a training data set. We then calculate the number of misclassified observations within the fold. The CV error rate is the proportion of misclassified observations across all folds.

Rather than partitioning the observations into folds, an alternative convention is to specify the 'hold-out' size for each test data set. Note that this convention is equivalent to the notion of folds. We allow the user to specify either option with the hold_out and num_folds arguments. The num_folds argument is the default option but is ignored if the hold_out argument is specified (i.e. is not NULL).

For the given classifier, two functions must be provided 1. to train the classifier and 2. to classify unlabeled observations. The training function is provided as train and the classification function as classify.

We expect that the first two arguments of the train function are x and y, corresponding to the data matrix and the vector of their labels, respectively. Additional arguments can be passed to the train function.

We stay with the usual R convention for the classify function. We expect that this function takes two arguments: 1. an object argument which contains the trained classifier returned from the

function specified in `train`; and 2. a `newdata` argument which contains a matrix of observations to be classified – the matrix should have rows corresponding to the individual observations and columns corresponding to the features (covariates). For an example, see `lda`.

## Value

the calculated CV error-rate estimate

## Examples

```
require('MASS')
iris_x <- data.matrix(iris[, -5])
iris_y <- iris[, 5]

# Because the \code{classify} function returns multiples objects in a list,
# we provide a wrapper function that returns only the class labels.
lda_wrapper <- function(object, newdata) { predict(object, newdata)$class }

set.seed(42)
errorest_cv(x = iris_x, y = iris_y, train = MASS:::lda, classify = lda_wrapper)
# Output: 0.02666667
```

---

`errorest_loo_boot` *Calculates the Leave-One-Out (LOO) Bootstrap Error Rate for a specified classifier given a data set.*

---

## Description

For a given data matrix and its corresponding vector of labels, we calculate the LOO bootstrap (LOO-Boot) error rate for a given classifier.

## Usage

```
errorest_loo_boot(x, y, train, classify,
  num_bootstraps = 50, ...)
```

## Arguments

| | |
|---|---|
| `x` | a matrix of n observations (rows) and p features (columns) |
| `y` | a vector of n class labels |
| `train` | a function that builds the classifier. (See details.) |
| `classify` | a function that classifies observations from the constructed classifier from `train`. (See details.) |
| `num_bootstraps` | the number of bootstrap replications |
| `...` | additional arguments passed to the function specified in `train`. |

## Details

To calculate the LOO-Boot error rate, we sample from the data with replacement to obtain a bootstrapped training data set. We then train the given classifier (given in `train`) on the bootstrapped training data set and classify the observations from the original data set given in the matrix `x` that are not contained in the current bootstrapped training data set. We repeat this process `num_bootstraps` times. Then, for each observation in the original data set, we compute the proportion of times the observation was misclassified, based on the true labels given in `y`. We report the average of these proportions as the LOO-Boot error rate.

For the given classifier, two functions must be provided 1. to train the classifier and 2. to classify unlabeled observations. The training function is provided as `train` and the classification function as `classify`.

We expect that the first two arguments of the `train` function are `x` and `y`, corresponding to the data matrix and the vector of their labels, respectively. Additional arguments can be passed to the `train` function.

We stay with the usual R convention for the `classify` function. We expect that this function takes two arguments: 1. an `object` argument which contains the trained classifier returned from the function specified in `train`; and 2. a `newdata` argument which contains a matrix of observations to be classified – the matrix should have rows corresponding to the individual observations and columns corresponding to the features (covariates). For an example, see `lda`.

## Value

the LOO-Boot error rate estimate

## Examples

```
require('MASS')
iris_x <- data.matrix(iris[, -5])
iris_y <- iris[, 5]

# Because the \code{classify} function returns multiples objects in a list,
# we provide a wrapper function that returns only the class labels.
lda_wrapper <- function(object, newdata) { predict(object, newdata)$class }
set.seed(42)
errorest_loo_boot(x = iris_x, y = iris_y, train = MASS:::lda, classify = lda_wrapper)
# Output: 0.02307171
```

---

# partition_data *Helper function that partitions a data set into training and test data sets.*

---

## Description

The function randomly partitions a data set into training and test data sets with a specified percentage of observations assigned to the training data set. The user can optionally preserve the proportions of the original data set.

## Usage

```
partition_data(x, y, split_pct = 2/3,
  preserve_proportions = FALSE)
```

## Arguments

| | |
|---|---|
| x | a matrix of n observations (rows) and p features (columns) |
| y | a vector of n class labels |
| split_pct | the percentage of observations that will be randomly assigned to the training data set. The remainder of the observations will be assigned to the test data set. |
| preserve_proportions | logical value. If TRUE, the training and test data sets will be constructed so that the original proportions are preserved. |

## Details

A named list is returned with the training and test data sets.

## Value

named list containing the training and test data sets:

- train_x: matrix of the training observations
- train_y: vector of the training labels (coerced to factors).
- test_x: matrix of the test observations
- test_y: vector of the test labels (coerced to factors).

## Examples

```
require('MASS')
x <- iris[, -5]
y <- iris[, 5]
set.seed(42)
data <- partition_data(x = x, y = y)
table(data$train_y)
table(data$test_y)

data <- partition_data(x = x, y = y, preserve_proportions = TRUE)
table(data$train_y)
table(data$test_y)
```

`simdata` *Wrapper function to generate data from a variety of data-generating families for classification studies.*

## Description

We provide a wrapper function to generate random variates from any of the following data-generating families:

`simdata_normal`**:** Multivariate normal

`simdata_t`**:** Multivariate Student's t

`simdata_uniform`**:** Multivariate uniform

`simdata_contaminated`**:** Multivariate contaminated normal

`simdata_guo`**:** Simulation configuration from Guo et al. (2007)

`simdata_friedman`**:** Six simulation configurations from Friedman (1989)

## Usage

```
simdata(family = c("uniform", "normal", "t", "contaminated", "guo", "friedman"),
  ...)
```

## Arguments

| | |
|---|---|
| `family` | the family of distributions from which to generate data |
| `...` | optional arguments that are passed to the data-generating function |

## Details

This wrapper function is useful for simulation studies, where the performance of supervised and unsupervised learning methods and algorithms are evaluated. For each data-generating model, we generate $n_k$ observations $(k = 1, \ldots, K)$ from each of $K$ multivariate distributions.

Each family returns a list containing a matrix of the multivariate observations generated as well as the class labels for each observation.

For details about an individual data-generating family, please see its respective documentation.

## Value

named list containing:

**x:** A matrix whose rows are the observations generated and whose columns are the `p` features (variables)

**y:** A vector denoting the population from which the observation in each row was generated.

## Examples

```
data_normal <- simdata(family = "normal", n = c(10, 20), mean = c(0, 1), cov = diag(2), seed = 42)
data_uniform <- simdata(family = "uniform", delta = 2, seed = 42)
data_friedman <- simdata(family = "friedman", experiment = 4, seed = 42)
```

---

`simdata_contaminated` *Generates random variates from K multivariate contaminated normal populations.*

---

## Description

We generate $n_k$ observations $(k = 1, \ldots, K)$ from each of $K$ multivariate contaminated normal distributions. Let $N_p(\mu, \Sigma)$ denote the p-dimensional multivariate normal distribution with mean vector $\mu$ and positive-definite covariance matrix $\Sigma$. Then, let the $k$th population have a $p$-dimensional multivariate contaminated normal distribution:

## Usage

```
simdata_contaminated(n, mean, cov, epsilon = rep(0, K),
  kappa = rep(1, K), seed = NULL)
```

## Arguments

| | |
|---|---|
| `n` | a vector (of length K) of the sample sizes for each population |
| `mean` | a vector or a list (of length K) of mean vectors |
| `cov` | a symmetric matrix or a list (of length K) of symmetric covariance matrices. |
| `epsilon` | a vector (of length K) indicating the probability of sampling a contaminated population (i.e., outlier) for each population |
| `kappa` | a vector (of length K) that determines the amount of scale contamination for each population |
| `seed` | seed for random number generation (If `NULL`, does not set seed) |

## Details

$$(1 - \epsilon_k)N_p(\mu_k, \Sigma_k) + \epsilon_k N_p(\mu_k, \kappa_k \Sigma_k),$$

where $\epsilon_k \in [0, 1]$ is the probability of sampling from a contaminated population (i.e., outlier) and $\kappa_k \geq 1$ determines the amount of scale contamination. The contaminated normal distribution can be viewed as a mixture of two multivariate normal random distributions, where the second has a scaled covariance matrix, which can introduce extreme outliers for sufficiently large $\kappa_k$.

The number of populations, `K`, is determined from the length of the vector of sample sizes, coden. The mean vectors and covariance matrices each can be given in a list of length `K`. If one covariance matrix is given (as a matrix or a list having 1 element), then all populations share this common covariance matrix. The same logic applies to population means.

The contamination probabilities in epsilon can be given as a numeric vector or a single value, in which case the degrees of freedom is replicated K times. The same idea applies to the scale contamination in the kappa argument.

By default, epsilon is a vector of zeros, and kappa is a vector of ones. Hence, no contamination is applied by default.

## Value

named list containing:

**x:** A matrix whose rows are the observations generated and whose columns are the p features (variables)

**y:** A vector denoting the population from which the observation in each row was generated.

## Examples

```
# Generates 10 observations from each of two multivariate contaminated normal
# populations with equal covariance matrices. Each population has a
# contamination probability of 0.05 and scale contamination of 10.
mean_list <- list(c(1, 0), c(0, 1))
cov_identity <- diag(2)
data <- simdata_contaminated(n = c(10, 10), mean = mean_list,
                             cov = cov_identity, epsilon = 0.05, kappa = 10,
                             seed = 42)
dim(data$x)
table(data$y)

# Generates 10 observations from each of three multivariate contaminated
# normal populations with unequal covariance matrices. The contamination
# probabilities and scales differ for each population as well.
set.seed(42)
mean_list <- list(c(-3, -3), c(0, 0), c(3, 3))
cov_list <- list(cov_identity, 2 * cov_identity, 3 * cov_identity)
data2 <- simdata_contaminated(n = c(10, 10, 10), mean = mean_list,
                              cov = cov_list, epsilon = c(0.05, 0.1, 0.2),
                              kappa = c(2, 5, 10))
dim(data2$x)
table(data2$y)
```

---

simdata_friedman *Generates data from 3 multivariate normal data populations having the covariance structure from Friedman (1989).*

---

## Description

In a widely cite paper, Friedman (1989) described six simulation configurations to study classifiers. This function provides an interface to all six configurations.

## Usage

```
simdata_friedman(n = rep(15, K), p = 10, experiment = 1,
  seed = NULL)
```

## Arguments

| | |
|---|---|
| n | a vector (of length 3) of the sample sizes for each population |
| p | number of features of the generated data |
| experiment | the experiment number from the RDA paper |
| seed | seed for random number generation (If NULL, does not set seed) |

## Details

We generate $n_k$ observations $(k = 1, \ldots, K)$ from each of $K = 3$ multivariate normal distributions. Let the $k$th population have a $p$-dimensional multivariate normal distribution, $N_p(\mu_k, \Sigma_k)$ with mean vector $\mu_k$ and positive-definite covariance matrix $\Sigma_k$. Each covariance matrix $\Sigma_k$ consists of a covariance structure based on the experiment chosen.

Here, we provide a brief description of each of the six experimental configurations. For more information, see Friedman (1989). We use Friedman's original setup except we fix the number of observations rather than randomly choosing the number of class observations.

Define $I_p$ as the $p$-dimensional identity matrix.

Experiment #1 – Equal, Spherical Covariance Matrices

Each of the three classes are generated from a population with covariance matrix, $I_p$. The population mean of the first class is the origin. The means of the other two classes are taken to be 3.0 in two orthogonal directions.

Experiment #2 – Unequal, Spherical Covariance Matrices

Let each of the classes have covariance matrix $k * I_p$, where $k$ is the class number $(1 \leq k \leq 3)$. Similar to experiment #1, the population mean of the first class is the origin; the means for classes 2 and 3 are shifted in orthogonal directions, class 2 by a distance of 3.0 and class 3 by a distance of 4.0.

Experiment #3 – Equal, Highly Ellipsoidal Covariance Matrices

The covariance matrices of all three classes are equal and highly ellipsoidal. The location differences between the classes are concentrated in the low-variance subspace. The $j$th eigenvalue $(j = 1, \ldots, p)$ of the common covariance matrices is

$$e_j = [9(j-1)/(p-1) + 1]^2,$$

so that the ratio of the largest to smallest eigenvalues is 100.

The population mean of the first class is the origin. The mean vectors for the second and third classes are in terms of the population eigenvalues. The mean of the $j$th feature for class 2 is

$$\mu_{2j} = 2.5\sqrt{e_j/p}\frac{p-j}{p/2-1},$$

where $e_j$ is the $j$th eigenvalue given above. The mean of the $j$th feature for class 3 is

$$\mu_{3j} = (-1)^j \mu_{2j}.$$

Experiment #4 – Equal, Highly Ellipsoidal Covariance Matrices

Similar to Experiment #3, the covariance matrices of all three classes are equal and highly ellipsoidal. However, in this experiment the location differences between the classes are concentrated in the high-variance subspace. The $j$th eigenvalue $(j = 1, \ldots, p)$ of the common covariance matrices is

$$[9 * (j-1)/(p-1) + 1]^2,$$

so the ratio of the largest to smallest eigenvalues is 100.

The population mean of the first class is the origin. The mean vectors for the second and third classes are in terms of the population eigenvalues. The mean of the $j$th feature for class 2 is

$$\mu_{2j} = 2.5\sqrt{e_j/p}\frac{j-1}{p/2-1},$$

where $e_j$ is the $j$th eigenvalue given above. The mean of the $j$th feature for class 3 is

$$\mu_{3j} = (-1)^j \mu_{2j}.$$

Experiment #5 – Unequal, Highly Ellipsoidal Covariance Matrices

In this experiment, the class covariance matrices are highly ellipsoidal and very unequal. The eigenvalues for the first class are given by

$$e_{1j} = [9(j-1)/(p-1) + 1]^2,$$

so that the ratio of the largest to smallest eigenvalues is 100. The eigenvalues for the second class are

$$e_{2j} = [9(p-j)/(p-1) + 1]^2.$$

The eigenvalues for class 3 are given by

$$e_{3j} = \{9[j - (p-1)/2]/(p-1)\}^2.$$

For the first two classes, the ratio of the largest to the smallest eigenvalues is 100, but their high and low variance subspaces are complementary of each other. This ratio for the third class is $(p+1)^2$. The third class has low variance in the subspace of intermediate variance for the first two classes, and high variance where they have their complementary high/low variances.

Each class' mean vector is the origin so that the class distributions differ only in their covariance matrices.

Experiment #6 – Unequal, Highly Ellipsoidal Covariance Matrices

This experiment uses the same covariance structures described for Experiment #5. The population means, however, are different. The mean vector of the first class is the origin. The mean of the $j$th feature for class 2 is

$$\mu_{2j} = 14/\sqrt{p},$$

and class 3's mean vector is defined such that

$$\mu_{3j} = (-1)^j \mu_{2j}.$$

## Value

named list containing:

**x:** A matrix whose rows are the observations generated and whose columns are the p features (variables)

**y:** A vector denoting the population from which the observation in each row was generated.

## References

Friedman, J. H. (1989), "Regularized Discriminant Analysis," Journal of American Statistical Association, 84, 405, 165-175.

## Examples

```
# Generates 10 observations from three multivariate normal populations having
# the covariance structure given in Friedman's (1989) fifth experiment.
sample_sizes <- c(10, 20, 30)
p <- 20
data <- simdata_friedman(n = sample_sizes, p = p, experiment = 5, seed = 42)
dim(data$x)
table(data$y)

# Generates 15 observations from each of three multivariate normal
# populations having the covariance structure given in Friedman's (1989)
# sixth experiment.
sample_sizes <- c(15, 15, 15)
p <- 20
set.seed(42)
data2 <- simdata_friedman(n = sample_sizes, p = p, experiment = 6)
dim(data2$x)
table(data2$y)
```

---

# simdata_guo *Generates data from* K *multivariate normal data populations having the covariance structure from Guo et al. (2007).*

---

## Description

We generate $n_k$ observations $(k = 1, \ldots, K)$ from each of $K$ multivariate normal distributions. Let the $k$th population have a $p$-dimensional multivariate normal distribution, $N_p(\mu_k, \Sigma_k)$ with mean vector $\mu_k$ and positive-definite covariance matrix $\Sigma_k$. Each covariance matrix $\Sigma_k$ consists of block-diagonal autocorrelation matrices.

## Usage

```
simdata_guo(n, mean, block_size, num_blocks, rho,
  sigma2 = 1, seed = NULL)
```

## Arguments

| | |
|---|---|
| n | a vector (of length K) of the sample sizes for each population |
| mean | a vector or a list (of length K) of mean vectors |
| block_size | a vector (of length K) of the sizes of the square block matrices for each population. See details. |
| num_blocks | a vector (of length K) giving the number of block matrices for each population. See details. |
| rho | a vector (of length K) of the values of the autocorrelation parameter for each class covariance matrix |
| sigma2 | a vector (of length K) of the variance coefficients for each class covariance matrix |
| seed | seed for random number generation (If NULL, does not set seed) |

## Details

The $k$th class covariance matrix is defined as

$$\Sigma_k = \Sigma^{(\rho)} \oplus \Sigma^{(-\rho)} \oplus \ldots \oplus \Sigma^{(\rho)},$$

where $\oplus$ denotes the direct sum and the $(i, j)$th entry of $\Sigma^{(\rho)}$ is

$$\Sigma_{ij}^{(\rho)} = \{\rho^{|i-j|}\}.$$

The matrix $\Sigma^{(\rho)}$ is referred to as a block. Its dimensions are provided in the block_size argument, and the number of blocks are specified in the num_blocks argument.

Each matrix $\Sigma_k$ is generated by the cov_block_autocorrelation function.

The number of populations, K, is determined from the length of the vector of sample sizes, coden. The mean vectors can be given in a list of length K. If one mean is given (as a vector or a list having 1 element), then all populations share this common mean.

The block sizes can be given as a numeric vector or a single value, in which case the degrees of freedom is replicated K times. The same logic applies to num_blocks, rho, and sigma2.

For each class, the number of features, p, is computed as block_size * num_blocks. The values for p must agree for each class.

The block-diagonal covariance matrix with autocorrelated blocks was popularized by Guo et al. (2007) for studying classification of high-dimensional data.

## Value

named list containing:

**x:** A matrix whose rows are the observations generated and whose columns are the p features (variables)

**y:** A vector denoting the population from which the observation in each row was generated.

## References

Guo, Y., Hastie, T., & Tibshirani, R. (2007). "Regularized linear discriminant analysis and its application in microarrays," Biostatistics, 8, 1, 86-100.

## Examples

```
# Generates 10 observations from two multivariate normal populations having
# a block-diagonal autocorrelation structure.
block_size <- 3
num_blocks <- 3
p <- block_size * num_blocks
means_list <- list(seq_len(p), -seq_len(p))
data <- simdata_guo(n = c(10, 10), mean = means_list, block_size = block_size,
                    num_blocks = num_blocks, rho = 0.9, seed = 42)
dim(data$x)
table(data$y)

# Generates 15 observations from each of three multivariate normal
# populations having block-diagonal autocorrelation structures. The
# covariance matrices are unequal.
p <- 16
block_size <- c(2, 4, 8)
num_blocks <- p / block_size
rho <- c(0.1, 0.5, 0.9)
sigma2 <- 1:3
mean_list <- list(rep.int(-5, p), rep.int(0, p), rep.int(5, p))

set.seed(42)
data2 <- simdata_guo(n = c(15, 15, 15), mean = mean_list,
                    block_size = block_size, num_blocks = num_blocks,
                    rho = rho, sigma2 = sigma2)
dim(data2$x)
table(data2$y)
```

---

`simdata_normal` *Generates random variates from K multivariate normal populations.*

---

## Description

We generate $n_k$ observations $(k = 1, \ldots, K)$ from each of $K$ multivariate normal distributions. Let the $k$th population have a $p$-dimensional multivariate normal distribution, $N_p(\mu_k, \Sigma_k)$ with mean vector $\mu_k$ and positive-definite covariance matrix $\Sigma_k$.

## Usage

```
simdata_normal(n, mean, cov, seed = NULL)
```

## Arguments

| | |
|---|---|
| n | a vector (of length K) of the sample sizes for each population |
| mean | a vector or a list (of length K) of mean vectors |
| cov | a symmetric matrix or a list (of length K) of symmetric covariance matrices. |
| seed | seed for random number generation (If NULL, does not set seed) |

## Details

The number of populations, K, is determined from the length of the vector of sample sizes, coden. The mean vectors and covariance matrices each can be given in a list of length K. If one covariance matrix is given (as a matrix or a list having 1 element), then all populations share this common covariance matrix. The same logic applies to population means.

## Value

named list containing:

**x:** A matrix whose rows are the observations generated and whose columns are the p features (variables)

**y:** A vector denoting the population from which the observation in each row was generated.

## Examples

```
# Generates 10 observations from each of two multivariate normal populations
# with equal covariance matrices.
mean_list <- list(c(1, 0), c(0, 1))
cov_identity <- diag(2)
data_generated <- simdata_normal(n = c(10, 10), mean = mean_list,
                                 cov = cov_identity, seed = 42)
dim(data_generated$x)
table(data_generated$y)

# Generates 10 observations from each of three multivariate normal
# populations with unequal covariance matrices.
set.seed(42)
mean_list <- list(c(-3, -3), c(0, 0), c(3, 3))
cov_list <- list(cov_identity, 2 * cov_identity, 3 * cov_identity)
data_generated2 <- simdata_normal(n = c(10, 10, 10), mean = mean_list,
                                  cov = cov_list)
dim(data_generated2$x)
table(data_generated2$y)
```

simdata_t *Generates random variates from K multivariate Student's t populations.*

## Description

We generate $n_k$ observations $(k = 1, \ldots, K_0)$ from each of $K_0$ multivariate Student's t distributions such that the Euclidean distance between each of the means and the origin is equal and scaled by $\Delta \geq 0$.

## Usage

```
simdata_t(n, centroid, cov, df, seed = NULL)
```

## Arguments

| | |
|---|---|
| n | a vector (of length K) of the sample sizes for each population |
| centroid | a vector or a list (of length K) of centroid vectors |
| cov | a symmetric matrix or a list (of length K) of symmetric covariance matrices. |
| df | a vector (of length K) of the degrees of freedom for each population |
| seed | seed for random number generation (If NULL, does not set seed) |

**x:** A matrix whose rows are the observations generated and whose columns are the p features (variables)

**y:** A vector denoting the population from which the observation in each row was generated.

## Details

Let $\Pi_k$ denote the $k$th population with a $p$-dimensional multivariate Student's t distribution, $T_p(\mu_k, \Sigma_k, c_k)$, where $\mu_k$ is the population location vector, $\Sigma_k$ is the positive-definite covariance matrix, and $c_k$ is the degrees of freedom.

For small values of $c_k$, the tails are heavier, and, therefore, the average number of outlying observations is increased.

The number of populations, K, is determined from the length of the vector of sample sizes, coden. The centroid vectors and covariance matrices each can be given in a list of length K. If one covariance matrix is given (as a matrix or a list having 1 element), then all populations share this common covariance matrix. The same logic applies to population centroids. The degrees of freedom can be given as a numeric vector or a single value, in which case the degrees of freedom is replicated K times.

## Examples

```
# Generates 10 observations from each of two multivariate t populations
# with equal covariance matrices and equal degrees of freedom.
centroid_list <- list(c(3, 0), c(0, 3))
cov_identity <- diag(2)
data_generated <- simdata_t(n = c(10, 10), centroid = centroid_list,
                            cov = cov_identity, df = 4, seed = 42)
dim(data_generated$x)
table(data_generated$y)

# Generates 10 observations from each of three multivariate t populations
# with unequal covariance matrices and unequal degrees of freedom.
set.seed(42)
centroid_list <- list(c(-3, -3), c(0, 0), c(3, 3))
cov_list <- list(cov_identity, 2 * cov_identity, 3 * cov_identity)
data_generated2 <- simdata_t(n = c(10, 10, 10), centroid = centroid_list,
                             cov = cov_list, df = c(4, 6, 10))
dim(data_generated2$x)
table(data_generated2$y)
```

---

`simdata_uniform` *Generates random variates from multivariate uniform populations.*

---

## Description

We generate $n$ observations from each of $K_0$ multivariate uniform distributions such that the Euclidean distance between each of the populations and the origin is equal and scaled by $\Delta \geq 0$.

## Usage

```
simdata_uniform(n = rep(25, 5), delta = 0, seed = NULL)
```

## Arguments

| | |
|---|---|
| `n` | a vector (of length $K_0$) of the sample sizes for each population |
| `delta` | the fixed distance between each population and the origin |
| `seed` | seed for random number generation. (If `NULL`, does not set seed) |

## Details

To define the populations, let $x = (X_1, \ldots, X_p)'$ be a multivariate uniformly distributed random vector such that $X_j \sim U(a_j^{(k)}, b_j^{(k)})$ is an independently distributed uniform random variable with $a_j^{(k)} < b_j^{(k)}$ for $j = 1, \ldots, p$.

For each population, we set the mean of the distribution along one feature to $\Delta$, while the remaining features have mean 0. The objective is to have unit hypercubes with $p = K_0$ where the population centroids separate from each other in orthogonal directions as $\Delta$ increases, with no overlap for $\Delta \geq 1$.

Hence, let $(a_k^k, b_k^{(k)}) = c(\Delta - 1/2, \Delta + 1/2)$. For the remaining ordered pairs, let $(a_j^{(k)}, b_j^{(k)}) = (-1/2, 1/2)$.

We generate $n_k$ observations from $k$th population.

For $\Delta = 0$, the $K_0 = 5$ populations are equal.

Notice that the support of each population is a unit hypercube with $p = K_0$ features. Moreover, for $\Delta \geq 1$, the populations are mutually exclusive and entirely separated.

## Value

named list containing:

**x:** A matrix whose rows are the observations generated and whose columns are the p features (variables)

**y:** A vector denoting the population from which the observation in each row was generated.

## Examples

```
data_generated <- simdata_uniform(seed = 42)
dim(data_generated$x)
table(data_generated$y)

data_generated2 <- simdata_uniform(n = 10 * seq_len(5), delta = 1.5)
table(data_generated2$y)
sample_means <- with(data_generated2,
                     tapply(seq_along(y), y, function(i) {
                            colMeans(x[i,])
                     }))
(sample_means <- do.call(rbind, sample_means))
```

---

sortinghat *sortinghat*

---

## Description

sortinghat is a classification framework to streamline the evaluation of classifiers (classification models and algorithms) and seeks to determine the best classifiers on a variety of simulated and benchmark data sets. Several error-rate estimators are included to evaluate the performance of a classifier. This package is intended to complement the well-known 'caret' package.

`which_min` *Helper function that determines which element in a vector is the minimum. Ties can be broken randomly or via first/last ordering.*

## Description

The `which_min` function is intended to be an alternative to the base `which.min` function when a specific tie-breaking method is necessary.

## Usage

```
which_min(x, break_ties = c("random", "first", "last"))
```

## Arguments

| | |
|---|---|
| `x` | vector |
| `break_ties` | method to break ties. The `random` method selects the index of the minimum elements randomly, while the `first` and `last` options imply that the first or last instance of the minimum element will be chosen, respectively. |

## Value

location of the minimum element in the vector `x`. If there is a tie, we break the tie with the method specified in `break_ties`.

## Examples

```
set.seed(42)
z <- runif(5)
z <- c(z[1], z[1], z)

which_min(z)
which_min(z, break_ties = "first")
which_min(z, break_ties = "last")
```

# Index