Package 'MuViCP'

October 12, 2022

Type Package

Title MultiClass Visualizable Classification using Combination of Projections

Version 1.3.2

Date 2016-02-22

Author Mohit Dayal

Maintainer Mohit Dayal <mohitdayal2000@gmail.com>

Description An ensemble classifier for multiclass classification. This is a novel classifier that natively works as an ensemble. It projects data on a large number of matrices, and uses very simple classifiers on each of these projections. The results are then combined, ideally via Dempster-Shafer Calculus.

Imports sm, MASS, gtools, parallel

License GPL-3

NeedsCompilation no

Depends R (>= 2.10)

Repository CRAN

Date/Publication 2016-02-22 22:43:49

R topics documented:

MuViCP-package *MuViCP*

Description

MultiClass Visualizable Classification using Combination of Projections

Details

This is a novel classifier that natively works as an ensemble. It projects data on a large number of matrices, and uses very simple classifiers on each of these projections. The results are then combined, ideally via Dempster-Shafer Calculus.

Author(s)

Mohit Dayal

---

ab.dist *Distance Functions for nearest neighbours*

Description

These functions compute Absolute (`ab.dist`), Euclidean (`eu.dist`) or Mahalanobis (`mh.dist`) distances between two points. The variant functions (`*.matY`), accomplish the same task, but between a point on the one hand, and every point specified as rows of a matrix on the other.

Usage

```
ab.dist(x, y)
eu.dist(x, y)
mh.dist(x, y, A)
ab.dist.matY(x, Y)
eu.dist.matY(x, Y)
mh.dist.matY(x, Y, A)
```

Arguments

`x`	The vector (point) from which distance is sought.
`y`	The vector (point) to which distance is sought.
`Y`	A set of points, specified as rows of a matrix, to which distances are sought.
`A`	The inverse matrix to use for the Mahalanobis distance.

Details

These functions are used internally to decide how the nearest neighbours shall be calculated; the user need not call any of these functions directly. Rather, the choice of distance is specified as a string ('euclidean' or 'absolute' or 'mahal').

Value

Either a single number for the distance, or a vector of distances, corresponding to each row of `Y`.

Author(s)

Mohit Dayal

See Also

get.NN

Examples

```
x <- c(1,2)
y <- c(0,3)
mu <- c(1,3)
Sigma <- rbind(c(1,0.2),c(0.2,1))
Y <- MASS::mvrnorm(20, mu = mu, Sigma = Sigma)
ab.dist(x,y)
eu.dist(x,y)
mh.dist(x,y,Sigma)
ab.dist.matY(x,Y)
eu.dist.matY(x,Y)
mh.dist.matY(x,Y,Sigma)
```

`basis_random` *Generate a random basis*

Description

Generate a random basis

Usage

```
basis_random(n, d = 2)
```

Arguments

`n`	dimensionality of data
`d`	dimensionality of target projection

`bel.builder` *Building Belief Functions*

Description

These are a set of functions that can be used to build belief functions (hence the name `*.builder`). Each of these returns a function that can be used to classify points in two dimensions.

The algorithm used can be judged from the first three letters. Thus the `kde_bel` function uses the kernel density estimate (kde), the `knn_bel` function uses the kernel density estimate together with information on the Nearest Neighbours, the `jit_bel` function uses jittering of the point in the neighbourhood. Finally, the `cor_bel` function uses the kde but includes a factor for self-correction.

These generated functions (return values) are meant to be passed to the `ensemble` function to build an ensemble.

Usage

```
kde_bel.builder(labs, test, train, options = list(coef = 0.90))
knn_bel.builder(labs, test, train, options = list(k = 3, p = FALSE,
dist.type = c('euclidean', 'absolute', 'mahal'), out = c('var', 'cv'),
coef = 0.90))
jit_bel.builder(labs, test, train, options = list(k = 3, p = FALSE, s =
5, dist.type = c('euclidean', 'absolute', 'mahal'), out = c('var',
'cv'), coef = 0.90))
```

Arguments

`labs`	The possible labels for the points. Can be strings. Must be of the same length as `train`
`test`	The indices of the test data in `P`
`train`	The indices of the training data in `P`
`options`	A list of arguments that determine the behaviour of the constructed belief function.
`k`	The number of nearest neighbours to consider, specified as a definite integer
`p`	The number of nearest neighbours to consider, specified as a fraction of the test set
`s`	For the jitter belief function : how many times should each point be jittered in the neighbourhood? Usually, 2 or 3 works.
`dist.type`	The type of distance to use when computing nearest neighbours. Can be one of "euclidean", "absolute", or "mahal"
`out`	Should beliefs be built from the variance (`var`) or the coefficient of variation(`cv`)? Also see the Details section below.
`coef`	The classifier only assigns the class labels that actually occur, that is, ignorance is, by default not accounted for. This argument specifies what amount of belief should be allocated to ignorance; the beliefs to the other classes are correspondingly adjusted. Note that for the 'corrected' classifier, the actual belief assigned to ignorance may be higher than this for some projections. See Details.

Details

Each of these functions uses a different algorithm for classification.

The `kde_bel.builder` returns a classifier that simply evaluates the kernel density estimate of each class on each point, and classifies that point to that class which has the maximum density on it.

The `knn_bel.builder` returns a classifier that tries to locate `k` (or `p*length(train)`) nearest neighbours of each of the points in the test set. It then evaluates the kernel density estimate of each class in the training set on each of these nearest neighbours, and at each of the testing points. With argument `var`, the variance of the set of density values, centered at the density value at the testing point itself, is taken as a measure of that point belonging to this class. With argument `cv`, the coefficient of variation is used instead, and for the mean, one uses the density value on the point itself. Generally, the `var` classifier has higher accuracy.

The `jit_bel.builder` works very similar to the `knn_bel.builder` classifier, but instead uses the nearest neighbour information to determine a point "neighbourhood". The test points are then jittered in this neighbourhood, and on these fake points the kernel density is evaluated. The `var` and `cv` work here as they work in the `knn_bel.builder` classifier.

Value

A Classifier function that can be passed on to the `ensemble` function.

Alternately, 2-D projected data may directly be passed to the classifier function returned, in which case, a matrix of dimensions (Number of Classes) x (length(test)) is returned. Each column sums to 1, and represents the partial assignment of that point to each of the classes. The rows are named after the class names, while the columns are named after the test points. Ignorance is represented by the special symbol 'Inf' and is the last class in the matrix.

Author(s)

Mohit Dayal

Examples

```
##Setting Up
data(cancer)
table(cancer$V2)
colnames(cancer)[1:2] <- c('id', 'type')
cancer.d <- as.matrix(cancer[,3:32])
labs <- cancer$type
test_size <- floor(0.15*nrow(cancer.d))
train <- sample(1:nrow(cancer.d), size = nrow(cancer.d) - test_size)
test <- which(!(1:569 %in% train))
truelabs = labs[test]

projectron <- function(A) cancer.d %*% A

seed <- .Random.seed
F <- projectron(basis_random(30))

##Simple Density Classification
kdebel <- kde_bel.builder(labs = labs[train], test = test, train = train)
```

```
x1 <- kdebel(F)
predicted1 <- apply(x1, MARGIN = 2, FUN = function(x) names(which.max(x)))
table(truelabs, predicted1)

##Density Classification Using Nearest Neighbor Information
knnbel <- knn_bel.builder(labs = labs[train], test = test, train =
train, options = list(k = 3, p = FALSE, dist.type = 'euclidean', out = 'var', coef
= 0.90))
x2 <- knnbel(F)
predicted2 <- apply(x2, MARGIN = 2, FUN = function(x) names(which.max(x)))
table(truelabs, predicted2)

##Same as above but now using the Coefficient of Variation for Classification
knnbel2 <- knn_bel.builder(labs = labs[train], test = test, train =
train, options = list(k = 3, p = FALSE, dist.type = 'euclidean', out = 'cv', coef =
0.90))
x3 <- knnbel2(F)
predicted3 <- apply(x3, MARGIN = 2, FUN = function(x) names(which.max(x)))
table(truelabs, predicted3)

##Density Classification Using Jitter & NN Information
jitbel <- jit_bel.builder(labs = labs[train], test = test, train =
train, options = list(k = 3, s = 2, p = FALSE, dist.type = 'euclidean', out =
'var', coef = 0.90))
x4 <- jitbel(F)
predicted4 <- apply(x4, MARGIN = 2, FUN = function(x) names(which.max(x)))
table(truelabs, predicted4)

##Same as above but now using the Coefficient of Variation for Classification
jitbel2 <- jit_bel.builder(labs = labs[train], test = test, train =
train, options = list(k = 3, p = FALSE, dist.type = 'euclidean', out =
'cv', s = 2, coef = 0.90))
x5 <- jitbel2(F)
predicted5 <- apply(x5, MARGIN = 2, FUN = function(x) names(which.max(x)))
table(truelabs, predicted5)
```

---

bpa *Basic Probability Assignment Objects*

Description

These functions can be used to create, combine and print basic probability assignment objects for classificiation. A Basic Probabilty Assignment is a similar to a probabilty mass function, except that it has an additional mass for the concept for "ignorance".

Usage

```
bpa(n = 1, setlist = c(1:n, Inf), mlist = c(rep(0, n), 1))
## S3 method for class 'bpa'
print(x, verbose = FALSE, ...)
```

Arguments

n	The number of distinct values that need to be represented. Usually set to the number of classes in the data.
setlist	A subset of 1:n, indicating those elements that have positive mass. The special value 'Inf' is used to denote the whole set, which is ignorance in dempster-shafer terms.
mlist	The actual masses assigned to the elements in the setlist.
x	The bpa object to be printed.
verbose	If FALSE (default), simply prints out the basic probability assignment. If TRUE, prints a list of all member functions as well.
...	Additional arguments to print method. Not Used.

Details

It should be noted that these functions are fairly simplistic, since they were designed to be fast, and work with classification only. In particular, if you have set-valued elements, the combination function will likely give the wrong answer unless the sets are non-intersecting. For the same reason, belief functions have not been implemented either, since for atomic elements, bpa = belief function.

Value

The bpa function returns a list of functions which can be used to query and / or manipulate the create bpa object.

get.N	Get the number of distinct values represented in the bpa. Usually set to the number of classes.
get.setlist	Get the sets represented in the bpa
get.full.m	Get the masses assigned to each of the elements.
get.focal.elements	Get only those elements that have a positive mass attached to them. Such elements are called the focal elements of the bpa.
get.m	Get the masses attached only to the focal elements, that is the non-zero elements of the mlist.
get.mass	Get the masses attached to certain specified elements of the bpa. The elements are specifed as a vector via the argument s.
assign.bpa	Can be used to re-assign mass to certain specified elements of the bpa. The argument s is the same as the setlist, and m is the same as mlist.
get.assigned.class	Returns a vector of all possible classes, in decreasing order of assigned probabilty. (That is, the first element is the most likely class, and the last element is the least likely class.)
get.assigned.ratios	Returns a vector of length N-1, each of whose elements is the ratio of the probability assigned to the successive classes. That is, the first element is the ratio of the probability assigned to the most likely class to that assigned to the next most likely class, and so on.

set.name	Can be passed a string to name the bpa object. (Used by the bpamat function; names are set to the row number.)
get.name	Returns the name of the bpa object. If not assigned, returns NULL.
set.info	Can be used to store as auxiliary information. (Used internally by the bpamat function to store the random seed, or the special character 'C' if the bpa object resulted from a combination.)
get.info	Returns whatever was stored as info. If empty, NULL is returned.

Author(s)

Mohit Dayal

References

Gordon, J. and Shortliffe, E. H. (1984). The dempster-shafer theory of evidence. Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project, 3:832-838. Shafer, G. (1986). The combination of evidence. International Journal of Intelligent Systems, 1(3):155-179.

See Also

combine.bpa.bs, combine.bpa.ds, combine.bpa.list.ds, combine.bpa.list.ds

Examples

```
##Empty bpa - All mass is attached to ignorance
b1 <- bpa(3)
b1

##Add a set to this bpa
b1$assign.bpa(s = c(1,2), m = c(0.3,0.4))
print(b1, verbose = TRUE)

##The same thing in a different way - classes can be named
##Note that the print method omits empty classes
b0 <- bpa(3, c('A','B','C', Inf), c(0.3, 0.4, 0, 0.3))
b0

##Another bpa
##Again, class '2' has been omitted
b2 <- bpa(3)
b2$assign.bpa(s = c(1,3), m = c(0.7,0.1))
b2

##Combine
b3 <- combine.bpa.ds(b1,b2)
b3
combine.bpa.bs(b1,b2)

##As a list, should be same answer as above
b4 <- combine.bpa.list.ds(list(b1,b2))
```

```
b4
combine.bpa.list.bs(list(b1,b2))
```

---

bpamat *Matrices of Basic Probability Assignment Objects*

Description

These functions enhance the functionality provided via bpa objects. They essentially provide for the storage of several bpa objects at once as a matrix.

Usage

```
bpamat(info = NULL, mat = NULL)
## S3 method for class 'bpamat'
print(x, ...)
```

Arguments

info	A piece of auxiliary information that you want stored along with the matrix of bpa's. (Used internally to store the random seed, or the special character 'C' if the bpa object resulted from a combination.)
mat	The matrix of bpa's. Each column represents a point, and the rows represent classes. Should have column names set to the row names of the points, and row names set to the names of the classes
x	The bpamat object to be printed.
...	Additional arguments to print method. Not Used.

Details

The ensemble function returns objects of this type.

Value

The bpamat function returns a list of functions which can be used to query and / or manipulate the create bpa object.

set.info	Takes a single argument which is set as the auxiliary information you want stored with the matrix
get.info	Returns the auxiliary information stored with the matrix
assign.mat	Takes a single argument, which should be a matrix of bpa's, to be stored inside the bpamat object.
get.classify	Returns a vector as long as the number of points stored in the bpamat object. The elements are named after the points, and are the current classification of the point, based on the bpamat object.

get.point	Returns the bpa corresponding to a single point, whose name is passed as the argument.
get.mat	Returns the matrix of bpa's.
get.setlist	Returns the class names that occur in the current matrix
get.pointlist	Returns the names of all the points whose bpa's are stored in the current matrix.

Author(s)

Mohit Dayal

See Also

bpa, combine.bpamat.bs, combine.bpamat.ds, combine.bpamat.list.bs, combine.bpamat.list.ds

Examples

```
data(cancer)
table(cancer$V2)
colnames(cancer)[1:2] <- c('id', 'type')

cancer.d <- as.matrix(cancer[,3:32])
labs <- cancer$type
test_size <- floor(0.15*nrow(cancer.d))
train <- sample(1:nrow(cancer.d), size = nrow(cancer.d) - test_size)
test <- which(!(1:569 %in% train))
truelabs <- labs[test]

projectron <- function(A)
    cancer.d %*% A

kdebel <- kde_bel.builder(labs = labs[train], test = test, train =
train)

##A projection
seed1 <- .Random.seed
F1 <- projectron(basis_random(30))
x1 <- kdebel(F1)
y1 <- bpamat(info = seed1, mat = x1)
y1
predicted1 <- y1$get.classify()
table(truelabs, predicted1)

##Another projection
seed2 <- .Random.seed
F2 <- projectron(basis_random(30))
x2 <- kdebel(F2)
y2 <- bpamat(info = seed2, mat = x2)
y2
predicted2 <- y2$get.classify()
table(truelabs, predicted2)
```

```
z1 <- combine.bpamat.bs(y1, y2)
z2 <- combine.bpamat.ds(y1, y2)
table(truelabs, z1$get.classify())
table(truelabs, z2$get.classify())

##Same result
w1 <- combine.bpamat.list.bs(list(y1, y2))
w2 <- combine.bpamat.list.ds(list(y1, y2))
```

`cancer` *Wisconsin Breast Cancer Data from UCI website*

Description

This is part of the Breast Cancer Data, publicly available from the UCI Machine Learning Datasets webpage at https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Original%29

Usage

```
data(cancer)
```

Details

Ten real-valued features are computed for each cell nucleus:

a) radius (mean of distances from center to points on the perimeter) b) texture (standard deviation of gray-scale values) c) perimeter d) area e) smoothness (local variation in radius lengths) f) compactness (perimeter^2 / area - 1.0) g) concavity (severity of concave portions of the contour) h) concave points (number of concave portions of the contour) i) symmetry j) fractal dimension ("coastline approximation" - 1)

The mean, standard error, and "worst" or largest (mean of the three largest values) of these features were computed for each image, resulting in 30 features. For instance, field 3 is Mean Radius, field 13 is Radius SE, field 23 is Worst Radius.

All feature values are recoded with four significant digits.

Missing attribute values: none

Class distribution: 357 benign, 212 malignant

Value

A Data Frame with 569 rows and 32 columns. The first column is some sort of serial number, and the second column is the class label ('M' for Malignant or 'B' for Benign). The rest of the columns are features.

Author(s)

Mohit Dayal

References

1. O. L. Mangasarian and W. H. Wolberg: "Cancer diagnosis via linear programming", SIAM News, Volume 23, Number 5, September 1990, pp 1 & 18.

2. William H. Wolberg and O.L. Mangasarian: "Multisurface method of pattern separation for medical diagnosis applied to breast cytology", Proceedings of the National Academy of Sciences, U.S.A., Volume 87, December 1990, pp 9193-9196.

3. O. L. Mangasarian, R. Setiono, and W.H. Wolberg: "Pattern recognition via linear programming: Theory and application to medical diagnosis", in: "Large-scale numerical optimization", Thomas F. Coleman and Yuying Li, editors, SIAM Publications, Philadelphia 1990, pp 22-30.

4. K. P. Bennett & O. L. Mangasarian: "Robust linear programming discrimination of two linearly inseparable sets", Optimization Methods and Software 1, 1992, 23-34 (Gordon & Breach Science Publishers).

`combine.ds` *Combining Basic Probability Assignments*

Description

These functions can be used to combine one or several basic probability assignments (bpa). In the limited context that we support here, a bpa is nothing but a discrete distribution, that may have an additional mass for ignorance.

The suffix tells how the combination will be done : `ds` denotes that the Dempster-Shafer rules will be used, `bs` denotes that Bayes' rule will be used. Thus the function `combine.ds` combines two numeric vectors by Dempster-Shafer rules.

The first middle denotes what kind of object a function operates on. Thus `combine.bpa.ds` combines two `bpa` objects by Dempster-Shafer rules, while `combine.bpamat.ds` does the same for two bpamat objects.

Finally, the second middle may be used - if set to `list`, it combines lists of objects. Thus, the function `combine.bpa.list.ds` combines lists of `bpa` objects by Dempster-Shafer rules.

Usage

```
combine.bs(x, y)
combine.ds(x, y)
combine.bpa.bs(b1, b2)
combine.bpa.ds(b1, b2)
combine.bpa.list.bs(blist)
combine.bpa.list.ds(blist)
combine.bpamat.bs(bmat1, bmat2)
combine.bpamat.ds(bmat1, bmat2)
combine.bpamat.list.bs(bmatlist)
combine.bpamat.list.ds(bmatlist)
```

Arguments

x	A numeric vector representing a bpa.
y	A numeric vector representing a bpa.
b1	The first bpa object that needs to be combined.
b2	The second bpa object that needs to be combined.
blist	A list of bpa's to be be combined.
bmat1	The first bpa matrix that needs to be combined.
bmat2	The second bpa matrix that needs to be combined.
bmatlist	A list of bpa matrices to be be combined.

Value

The combine.ds functions returns a numeric vector representing the new bpa.

The combine.bpamat.bs, combine.bpamat.ds, combine.bpamat.list.bs and combine.bpamat.list.bs functions themselves returns a bpamat object.

The combine.bpa.bs, combine.bpa.ds, combine.bpa.list.bs and the combine.bpa.list.ds functions themselves returns a bpa object.

Author(s)

Mohit Dayal

References

Gordon, J. and Shortliffe, E. H. (1984). The dempster-shafer theory of evidence. Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project, 3:832-838. Shafer, G. (1986). The combination of evidence. International Journal of Intelligent Systems, 1(3):155-179.

Examples

```
##Very Strong, Consistent Testimony
vstrong <- c(0.85, 0.07, 0.08)
##Strong, Consistent Testimony
strong <- c(0.7, 0.15, 0.15)
##Somewhat Ambiguous Testimony
amb <- c(0.55, 0.40, 0.05)
##More Diffuse Testimony
amb2 <- c(0.55, 0.20, 0.25)

fn_gen <- function(par)
{
    x <- gtools::rdirichlet(2, par)
    y <- x
    y <- t(apply(y, MARGIN = 1, FUN = function(x) x * 0.9))
    y <- cbind(y, 0.1)
    return(y)
```

```
}

a1 <- fn_gen(vstrong)
combine.bs(a1[1,], a1[2,])
combine.ds(a1[1,], a1[2,])

a2 <- fn_gen(strong)
combine.bs(a2[1,], a2[2,])
combine.ds(a2[1,], a2[2,])

a3 <- fn_gen(amb)
combine.bs(a3[1,], a3[2,])
combine.ds(a3[1,], a3[2,])

a4 <- fn_gen(amb2)
combine.bs(a4[1,], a4[2,])
combine.ds(a4[1,], a4[2,])

##For bpa or bpamat examples, see the relevant help files
```

`ensemble` *Ensemble Objects for Classification*

Description

This function provides for the creation, and storage of an ensemble of simpler classifiers. Right now, only a single type of classifier is available; this shall be fixed in the future.

Usage

```
ensemble(dat, train, test, labs, bel.type = c('kde', 'knn', 'jit'), bel_options)
```

Arguments

`dat`	The full data matrix, including both test and train rows.
`train`	A vector containing the row numbers (or names) of the training data in the matrix `dat`.
`test`	A vector containing the row numbers (or names) of the test data in the matrix `dat`.
`labs`	Labels for the training data; must be of the same length as `train`.
`bel.type`	The type of belief function to build the Ensemble. For more details, see help on Belief.
`bel_options`	The options list that should be passed to the belief function.

Details

The simpler classifiers must work in 2-dimensions projections of the data.

Value

Returns a list of functions which can be used to query and / or manipulate the created `ensemble` object.

`try.matrices`	Takes a single argument `n` which is the number of matrices you want to try.
`get.bpamats`	Returns a list of bpa.mat objects representing the classifications recieved from each projection.

Author(s)

Mohit Dayal

See Also

belief

Examples

```
data(cancer)
table(cancer$V2)
colnames(cancer)[1:2] <- c('id', 'type')

cancer.d <- as.matrix(cancer[,3:32])
labs <- cancer$type
test_size <- floor(0.15*nrow(cancer.d))
train <- sample(1:nrow(cancer.d), size = nrow(cancer.d) - test_size)
test <- which(!(1:569 %in% train))
truelabs = labs[test]

e <- ensemble(dat = cancer.d, labs = labs[train], train = train, test =
test, bel.type = 'kde', bel_options = list(coef = 0.90))

##Try more matrices than that in real life!
##Also increase the mc.cores parameter if you have more cores!
e$try.matrices(n = 3, mc.cores = 1)

y <- e$get.bpamats()
length(y)

##Can see results from each projection
##b.1  <- bpamat(mat = y[[1]]$get.mat())
##b.2  <- bpamat(mat = y[[2]]$get.mat())
##b.12b <- combine.bpa.mat.bs(b.1, b.2)
##b.12d <- combine.bpa.mat.ds(b.1, b.2)
##b.12b$get.classify()
##b.12d$get.classify()

##All the results
##b.n <- lapply(y, function(x) x$get.classify())
##allb <- combine.bpa.mat.list.bs(e$get.bpamats())
##alld <- combine.bpa.mat.list.ds(e$get.bpamats())
```

```
##fn1 <- function(x)
##    table(truelabs, x$get.classify())

##fn2 <- function(x)
##    {
##        tmp <- table(truelabs, x$get.classify())
##        100 *sum(diag(tmp)) / sum(tmp)
##    }

##fn1(allb)
##fn2(allb)
##fn1(alld)
##fn2(alld)
```

`get.NN` *Function to find the nearest neighbours*

Description

This function locates the nearest neighbours of each point in the test set in the training set. Both sets must of the same dimensions and are passed as successive rows of the same matrix `P`.

User can decide whether a specified number of neighbours should be sought, or whether they should be sought as some fraction of the size of the training set.

Usage

```
get.NN(P, k = 2, p = !k, test, train, dist.type = c("euclidean",
"absolute", "mahal"), nn.type = c("which", "dist", "max"))
```

Arguments

`P`	The matrix of data. Contains both the training and test sets.
`k`	The number of nearest neighbours sought.
`p`	The number of nearest neighbours sought, specified as a fraction of the training set.
`test`	The rows of the matrix `P` that contain the test data.
`train`	The rows of the matrix `P` that contain the training data.
`dist.type`	The type of distance to use when determining neighbours.
`nn.type`	What should be returned? Either the actual distances (`dist`) or their locations (rows) in `P` (`which`) or the k-th maximum distances `max`

Details

This function is used internally to compute the nearest neighbours; the user need not call any of these functions directly.

Value

Returns a matrix of dimensions (Number of Nearest Neighbours) x (Rows in Test Set). Each column contains the nearest neighbours of the corresponding row in the training set.

Author(s)

Mohit Dayal

Examples

```
require(MASS)
mu <- c(3,4)
Sigma <- rbind(c(1,0.2),c(0.2,1))
Y <- mvrnorm(20, mu = mu, Sigma = Sigma)
test <- 1:4
train <- 5:20
nn1a <- get.NN(Y, k = 3, test = 1:4, train = 5:20, dist.type =
'euclidean', nn.type = 'which')
nn1b <- get.NN(Y, k = 3, test = 1:4, train = 5:20, dist.type =
'euclidean', nn.type = 'dist')
nn1c <- get.NN(Y, k = 3, test = 1:4, train = 5:20, dist.type =
'euclidean', nn.type = 'max')
nn2 <- get.NN(Y, p = 0.3, test = 1:4, train = 5:20, dist.type =
'euclidean', nn.type = 'which')
```

`least.k` *Functions to find the few smallest elements in a vector.*

Description

Given a numeric vector, these functions compute and return the few smallest elements in it. The number of elements returned is specified as either a definite number (`k`) or as a proportion of the vector length (`p`). The variant functions (`which.*`), accomplish the same task, but return instead the position of such elements in the vector.

Usage

```
least.k(x, k)
least.p(x, p)
which.least.k(x, k)
which.least.p(x, p)
```

Arguments

`x`	The numeric vector.
`k`	The number of smallest elements sought.
`p`	The number of smallest elements sought, specified as proportion of the length of `x`.

Details

These functions are used internally in the determination of nearest neighbours; the user need not call any of these functions directly. Rather, the choice is specified via the arguments k and p.

Value

Either the smallest values themselves or, for the which.* functions, their positions in the vector.

Author(s)

Mohit Dayal

See Also

get.NN

Examples

```
x <- rnorm(10)
least.k(x, 3)
least.p(x, 0.3)
which.least.k(x, 3)
which.least.p(x, 0.3)
```

Index