Package 'PRECISION'

July 6, 2016

Type Package

Title PaiREd miCrorna sImulation on Study desIgn for mOlecular classificatioN

Version 0.1.0

Date 05/26/2016

Author Huei-Chung Huang, Li-Xuan Qin

Maintainer Huei-Chung Huang <huangh4@mskcc.org>

Description Allow users to reuse a unique pair of Agilent microRNA microarray datasets and to reproduce and extend the simulation studies reported in the paper at the URL below.

License GPL (>= 2)

Depends R (>= 3.0.2)

Imports glmnet, limma, pamr, preprocessCore, ruv, sva, vsn

URL http://clincancerres.aacrjournals.org/content/20/13/3371.long

LazyData TRUE

RoxygenNote 5.0.1

Suggests knitr, rmarkdown

VignetteBuilder knitr

R topics documented:

`amplify.ary.eff` *Array effect amplification*

Description

Amplify array effect in pre-specified slides by either a location shift or a scale change.

Usage

```
amplify.ary.eff(ary.eff, amplify.ary.id, amplify.level, type = "shift")
```

Arguments

`ary.eff`	the estimated array effect dataset to be modified. The dataset must have rows as probes and columns as samples.
`amplify.ary.id`	the array IDs specified to have its array effect amplified. If `type = "shift"` or `type = "scale1"`, a vector of array IDs must be supplied. If `type = "scale2"`, a list of vectors of array IDs must be supplied.
`amplify.level`	a multiplier specified to amplify array effect by. A numeric multiplier must be supplied if `type = "shift"` or `type = "scale1"`. A vector of multipliers must be supplied if type = "scale2" and it must have an equal length to the `amplify.ary.id` list.
`type`	a choice of amplification type, either "shift", "scale1" or "scale2" for either location shift or scale change. By default `type = "shift"`. Location shift moves the entire specified arrays up or down by a constant. Scale change 1 within each array, re-scales expressions that are in inter-quartiles towards the first and the third quartiles; expressions that are outside of the inter-quartile range remain unchanged. Scale change 2 re-scales the expressions by the power of constants that are specified by the user for each batch.

Value

an array-effect-amplified set of array effects

Examples

```
## Not run:
smp.eff <- estimate.smp.eff(uhdata = uhdata.pl)
ary.eff <- estimate.ary.eff(uhdata = uhdata.pl,
                            nuhdata = nuhdata.pl)

ctrl.genes <- unique(rownames(uhdata.pl))[grep("NC", unique(rownames(uhdata.pl)))]

smp.eff.nc <- smp.eff[!rownames(smp.eff) %in% ctrl.genes, ]
ary.eff.nc <- ary.eff[!rownames(ary.eff) %in% ctrl.genes, ]

ary.eff.nc.tr <- ary.eff.nc[, c(1:64, 129:192)]

# location shift
ary.eff.nc.tr.shift <- amplify.ary.eff(ary.eff = ary.eff.nc.tr,
                                       amplify.ary.id = colnames(ary.eff.nc.tr)[1:64],
                                       amplify.level = 2, type = "shift")

# scale change 1
ary.eff.nc.tr.scale1 <- amplify.ary.eff(ary.eff = ary.eff.nc.tr,
                                        amplify.ary.id = colnames(ary.eff.nc.tr)[1:64],
                                        amplify.level = 2, type = "scale1")

# scale change 2
amplify.ary.id <- list(1:40, 41:64, (129:160) - 64, (161:192) - 64)
for(i in 1:length(amplify.ary.id))
  amplify.ary.id[[i]] <- colnames(ary.eff.nc.tr)[amplify.ary.id[[i]]]
amplify.level <- c(1.2, 1.3, 1/3, 2/3)

ary.eff.nc.tr.scale2 <- amplify.ary.eff(ary.eff = ary.eff.nc.tr,
                                        amplify.ary.id = amplify.ary.id,
                                        amplify.level = amplify.level,
                                        type = "scale2")


par(mfrow = c(2, 2), mar = c(4, 3, 2, 2))
rng <- range(ary.eff.nc.tr, ary.eff.nc.tr.shift,
             ary.eff.nc.tr.scale1, ary.eff.nc.tr.scale2)
boxplot(ary.eff.nc.tr, main = "original",
        ylim = rng, pch = 20, cex = 0.2, xaxt = "n")
boxplot(ary.eff.nc.tr.shift, main = "shifted",
        ylim = rng, pch = 20, cex = 0.2, xaxt = "n")
boxplot(ary.eff.nc.tr.scale1, main = "scaled 1",
        ylim = rng, pch = 20, cex = 0.2, xaxt = "n")
boxplot(ary.eff.nc.tr.scale2, main = "scaled 2",
        ylim = rng, pch = 20, cex = 0.2, xaxt = "n")

## End(Not run)
```

blocking.design *Blocking Design*

Description

Assign arrays to samples with blocking by (8-plex Agilent) array slide.

Usage

```
blocking.design(seed, num.smp)
```

Arguments

seed	an integer used to initialize a pseudorandom number generator.
num.smp	number of arrays. It must be a multiple of 8.

Value

a vector of array IDs in the order of assigning to samples that are assumed to be sorted by sample group of interest As a result, the first half of the array IDs are assigned to group 1 and the second half of the array IDs are assigned to group 2.

Examples

```
blocking.design(seed = 1, num.smp = 128)
```

calc.confounding.level *Level of confounding calculation*

Description

Calculate the level of confounding between handling effects and sample group of interest for array data.

Usage

```
calc.confounding.level(data, group.id, nbe.genes)
```

Arguments

data	expression dataset. The dataset must have rows as probes and columns as samples.
group.id	a vector of sample-group labels for each sample of the dataset.
nbe.genes	a vector of non-biological genes indicated as TRUE. It must have an equal length to the number of probes in the dataset.

Value

a list of two elements:

locc	the level of confounding
k_pc	the most correlated principal component of the non-biological genes in the dataset with the sample group

Examples

```
## Not run:
smp.eff <- estimate.smp.eff(uhdata = uhdata.pl)
ary.eff <- estimate.ary.eff(uhdata = uhdata.pl,
                            nuhdata = nuhdata.pl)

ctrl.genes <- unique(rownames(uhdata.pl))[grep("NC", unique(rownames(uhdata.pl)))]

smp.eff.nc <- smp.eff[!rownames(smp.eff) %in% ctrl.genes, ]
ary.eff.nc <- ary.eff[!rownames(ary.eff) %in% ctrl.genes, ]

group.id <- substr(colnames(smp.eff.nc), 7, 7)

smp.eff.train.ind <- colnames(smp.eff.nc)[c(sample(which(group.id == "E"), size = 64),
sample(which(group.id == "V"), size = 64))]
ary.eff.train.ind <- colnames(ary.eff.nc)[c(1:64, 129:192)]

# randomly created a vector of Boolean for nbe.genes
nbe.genes <- sample(c(TRUE, FALSE), size = nrow(smp.eff.nc), replace = TRUE)

calc.confounding.level(data = smp.eff.nc[, smp.eff.train.ind],
                       group.id = substr(smp.eff.train.ind, 7, 7),
                       nbe.genes = nbe.genes)

## End(Not run)
```

---

confounding.design *Confounding Design*

Description

Assign arrays to samples with confounding design, intentionally assigning arrays to sample groups in the order of array collection. Since the non-uniformly-handled data had the earlier arrays processed by one technician and the later arrays processed by another, assigning the earlier arrays to one sample group and the later arrays to another results in confounding handling effects with the sample groups.

Usage

```
confounding.design(seed, num.smp, degree = "complete", rev.order = FALSE)
```

Arguments

seed	an integer used to initialize a pseudorandom number generator.
num.smp	number of arrays.
degree	level of confounding. It must be either "complete" or "partial" for "complete confounding" or "partial confounding" design. By default, degree = "complete".
rev.order	whether the array-to-sample-group assignment should be flipped. Originally the first half arrays are designated to be assigned to group 1 (endometrial sample group) and the second half to group 2 (ovarian sample group). If the array-to-sample-group assignment is flipped (rev.order = TRUE), the first half of the array IDs will be swapped with the second half of the array IDs. By default, rev.order = FALSE.

Value

a vector of array IDs in the order of assigning to samples that are assumed to be sorted by sample group of interest As a result, the first half of the array IDs are assigned to group 1 and the second half of the array IDs are assigned to group 2.

Examples

```
cc.ind <- confounding.design(seed = 1, num.smp = 128,
                             degree = "complete", rev.order = FALSE)
cc.ind <- confounding.design(seed = 1, num.smp = 128,
                             degree = "complete", rev.order = FALSE)
```

---

estimate.ary.eff *Estimated array effects*

Description

Estimate array effects from taking the differences between the expressions of the non-uniformly-handled and the uniformly-handled data, matched by samples.

Usage

```
estimate.ary.eff(uhdata, nuhdata)
```

Arguments

uhdata	the uniformly-handled expression dataset. The dataset must have rows as probes and columns as samples.
nuhdata	the non-uniformly-handled expression dataset. The dataset must have rows as probes and columns as samples and the same dimensions and the same probe names as the uniformly-handled dataset.

Value

an estimation of the array effects

Examples

```
ary.eff <- estimate.ary.eff(uhdata = uhdata.pl, nuhdata = nuhdata.pl)
```

---

estimate.smp.eff *Estimated Sample Effects*

Description

Estimate sample effects from the expressions of the uniformly-handled data.

Usage

```
estimate.smp.eff(uhdata)
```

Arguments

uhdata	the uniformly-handled expression dataset. The dataset must have rows as probes and columns as samples.

Value

an estimation of the sample effects

Examples

```
smp.eff <- estimate.smp.eff(uhdata = uhdata.pl)
```

---

lasso.intcv *Least absolute shrinkage and selection operator through internal cross validation*

Description

Build a LASSO classifier using internal cross validation to choose the turning parameter, with a 5-fold cross validation as default.

Usage

```
lasso.intcv(kfold = 5, X, y, seed, alp = 1)
```

Arguments

kfold	number of folds. By default, kfold = 5.
X	expression dataset to be trained. This dataset must have rows as probes and columns as samples.
y	a vector of sample group of each sample for the dataset to be trained. It must have an equal length to the number of samples in X.
seed	an integer used to initialize a pseudorandom number generator.
alp	alpha, the penalty type. It can be any numeric value from 0 to 1. By default, alp = 1 which is for LASSO. alp = 0 is for ridge and any value in between is for elastic net.

Value

a list of 4 elements:

mc	an internal misclassification error rate
time	the processing time of performing internal validation with LASSO
model	a LASSO classifier, resulted from cv.fit
cfs	estimated coefficients for the final classifier

References

Friedman, J., Hastie, T. and Tibshirani, R. (2008) Regularization Paths for Generalized Linear Models via Coordinate Descent, http://www.stanford.edu/~hastie/Papers/glmnet.pdf Journal of Statistical Software, Vol. 33(1), 1-22 Feb 2010

Examples

```
set.seed(101)
smp.eff <- estimate.smp.eff(uhdata = uhdata.pl)
ctrl.genes <- unique(rownames(uhdata.pl))[grep("NC", unique(rownames(uhdata.pl)))]
smp.eff.nc <- smp.eff[!rownames(smp.eff) %in% ctrl.genes, ]
group.id <- substr(colnames(smp.eff.nc), 7, 7)

smp.eff.train.ind <- colnames(smp.eff.nc)[c(sample(which(group.id == "E"), size = 64),
                                            sample(which(group.id == "V"), size = 64))]
smp.eff.nc.tr <- smp.eff.nc[, smp.eff.train.ind]

lasso.int <- lasso.intcv(X = smp.eff.nc.tr,
                         y = substr(colnames(smp.eff.nc.tr), 7, 7),
                         kfold = 5, seed = 1, alp = 1)
```

`lasso.predict` *Prediction with least absolute shrinkage and selection operator classifier*

Description

Predict from a least absolute shrinkage and selection operator fit.

Usage

```
lasso.predict(lasso.intcv.model, pred.obj, pred.obj.group.id)
```

Arguments

`lasso.intcv.model`	a LASSO classifier built with `lasso.intcv()`.
`pred.obj`	expression dataset to have its sample group predicted. The dataset must have rows as probes and columns as samples. It must have an equal number of probes as the dataset being trained.
`pred.obj.group.id`	a vector of sample-group labels for each sample of the dataset to be predicted. It must have an equal length to the number of samples as `pred.obj`.

Value

a list of 3 elements:

`pred`	predicted sample group for each sample
`mc`	a predicted misclassification error rate (external validation)
`prob`	predicted probability for each sample

References

Friedman, J., Hastie, T. and Tibshirani, R. (2008) Regularization Paths for Generalized Linear Models via Coordinate Descent, http://www.stanford.edu/~hastie/Papers/glmnet.pdf Journal of Statistical Software, Vol. 33(1), 1-22 Feb 2010

Examples

```
set.seed(101)
smp.eff <- estimate.smp.eff(uhdata = uhdata.pl)
ctrl.genes <- unique(rownames(uhdata.pl))[grep("NC", unique(rownames(uhdata.pl)))]
smp.eff.nc <- smp.eff[!rownames(smp.eff) %in% ctrl.genes, ]
group.id <- substr(colnames(smp.eff.nc), 7, 7)

smp.eff.train.ind <- colnames(smp.eff.nc)[c(sample(which(group.id == "E"), size = 64),
                                            sample(which(group.id == "V"), size = 64))]
smp.eff.test.ind <- colnames(smp.eff.nc)[!colnames(smp.eff.nc) %in% smp.eff.train.ind]
```

```
smp.eff.nc.tr <- smp.eff.nc[, smp.eff.train.ind]
smp.eff.nc.te <- smp.eff.nc[, smp.eff.test.ind]

lasso.int <- lasso.intcv(X = smp.eff.nc.tr,
                         y = substr(colnames(smp.eff.nc.tr), 7, 7),
                         kfold = 5, seed = 1, alp = 1)

lasso.pred <- lasso.predict(lasso.intcv.model = lasso.int,
                            pred.obj = smp.eff.nc.te,
                            pred.obj.group.id = substr(colnames(smp.eff.nc.te), 7, 7))
lasso.int$mc
lasso.pred$mc
```

`limma.pbset` *Differential expression analysis of probe-set data*

Description

Perform two-group differential expression analysis using "limma".

Usage

```
limma.pbset(data, group.id, group.id.level = c("E", "V"), pbset.id = NULL)
```

Arguments

`data`	expression dataset to be analyzed. The dataset must have rows as unique probe-sets and columns as samples.
`group.id`	a vector of sample-group labels for each sample of the dataset. It must be a 2-level non-numeric factor vector.
`group.id.level`	a vector of sample-group label level. It must have two and only two elements and the first element is the reference. By default, `group.id.level = c("E", "V")`. That is in our study, we compare endometrial tumor samples to ovarian tumor samples, with endometrial as our reference.
`pbset.id`	a vector of unique probe-set names. By default, `pbset.id = NULL` for it to be the row names of the dataset.

Value

a data frame with differential expression analysis results, group means and group standard deviations, for each unique probe-set.

References

Ritchie ME, Phipson B, Wu D, Hu Y, Law CW, Shi W and Smyth GK (2015). "limma powers differential expression analyses for RNA-sequencing and microarray studies." Nucleic Acids Research, 43(7), pp. e47.

Examples

```
uhdata.psl <- med.sum.pbset(data = uhdata.pl,
                            num.per.unipbset = 10)
ctrl.genes <- unique(rownames(uhdata.pl))[grep("NC", unique(rownames(uhdata.pl)))]

uhdata.psl.nc <- uhdata.psl[!rownames(uhdata.psl) %in% ctrl.genes, ]

group.id <- substr(colnames(uhdata.psl.nc), 7, 7)
group.id.level <- levels(as.factor(group.id))

limma.fit.uhdata<- limma.pbset(data = uhdata.psl.nc,
                               group.id = group.id,
                               group.id.level = group.id.level)
                               table(limma.fit.uhdata$P.Value < 0.01,
                               dnn = "DE genes")
```

`med.norm` *Median normalization*

Description

Normalize training dataset so that each array shares a same median and store the median from the training dataset as the reference to frozen median normalize test dataset.

Usage

```
med.norm(train, test = NULL)
```

Arguments

`train`	training dataset to be median normalized. The dataset must have rows as probes and columns as samples.
`test`	test dataset to be frozen median normalized. The dataset must have rows as probes and columns as samples. The number of rows must equal to the number of rows in the training set. By default, the test set is not specified (`test = NULL`) and no frozen normalization will be performed.

Value

a list of two datasets:

`train.mn`	the normalized training set
`test.fmn`	the frozen normalized test set, if test set is specified

Examples

```
set.seed(101)
group.id <- substr(colnames(nuhdata.pl), 7, 7)
train.ind <- colnames(nuhdata.pl)[c(sample(which(group.id == "E"), size = 64),
                                   sample(which(group.id == "V"), size = 64))]
train.dat <- nuhdata.pl[, train.ind]
test.dat <- nuhdata.pl[, !colnames(nuhdata.pl) %in% train.ind]
data.mn <- med.norm(train = train.dat)
str(data.mn)
data.mn <- med.norm(train = train.dat, test = test.dat)
str(data.mn)
```

---

med.sum.pbset *Probe-set median summarization*

Description

Summarize probe-set using median of each unique probe and only takes in data matrix with the same number of probes per unique probe-set.

Usage

```
med.sum.pbset(data, pbset.id = NULL, num.per.unipbset = 10)
```

Arguments

data	expression dataset to be summarized. The dataset must have rows as probes and columns as samples. It must be a data matrix with the same number of probes per unique probe-set. If it is already on the probe-set level, no manipulation will be done.
pbset.id	a vector of unique probe-set names. If it is not specified, then by default it is set to be the unique probe names of the data.
num.per.unipbset	number of probes for each unique probe-set. By default, num.per.unipbset = 10.

Value

probe-set median summarized data

Examples

```
## Not run:
uhdata.psl <- med.sum.pbset(data = uhdata.pl,
                            num.per.unipbset = 10)

## End(Not run)
```

nuhdata.pl — *The non-uniformly-handled probe-level dataset, 10 probes for each unique probe*

Description

A five percent random subset of the non-uniformly-handled probe-level dataset, 10 probes per each unique probe. The expressions are on a log2 scale without background adjustmnet. This dataset consists of 181 unique probes, of which 6 are negatively biological control probes from Agilent array platform: "NC2_00079215", "NC1_00000215", "NC1_00000197", "NC2_00122731", "NC2_00092197", and "NC2_00106057". The sample IDs (the column names) ending with "E" or "V" are used to indicate whether a sample is endometrial or ovarian tumor sample. There are 96 endometrial and 96 ovarian tumor samples.

Usage

```
nuhdata.pl
```

Format

A data matrix with 1810 rows (probes) and 192 columns (samples).

pam.intcv — *Nearest shrunken centroid through internal cross validation*

Description

Build a PAM classifier using internal cross validation to choose the tuning parameter, with 5-fold cross validation as the default.

Usage

```
pam.intcv(X, y, vt.k = NULL, n.k = 30, kfold = 5, folds = NULL, seed)
```

Arguments

X	expression dataset to be trained. This dataset must have rows as probes and columns as samples.
y	a vector of sample group of each sample for the dataset to be trained. It must have an equal length to the number of samples in X.
vt.k	custom-specified threshold list. By default, vt.k = NULL and 30 values will be predetermined by the pamr package.
n.k	number of threshold values desired. By default, n.k = 30.
kfold	number of folds. By default, kfold = 5.

folds	pre-specifies samples to each fold. By default, folds = NULL for no pre-specification.
seed	an integer used to initialize a pseudorandom number generator.

Value

a list of 4 elements:

mc	an internal misclassification error rate
time	processing time of performing internal validation with PAM
model	a PAM classifier, resulted from pamr.train
cfs	estimated coefficients for the final classifier

References

T. Hastie, R. Tibshirani, Balasubramanian Narasimhan and Gil Chu (2014). pamr: Pam: prediction analysis for microarrays. R package version 1.55. https://CRAN.R-project.org/package=pamr

Examples

```
set.seed(101)
smp.eff <- estimate.smp.eff(uhdata = uhdata.pl)
ctrl.genes <- unique(rownames(uhdata.pl))[grep("NC", unique(rownames(uhdata.pl)))]
smp.eff.nc <- smp.eff[!rownames(smp.eff) %in% ctrl.genes, ]
group.id <- substr(colnames(smp.eff.nc), 7, 7)

smp.eff.train.ind <- colnames(smp.eff.nc)[c(sample(which(group.id == "E"), size = 64),
                                            sample(which(group.id == "V"), size = 64))]
smp.eff.nc.tr <- smp.eff.nc[, smp.eff.train.ind]

pam.int <- pam.intcv(X = smp.eff.nc.tr,
                     y = substr(colnames(smp.eff.nc.tr), 7, 7),
                     kfold = 5, seed = 1)
```

---

pam.predict *Prediction with nearest shrunken centroid classifier*

Description

Predict from a nearest shrunken centroid fit.

Usage

```
pam.predict(pam.intcv.model, pred.obj, pred.obj.group.id)
```

Arguments

pam.intcv.model	a PAM classifier built with pam.intcv().
pred.obj	expression dataset to have its sample group predicted. The dataset must have rows as probes and columns as samples. It must have an equal number of probes as the dataset being trained.
pred.obj.group.id	a vector of sample-group labels for each sample of the dataset to be predicted. It must have an equal length to the number of samples as pred.obj.

Value

a list of 3 elements:

pred	predicted sample group for each sample
mc	a predicted misclassification error rate (external validation)
prob	predicted probability for each sample

References

T. Hastie, R. Tibshirani, Balasubramanian Narasimhan and Gil Chu (2014). pamr: Pam: prediction analysis for microarrays. R package version 1.55. https://CRAN.R-project.org/package=pamr

Examples

```
set.seed(101)
smp.eff <- estimate.smp.eff(uhdata = uhdata.pl)
ctrl.genes <- unique(rownames(uhdata.pl))[grep("NC", unique(rownames(uhdata.pl)))]
smp.eff.nc <- smp.eff[!rownames(smp.eff) %in% ctrl.genes, ]
group.id <- substr(colnames(smp.eff.nc), 7, 7)

smp.eff.train.ind <- colnames(smp.eff.nc)[c(sample(which(group.id == "E"), size = 64),
                                            sample(which(group.id == "V"), size = 64))]
smp.eff.test.ind <- colnames(smp.eff.nc)[!colnames(smp.eff.nc) %in% smp.eff.train.ind]

smp.eff.nc.tr <- smp.eff.nc[, smp.eff.train.ind]
smp.eff.nc.te <- smp.eff.nc[, smp.eff.test.ind]

pam.int <- pam.intcv(X = smp.eff.nc.tr,
                     y = substr(colnames(smp.eff.nc.tr), 7, 7),
                     kfold = 5, seed = 1)

pam.pred <- pam.predict(pam.intcv.model = pam.int,
                        pred.obj = smp.eff.nc.te,
                        pred.obj.group.id = substr(colnames(smp.eff.nc.te), 7, 7))
pam.int$mc
pam.pred$mc
```

per.unipbset.truncate *Probe-level data truncation to a fixed number of probes per unique probe-set*

Description

Truncate probe-level dataset so that it has a fixed number of probes per unique probe-set. We are safe to do so if the variation among replicates for the same probe is small.

Usage

```
per.unipbset.truncate(data, pbset.id = NULL, num.per.unipbset = 10)
```

Arguments

data	probe-level expression dataset. The dataset must have rows as probes and columns as samples.
pbset.id	a vector of unique probe-set names. By default, pbset.id = NULL for it to be the row names of the dataset.
num.per.unipbset	number of probes for each unique probe-set to be truncated to. By default, num.per.unipbset = 10.

Value

truncated probe-level data

Examples

```
uhdata.pl.p5 <- per.unipbset.truncate(data = uhdata.pl,
num.per.unipbset = 5)
```

precision.simulate *Classification analysis of simulation study*

Description

Perform the simulation study in Qin et al. (see reference).

Usage

```
precision.simulate(seed, N, smp.eff.tr, smp.eff.te, ary.eff.tr, ary.eff.te,
  group.id.tr, group.id.te, design.list = c("CC+", "CC-", "PC+", "PC-"),
  norm.list = c("NN", "QN"), class.list = c("PAM", "LASSO"),
  batch.id = NULL, icombat = FALSE, isva = FALSE, iruv = FALSE,
  smp.eff.tr.ctrl = NULL, ary.eff.tr.ctrl = NULL, norm.funcs = NULL,
  class.funcs = NULL, pred.funcs = NULL)
```

Arguments

seed	an integer used to initialize a pseudorandom number generator.
N	number of simulation runs.
smp.eff.tr	the training set of the estimated sample effects. This dataset must have rows as probes and columns as samples.
smp.eff.te	the test set of the estimated sample effects. This dataset must have rows as probes and columns as samples. It must have the same number of probes and the same probe names as the training set of the estimated sample effects.
ary.eff.tr	the training set of the estimated array effects. This dataset must have rows as probes and columns as samples. It must have the same dimensions and the same probe names as the training set of the estimated sample effects.
ary.eff.te	the test set of the estimated array effects. This dataset must have rows as probes, columns as samples. It must have the same dimensions and the same probe names as the training set of the estimated array effects.
group.id.tr	a vector of sample-group labels for each sample of the training set of the estimated sample effects. It must be a 2-level non-numeric factor vector.
group.id.te	a vector of sample-group labels for each sample of the test set of the estimated sample effects. It must be a 2-level non-numeric factor vector.
design.list	a list of strings for study designs to be compared in the simulation study. The built-in designs are "CC+", "CC-", "PC+", "PC-", "BLK", and "STR" for "Complete Confounding 1", "Complete Confounding 2", "Partial Confounding 1", "Partial Confounding 2", "Blocking", and "Stratification" in Qin et al.
norm.list	a list of strings for normalization methods to be compared in the simulation study. The build-in available normalization methods are "NN", "QN", "MN", "VSN" for "No Normalization", "Quantile Normalization", "Median Normalization", "Variance Stabilizing Normalization". User can provide a list of normalization methods given the functions are supplied (also see norm.funcs).
class.list	a list of strings for classification methods to be compared in the simulation study. The built-in classification methods are "PAM" and "LASSO" for "prediction analysis for microarrays" and "least absolute shrinkage and selection operator". User can provide a list of classification methods given the correponding model-building and predicting functions are supplied (also see class.funcs and pred.funcs).
batch.id	a list of array indices grouped by batches when data were profiled. The length of the list must be equal to the number of batches in the data; the number of array indices must be the same as the number of samples. This is required if stratification study design is specified in design.list; otherwise batch.id = NULL.
icombat	an indicator for combat adjustment. By default, icombat = FALSE for no ComBat adjustment.
isva	an indicator for sva adjustment. By default, isva = FALSE for no sva adjustment.
iruv	an indicator for RUV-4 adjustment. By default, iruv = FALSE for no RUV-4 adjustment.

`smp.eff.tr.ctrl`
: the training set of the negative-control probe sample effect data if `iruv = TRUE`. This dataset must have rows as probes and columns as samples. It also must have the same number of samples and the same sample names as `smp.eff.tr`.

`ary.eff.tr.ctrl`
: the training set of the negative-control probe array effect data if `iruv = TRUE`. This dataset must have rows as probes and columns as samples. It also must have the same dimensions and the same probe names as `smp.eff.tr.ctrl`.

`norm.funcs`
: a list of strings for names of user-defined normalization method functions, in the order of `norm.list`, excluding any built-in normalization methods.

`class.funcs`
: a list of strings for names of user-defined classification model-building functions, in the order of `class.list`, excluding any built-in classification methods.

`pred.funcs`
: a list of strings for names of user-defined classification predicting functions, in the order of `class.list`, excluding any built-in classification methods.

Details

The classification anlaysis of simulation study consists of the following main steps:

First, `precision.simulate` requires the training and test sets for both estimated sample effects and estimated array effects. The effects can be simulated as follows (using `estimate.smp.eff` and `estimate.ary.eff`). The uniformly-handled dataset are used to approximate the biological effect for each sample, and the difference between the two arrays (one from the uniformly-handled dataset and the other from the non-uniformly-handled dataset, subtracting the former from the latter) for the same sample are used to approximate the handling effect for each array in the non-uniformly-handled dataset.

The samples are randomly split into a training set and a test set, balanced by tumor type (in Qin et al., training-to-test ratio is 2:1). The arrays were then non-randomly split to a training set and a test set (in Qin et al., training set n = 128 – the first 64 and last 64 arrays in the order of array processing; test set n = 64 – the middle 64 arrays). This setup allows different pairings of arrays and samples by various different training-and-test-set splits. Furthermore, biological signal strength and confounding level of the handling effects can be modified (using `reduce.signal` and `amplify.ary.eff`).

Second, for the training set, data are simulated through "virtual re-hybridization" (using `rehybridize`) by first assigning arrays to sample groups using a confounding design or a balanced design, and then summing the biological effect for a sample and the handling effect for its assigned array. Rehybridization allows us to examine the use of various array-assignment schemes, specified in `design.list`.

Third, the analysis for each simulated dataset follows the same steps as described for the analysis of the uniformly-handled data (also see documentation on `uni.handled.siumate`):

(1) data preprocessing (normalization methods are specified in `norm.list` and batch effects can be adjusted specified with `icombat`, `isva` and `iruv`)

(2) classifier training (classification methods are specified in `class.list`)

(3) classification error estimation using both cross-validation and external validation

The only difference is that here external validation is based on the test data from the uniformly-handled dataset and served as the gold standard for the misclassification error estimation.

For a given split of samples to training set versus test set, N datasets will be simulated and analyzed for each array-assignment scheme. For user-defined normalization method or classification method, please refer to the vignette.

Value

simulation study results – a list of array-to-sample assignments, fitted models, and misclassification error rates across simulation runs:

assign_store	array-to-sample assignments for each study design
model_store	models for each combination of study designs, normalization methods, and classification methods
error_store	internal and external misclassification error rates for each combination of study designs, normalization methods, and classification methods

References

http://clincancerres.aacrjournals.org/content/20/13/3371.long

Examples

```
## Not run:
set.seed(101)
smp.eff <- estimate.smp.eff(uhdata = uhdata.pl)
ary.eff <- estimate.ary.eff(uhdata = uhdata.pl,
                            nuhdata = nuhdata.pl)

ctrl.genes <- unique(rownames(uhdata.pl))[grep("NC", unique(rownames(uhdata.pl)))]

smp.eff.nc <- smp.eff[!rownames(smp.eff) %in% ctrl.genes, ]
ary.eff.nc <- ary.eff[!rownames(ary.eff) %in% ctrl.genes, ]

group.id <- substr(colnames(smp.eff.nc), 7, 7)

# randomly split sample effect data into training and test set with
# equal number of endometrial and ovarian samples
smp.eff.train.ind <- colnames(smp.eff.nc)[c(sample(which(group.id == "E"), size = 64),
                                            sample(which(group.id == "V"), size = 64))]
smp.eff.test.ind <- colnames(smp.eff.nc)[!colnames(smp.eff.nc) %in% smp.eff.train.ind]
smp.eff.train.test.split =
  list("tr" = smp.eff.train.ind,
       "te" = smp.eff.test.ind)

# non-randomly split array effect data into training and test set
ary.eff.train.test.split =
  list("tr" = c(1:64, 129:192),
       "te" = 65:128)

smp.eff.nc.tr <- smp.eff.nc[, smp.eff.train.ind]
smp.eff.nc.te <- smp.eff.nc[, smp.eff.test.ind]
ary.eff.nc.tr <- ary.eff.nc[, c(1:64, 129:192)]
```

```
ary.eff.nc.te <- ary.eff.nc[, 65:128]

# Simulation without batch adjustment
precision.results <- precision.simulate(seed = 1, N = 3,
                                        smp.eff.tr = smp.eff.nc.tr,
                                        smp.eff.te = smp.eff.nc.te,
                                        ary.eff.tr = ary.eff.nc.tr,
                                        ary.eff.te = ary.eff.nc.te,
                                    group.id.tr = substr(colnames(smp.eff.nc.tr), 7, 7),
                                    group.id.te = substr(colnames(smp.eff.nc.te), 7, 7),
                                        design.list = c("PC-", "STR"),
                                        norm.list = c("NN", "QN"),
                                        class.list = c("PAM", "LASSO"),
                                        batch.id = list(1:40,
                                                        41:64,
                                                        (129:160) - 64,
                                                        (161:192) - 64))

# Simulation with RUV-4 batch adjustment
smp.eff.ctrl <- smp.eff[rownames(smp.eff) %in% ctrl.genes, ]
ary.eff.ctrl <- ary.eff[rownames(ary.eff) %in% ctrl.genes, ]

smp.eff.tr.ctrl <- smp.eff.ctrl[, smp.eff.train.test.split$tr]
ary.eff.tr.ctrl <- ary.eff.ctrl[, ary.eff.train.test.split$tr]

precision.ruv4.results <- precision.simulate(seed = 1, N = 3,
                                             smp.eff.tr = smp.eff.nc.tr,
                                             smp.eff.te = smp.eff.nc.te,
                                             ary.eff.tr = ary.eff.nc.tr,
                                             ary.eff.te = ary.eff.nc.te,
                                    group.id.tr = substr(colnames(smp.eff.nc.tr), 7, 7),
                                    group.id.te = substr(colnames(smp.eff.nc.te), 7, 7),
                                             design.list = c("PC-", "STR"),
                                             norm.list = c("NN", "QN"),
                                             class.list = c("PAM", "LASSO"),
                                             batch.id = list(1:40,
                                                             41:64,
                                                             (129:160) - 64,
                                                             (161:192) - 64),
                                             iruv = TRUE,
                                             smp.eff.tr.ctrl = smp.eff.tr.ctrl,
                                             ary.eff.tr.ctrl = ary.eff.tr.ctrl)

## End(Not run)
```

---

quant.norm *Quantile normalization*

Description

Normalize training dataset with quantile normalization and store the quantiles from the training dataset as the references to frozen quantile normalize test dataset.

Usage

```
quant.norm(train, test = NULL)
```

Arguments

`train`	training dataset to be quantile normalized. The dataset must have rows as probes and columns as samples.
`test`	test dataset to be frozen quantile normalized. The dataset must have rows as probes and columns as samples. The number of rows must equal to the number of rows in the training set. By default, the test set is not specified (`test = NULL`) and no frozen normalization will be performed.

Value

a list of two datasets:

`train.mn`	the normalized training set
`test.fmn`	the frozen normalized test set, if test set is specified

References

Bolstad, B. M., Irizarry R. A., Astrand, M, and Speed, T. P. (2003) A Comparison of Normalization Methods for High Density Oligonucleotide Array Data Based on Bias and Variance. Bioinformatics 19(2) , pp 185-193. http://bmbolstad.com/misc/normalize/normalize.html

Examples

```
set.seed(101)
group.id <- substr(colnames(nuhdata.pl), 7, 7)
train.ind <- colnames(nuhdata.pl)[c(sample(which(group.id == "E"), size = 64),
                                   sample(which(group.id == "V"), size = 64))]
train.dat <- nuhdata.pl[, train.ind]
test.dat <- nuhdata.pl[, !colnames(nuhdata.pl) %in% train.ind]
data.qn <- quant.norm(train = train.dat)
str(data.qn)
data.qn <- quant.norm(train = train.dat, test = test.dat)
str(data.qn)
```

`reduce.signal` *Biological signal reduction*

Description

Reduce biological signal by decreasing the mean group difference between sample groups.

Usage

```
reduce.signal(smp.eff, group.id, group.id.level = c("E", "V"),
  reduce.multiplier = 1/2, pbset.id = NULL)
```

Arguments

`smp.eff`	the estimated sample effect dataset. The dataset must have rows as probes and columns as samples. It can only take in probe-level dataset with a fixed number of probes per unique probe-set.
`group.id`	a vector of sample-group labels for each sample of the estimated sample effect dataset.
`group.id.level`	a vector of sample-group label level. It must have two and only two elements and the first element is the reference. By default, `group.id.level = c("E", "V")`. That is in our study, we compare endometrial tumor samples to ovarian tumor samples, with endometrial as our reference.
`reduce.multiplier`	a multiplier specified to reduce between-sample-group signal by. By default, `reduce.multiplier = 1/2`.
`pbset.id`	a vector of unique probe-set names. If it is not specified, it is the unique probe names of the dataset, extracting from the row names.

Value

estimated sample effect data, with reduced biological signal

Examples

```
smp.eff <- estimate.smp.eff(uhdata = uhdata.pl)
ary.eff <- estimate.ary.eff(uhdata = uhdata.pl,
                            nuhdata = nuhdata.pl)

ctrl.genes <- unique(rownames(uhdata.pl))[grep("NC", unique(rownames(uhdata.pl)))]

smp.eff.nc <- smp.eff[!rownames(smp.eff) %in% ctrl.genes, ]
ary.eff.nc <- ary.eff[!rownames(ary.eff) %in% ctrl.genes, ]
group.id <- substr(colnames(smp.eff.nc), 7, 7)

redhalf.smp.eff.nc <- reduce.signal(smp.eff = smp.eff.nc,
                                    group.id = group.id,
```

```
                                    group.id.level = c("E", "V"),
                                    reduce.multiplier = 1/2)
```

`rehybridize` *Virtual rehybridization with an array-to-sample assignment*

Description

Create simulated dataset through "virtual rehybridization" for a given array-to-sample assignment.

Usage

```
rehybridize(smp.eff, ary.eff, group.id, group.id.level = c("E", "V"),
  ary.to.smp.assign, icombat = FALSE, isva = FALSE, iruv = FALSE,
  smp.eff.ctrl = NULL, ary.eff.ctrl = NULL)
```

Arguments

`smp.eff`	the estimated sample effect dataset. The dataset must have rows as probes and columns as samples.
`ary.eff`	the estimated array effect dataset. The dataset must have rows as probes and columns as samples. It must have the same dimensions and the same probe names as the estimated sample effect dataset.
`group.id`	a vector of sample-group labels for each sample of the estimated sample effect dataset. It must be a 2-level non-numeric factor vector.
`group.id.level`	a vector of sample-group label level. It must have two and only two elements and the first element is the reference. By default, `group.id.level = c("E", "V")`. That is in our study, we compare endometrial tumor samples to ovarian tumor samples, with endometrial as our reference.
`ary.to.smp.assign`	a vector of indices that assign arrays to samples (see details in `blocking.design`, `confounding.design` or `stratification.design`). It must have an equal length to the number of samples in the estimated sample effect dataset. The first half arrays in the vector have to be assigned to the sample group 1 and the second half to sample group 2.
`icombat`	an indicator for combat adjustment. By default, `icombat = FALSE` for no ComBat adjustment.
`isva`	an indicator for sva adjustment. By default, `isva = FALSE` for no sva adjustment.
`iruv`	an indicator for RUV-4 adjustment. By default, `iruv = FALSE` for no RUV-4 adjustment.
`smp.eff.ctrl`	the negative-control probe sample effect data if `iruv = TRUE`. This dataset must have rows as probes and columns as samples. It also must have the same number of samples and the same sample names as `smp.eff`.
`ary.eff.ctrl`	the negative-control probe array effect data if `iruv = TRUE`. It also must have the same dimensions and the same probe names as `smp.eff.ctrl`.

Value

simulated data, after batch adjustment if specified

Examples

```
## Not run:
smp.eff <- estimate.smp.eff(uhdata = uhdata.pl)
ary.eff <- estimate.ary.eff(uhdata = uhdata.pl,
                            nuhdata = nuhdata.pl)

ctrl.genes <- unique(rownames(uhdata.pl))[grep("NC", unique(rownames(uhdata.pl)))]

smp.eff.nc <- smp.eff[!rownames(smp.eff) %in% ctrl.genes, ]
ary.eff.nc <- ary.eff[!rownames(ary.eff) %in% ctrl.genes, ]

assign.ind <- confounding.design(seed = 1, num.smp = 192,
degree = "complete", rev.order = FALSE)

group.id <- substr(colnames(smp.eff.nc), 7, 7)

# no batch effect adjustment (default)
sim.data.raw <- rehybridize(smp.eff = smp.eff.nc,
                            ary.eff = ary.eff.nc,
                            group.id = group.id,
                            ary.to.smp.assign = assign.ind)

# batch effect adjusting with sva
sim.data.sva <- rehybridize(smp.eff = smp.eff.nc,
                            ary.eff = ary.eff.nc,
                            group.id = group.id,
                            ary.to.smp.assign = assign.ind,
                            isva = TRUE)

# batch effect adjusting with RUV-4
smp.eff.ctrl <- smp.eff[rownames(smp.eff) %in% ctrl.genes, ]
ary.eff.ctrl <- ary.eff[rownames(ary.eff) %in% ctrl.genes, ]

sim.data.ruv <- rehybridize(smp.eff = smp.eff.nc,
                            ary.eff = ary.eff.nc,
                            group.id = group.id,
                            ary.to.smp.assign = assign.ind,
                            iruv = TRUE,
                            smp.eff.ctrl = smp.eff.ctrl,
                            ary.eff.ctrl = ary.eff.ctrl)

## End(Not run)
```

`stratification.design` *Stratification Design*

Description

Assign arrays to samples with stratification, a study design assigning arrays in each batch to each sample group proportionally.

Usage

```
stratification.design(seed, num.smp, batch.id)
```

Arguments

seed	an integer used to initialize a pseudorandom number generator.
num.smp	number of arrays.
batch.id	a list of array indices grouped by batches when data were profiled. The length of the list must be equal to the number of batches in the data; the number of array indices must be the same as the number of samples.

Value

a vector of array IDs in the order of assigning to samples that are assumed to be sorted by sample group of interest As a result, the first half of the array IDs are assigned to group 1 and the second half of the array IDs are assigned to group 2.

Examples

```
batch.id <- list(1:40, 41:64, (129:160) - 64, (161:192) - 64)
str.ind <- stratification.design(seed = 1, num.smp = 128,
                                 batch.id = batch.id)
```

---

uhdata.pl *The uniformly-handled probe-level dataset, 10 probes for each unique probe*

Description

A five percent random subset of the uniformly-handled probe-level dataset, 10 probes per each unique probe. The expressions are on a log2 scale without background adjustmnet. This dataset consists of 181 unique probes, of which 6 are negatively biological control probes from Agilent array platform: "NC2_00079215", "NC1_00000215", "NC1_00000197", "NC2_00122731", "NC2_00092197", and "NC2_00106057". The sample IDs (the column names) ending with "E" or "V" are used to indicate whether a sample is endometrial or ovarian tumor sample. There are 96 endometrial and 96 ovarian tumor samples.

Usage

```
uhdata.pl
```

Format

A data matrix with 1810 rows (probes) and 192 columns (samples).

`uni.handled.simulate` *Classification analysis of uniformly-handled data*

Description

Perform classification analysis on the uniformly-handled data by re-assigning samples to training and test set. More details can be found in Qin et al. (see reference).

Usage

```
uni.handled.simulate(seed, N, smp.eff, norm.list = c("NN", "QN"),
  class.list = c("PAM", "LASSO"), norm.funcs = NULL, class.funcs = NULL,
  pred.funcs = NULL)
```

Arguments

`seed`	an integer used to initialize a pseudorandom number generator.
`N`	number of simulation runs.
`smp.eff`	the estimated sample effect dataset. This dataset must have rows as probes and columns as samples.
`norm.list`	a list of strings for normalization methods to be compared in the simulation study. The built-in normalization methods includes "NN", "QN", "MN", "VSN" for "No Normalization", "Quantile Normalization", "Median Normalization", "Variance Stabilizing Normalization". User can provide a list of normalization methods given the functions are supplied (also see `norm.funcs`).
`class.list`	a list of strings for classification methods to be compared in the simulation study. The built-in classification methods are "PAM" and "LASSO" for "prediction analysis for microarrays" and "least absolute shrinkage and selection operator". User can provide a list of classification methods given the correponding model-building and predicting functions are supplied (also see `class.funcs` and `pred.funcs`).
`norm.funcs`	a list of strings for names of user-defined normalization method functions, in the order of `norm.list`, excluding any built-in normalization methods.
`class.funcs`	a list of strings for names of user-defined classification model-building functions, in the order of `class.list`, excluding any built-in classification methods.
`pred.funcs`	a list of strings for names of user-defined classification predicting functions, in the order of `class.list`, excluding any built-in classification methods.

Details

The analysis for the uniformly-handled dataset consists of the following main steps:

(1) randomly split the data into a training set and a test set, balanced by sample group of interest

(2) preprocess the training data and the test data

(3) build a classifier using the preprocessed training data

(4) assess the mislcassification error rate of the classifier using the preprocessed test data

This analysis is repeated for `N` random splits of training set and test set.

Data preprocessing in (2) includes three steps: log2 transformation, normalization for training data and frozen normalization for test data, and probe-set summarization using median. Normalization methods are specified in `norm.list`.

Classifier building in (3) includes choosing the tuning parameter for each method using five-fold cross-validation and measuring classifier accuarcy using the misclassification error rate. Classification methods are specified in `class.list`

The error rate is evaluated by both external validation of test data and cross-validation of training data. For user-defined normalization method or classification method, please refer to the vignette.

Value

benchmark analysis results – a list of training-and-test-set splits, fitted models, and misclassification error rates across simulation runs:

`assign_store`	random training-and-test-set splits
`model_store`	models for each combination of normalization methods and classification methods
`error_store`	internal and external misclassification error rates for each combination of normalization methods and classification methods

References

http://clincancerres.aacrjournals.org/content/20/13/3371.long

Examples

```
## Not run:
smp.eff <- estimate.smp.eff(uhdata = uhdata.pl)

ctrl.genes <- unique(rownames(uhdata.pl))[grep("NC", unique(rownames(uhdata.pl)))]

smp.eff.nc <- smp.eff[!rownames(smp.eff) %in% ctrl.genes, ]

uni.handled.results <- uni.handled.simulate(seed = 1, N = 3,
                                            smp.eff = smp.eff.nc,
                                            norm.list = c("NN", "QN"),
                                            class.list = c("PAM", "LASSO"))

## End(Not run)
```

vs.norm *Variance stabilizing normalization*

Description

Normalize training dataset with vsn and store the fitted vsn model from the training dataset as the reference to frozen variance stabilizing normalize test dataset.

Usage

```
vs.norm(train, test = NULL)
```

Arguments

train	training dataset to be variance stabilizing normalized. The dataset must have rows as probes and columns as samples.
test	test dataset to be frozen variance stabilizing normalized. The dataset must have rows as probes and columns as samples. The number of rows must equal to the number of rows in the training set. By default, the test set is not specified (`test = NULL`) and no frozen normalization will be performed.

Value

a list of two datasets:

train.mn	the normalized training set
test.fmn	the frozen normalized test set, if test set is specified

References

Wolfgang Huber, Anja von Heydebreck, Holger Sueltmann, Annemarie Poustka and Martin Vingron. Variance Stabilization Applied to Microarray Data Calibration and to the Quantification of Differential Expression. Bioinformatics 18, S96-S104 (2002).

Examples

```
## Not run:
set.seed(101)
group.id <- substr(colnames(nuhdata.pl), 7, 7)
train.ind <- colnames(nuhdata.pl)[c(sample(which(group.id == "E"), size = 64),
                                    sample(which(group.id == "V"), size = 64))]
train.dat <- nuhdata.pl[, train.ind]
test.dat <- nuhdata.pl[, !colnames(nuhdata.pl) %in% train.ind]
data.vsn <- vs.norm(train = train.dat)
str(data.vsn)
data.vsn <- vs.norm(train = train.dat, test = test.dat)
str(data.vsn)

## End(Not run)
```

Index