Package ‘cubfits’

February 19, 2015

Version 0.1-0
Date 2014-05-15
Title Codon Usage Bias Fits
Depends R(>= 3.0.0), methods
Suggests seqinr, VGAM, EMCluster
Enhances pbdMPI (>= 0.2-2), parallel
LazyLoad yes
LazyData yes
Description Estimating mutation and selection coefficients on synonymous codon bias usage based on models of ribosome overhead cost (ROC). Multinomial logistic regression and Markov Chain Monte Carlo are used to estimate and predict protein production rates with/without the presence of expressions and measurement errors. Work flows with examples for simulation, estimation and prediction processes are also provided with parallelization speedup. The whole framework is tested with yeast genome and gene expression data of Yassour (2009).
License Mozilla Public License 2.0
URL https://github.com/snoweye/cubfits
Author Wei-Chen Chen [aut, cre], Russell Zaretzki [aut], William Howell [aut], Drew Schmidt [aut], Michael Gilchrist [aut], Students REU13 [ctb]
Maintainer Wei-Chen Chen <wccsnow@gmail.com>
NeedsCompilation yes
Repository CRAN
Date/Publication 2014-05-19 07:14:28
Description

Estimating mutation and selection coefficients on synonymous codon bias usage based on models of ribosome overhead cost (ROC). Multinomial logistic regression and Markov Chain Monte Carlo are used to estimate and predict protein production rates with/without the presence of expressions and measurement errors.

Details

Package: cubfits
Type: Package
License: Mozilla Public License 2.0
LazyLoad: yes
The install command is simply as

```
> R CMD INSTALL cubfits_*.tar.gz
```

from a command mode or

```
R> install.packages("cubfits")
```

inside an R session.

**Author(s)**

Wei-Chen Chen <wccsnow@gmail.com>, Russell Zaretzki, William Howell, Drew Schmidt, and Michael Gilchrist.

**References**

[https://github.com/snoweye/cubfits/](https://github.com/snoweye/cubfits/)

**See Also**

`init.function()`, `cubfits()`, `cubpred()`, and `cubappr()`.

**Examples**

```r
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

demo(roc.train, 'cubfits', ask = F, echo = F)
demo(roc.pred, 'cubfits', ask = F, echo = F)
demo(roc.appr, 'cubfits', ask = F, echo = F)
## End(Not run)
```

---

**Description**

Density, probability, quantile, random number generation, and MLE functions for the asymmetric Laplace distribution with parameters either in \( ASL(\theta, \mu, \sigma) \) or the alternative \( ASL^*(\theta, \kappa, \sigma) \).
Asymmetric Laplace Distribution

Usage

\[
\text{dasl}(x, \theta = 0, \mu = 0, \sigma = 1, \log = \text{FALSE})
\]
\[
\text{dasla}(x, \theta = 0, \kappa = 1, \sigma = 1, \log = \text{FALSE})
\]
\[
\text{pasl}(q, \theta = 0, \mu = 0, \sigma = 1, \text{lower.tail} = \text{TRUE}, \log.p = \text{FALSE})
\]
\[
\text{pasla}(q, \theta = 0, \kappa = 1, \sigma = 1, \text{lower.tail} = \text{TRUE}, \log.p = \text{FALSE})
\]
\[
\text{qasl}(p, \theta = 0, \mu = 0, \sigma = 1, \text{lower.tail} = \text{TRUE}, \log.p = \text{FALSE})
\]
\[
\text{qasla}(p, \theta = 0, \kappa = 1, \sigma = 1, \text{lower.tail} = \text{TRUE}, \log.p = \text{FALSE})
\]
\[
\text{rasl}(n, \theta = 0, \mu = 0, \sigma = 1)
\]
\[
\text{rasla}(n, \theta = 0, \kappa = 1, \sigma = 1)
\]
\[
\text{asl.optim}(x)
\]

Arguments

- \(x, q\) vector of quantiles.
- \(p\) vector of probabilities.
- \(n\) number of observations. If \(\text{length}(n) > 1\), the length is taken to be the number required.
- \(\theta\) center parameter.
- \(\mu, \kappa\) location parameters.
- \(\sigma\) shape parameter.
- \(\log, \log.p\) logical; if \(\text{TRUE}\), probabilities \(p\) are given as \(\log(p)\).
- \(\text{lower.tail}\) logical; if \(\text{TRUE}\) (default), probabilities are \(P[X \leq x]\) otherwise, \(P[X > x]\).

Details

The density \(f(x)\) of \(\text{ASL}^*(\theta, \kappa, \sigma)\) is given as

\[
\frac{\sqrt{2}}{\sigma} \sqrt{1 + \kappa^2} \exp\left(-\frac{\sqrt{2}\kappa}{\sigma} |x - \theta|\right) \text{if} \ x \geq \theta, \text{and} \ \frac{\sqrt{2}}{\sigma} \sqrt{1 + \kappa^2} \exp\left(-\frac{\sqrt{2}\kappa}{\sigma} |x - \theta|\right) \text{if} \ x < \theta.
\]

The parameter domains of \(\text{ASL}\) and \(\text{ASL}^*\) are \(\theta \in R, \sigma > 0, \kappa > 0, \text{and} \ \mu \in R\). The relation of \(\mu\) and \(\kappa\) are \(\kappa = \frac{\sqrt{2\sigma^2 + \mu^2}}{\sqrt{2\sigma}}\) or \(\mu = \frac{\sigma}{\sqrt{2}} (\frac{1}{\kappa} - \kappa)\).

Value

“dasl” and “dasla” give the densities, “pasl” and “pasla” give the distribution functions, “qasl” and “qasla” give the quantile functions, and “rasl” and “rasla” give the random numbers.

\text{asl.optim} returns the MLE of data \(x\) including \(\theta, \mu, \kappa, \text{and} \ \sigma\).
**Codon Adaptation Index**

**Author(s)**

Wei-Chen Chen <wccsnow@gmail.com>.

**References**


**Examples**

```r
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))
set.seed(1234)

dasl(-2:2)
dasl(-2:2)
pasl(-2:2)
pasl(-2:2)
qasl(seq(0, 1, length = 5))
qasl(seq(0, 1, length = 5))

dasl(-2:2, log = TRUE)
dasl(-2:2, log = TRUE)
pasl(-2:2, log.p = TRUE)
pasl(-2:2, log.p = TRUE)
qasl(log(seq(0, 1, length = 5)), log.p = TRUE)
qasl(log(seq(0, 1, length = 5)), log.p = TRUE)

set.seed(123)
rasl(5)
rasl(5)
asl.optim(rasl(5000))

## End(Not run)
```

---

**Function for Codon Adaptation Index (CAI)**

**Description**

Calculate the Codon Adaptation Index (CAI) for each gene. Used as a substitute for expression in cases of without expression measurements.

**Usage**

```
calc_cai_values(y, y.list, w = NULL)
```
Arguments

- `y` an object of format `y`.
- `y.list` an object of format `y.list`.
- `w` a specified relative frequency of synonymous codons.

Details

This function computes CAI for each gene. Typically, this method is completely based on entropy and information theory to estimate expression values of sequences according to their codon information.

If the input `w` is NULL, then empirical values are computed.

Value

A list with two named elements `cai` and `w` are returned where `cai` are CAI of input sequences (`y` and `y.list`) and `w` are the relative frequency used to compute those CAI’s.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References


See Also

calc_scu_values(), calc_scu_values().

Examples

```r
## Not run:
rm(list = ls())
library(cubfits, quietly = TRUE)

y <- ex.train$y
y.list <- convert.y.to.list(y)
CAI <- calc_cai_values(y, y.list)$CAI
plot(CAI, log10(ex.train$phi.Obs), main = "Expression vs CAI",
     xlab = "CAI", ylab = "Expression (log10)")

### Verify with the seqinr example.
library(seqinr, quietly = TRUE)
inputdatfile <- system.file("sequences/input.dat", package = "seqinr")
input <- read.fasta(file = inputdatfile, forceDNAtolower = FALSE)
names(input)[65] <- paste(names(input)[65], ".1", sep = "") # name duplicated.
input <- input[order(names(input))]

### Convert to cubfits format.
```
seq.string <- convert.seq.data.to.string(input)
new.y <- gen.y(seq.string)
new.y.list <- convert.y.to.list(new.y)
ret <- calc_cai_values(new.y, new.y.list)

### Rebuild w.
w <- rep(1, 64)
names(w) <- codon.low2up(rownames( Caitab))
for(i in 1:64){
id <- which(names(ret$w) == names(w)[i])
if(length(id) == 1){
w[i] <- ret$w[id]
}
}
CAI.res <- sapply(input, seqinr::cai, w = w)

### Plot.
plot(CAI.res, ret$CAI,
     main = "Comparison of seqinR and cubfits results",
     xlab = "CAI from seqinR", ylab = "CAI from cubfits", las = 1)
abline(c(0, 1))

## End(Not run)

---

### Controls

#### Default Controlling Options

**Description**

Default controls of **cubfits** include for models, optimizations, MCMC, plotting, global variables, etc.

**Usage**

```
.cubfitsEnv
.CF.CT
.CF.CONF
.CF.GV
.CF.DP
.CF.OP
.CF.AC
.CF.PT
.CF.PARAM
.CO.CT
```

**Format**

All are in lists and contain several controlling options.
Details

See `init.function()` for use cases of these objects.

- `.cubfitEnv` is a default environment to dynamically save functions and objects.
- `.CF_CT` is main controls of models. It currently includes
  
  ```
  model main models
  type.p proposal for hyper-parameters
  type.Phi proposal for Phi
  model.Phi prior of Phi
  init.Phi initial methods for Phi
  init.fit how is coefficient proposed
  parallel parallel functions
  adaptive method for adaptive MCMC
  ```

- `.CF_CONF` controls the initial and draw scaling. It currently includes
  
  ```
  scale.phi if phi were scaled to mean 1
  init.b.Scale initial b scale
  init.phi.Scale initial phi scale
  p.nclass number of classes if mixture phi
  b.DrawScale drawing scale for b if random walk
  p.DrawScale drawing scale for p if random walk
  phi.DrawScale random walk scale for phi
  phi.DrawScale.pred random walk scale for phi.pred
  ```

- `.CF_GV` contains global variables for amino acids and codons. It currently includes
  
  ```
  amino.acid amino acids
  synonymous.codon synonymous codons of amino acids
  amino.acid.split amino acid 'S' is split
  synonymous.codon.split synonymous codons of split amino acid
  ```

- `.CF_OP` controls optimizations. It currently includes
  
  ```
  optim.method method for optim()
  stable.min.exp minimum exponent
  stable.max.exp maximum exponent
  E.Phi expected Phi
  lower.optim lower of derivative of logL(x)
  upper.optim upper of derivative of logL(x)
  lower.integrate lower of integration of L(x)
  upper.integrate upper of integration of L(x)
  ```

- `.CF_DP` is for dumping MCMC iterations. It currently includes
  
  ```
  dump if dumping within MCMC
  iter iterations per dumping
  ```
Controls

prefix.dump  path and file names of dumping
trace.acceptance  if trace acceptance rate
verbose  if verbose
iterThin  iterations to thin chain
report  iterations to report
report.proc  iterations to report proc.time()

- .CF.AC controls adaptive MCMC. It currently includes
  
  renew.iter  per renewing iterations
  target.accept.lower  target acceptance lower bound
  target.accept.upper  target acceptance upper bound
  scale.increase  10% more
  scale.decrease  10% less
  sigma2.lower  lower bound of sigma^2
  sigma2.upper  upper bound of sigma^2

- .CF.PT controls the plotting format. It currently includes
  
  color  color for codons.

- .CF.PARAM controls the parameters and hyperparameters of priors. It currently includes
  
  phi.meanlog  mean of phi in loca scale
  phi.sdlog  standard deviation of phi in loca scale
  hp.gamma.shape  hyperparameters of gamma distribution
  hp.gamma.scale  hyperparameters of gamma distribution
  hp.gamma.inflated  inflate gamma variance if overwrite
  hp.overwrite  if allow my.pInit() to overwrite

- .CO.CT controls the constrained optimization function. It currently includes
  
  debug  message printing level of debugging.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

init.function(), cubfits(), cubpred(), cubappr(), and mixnormerr.optim().

Examples

## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

.onlineCT
.onlineCONF
.onlineDP
.onlineGV
.onlineOP
.onlineAC
.onlinePT
.onlinePARAM
.onlineCT

ls(.cubfitsEnv)
init.function()
ls(.cubfitsEnv)

## End(Not run)

---

Covert utility

Convert Data Frame to Other Formats

**Description**

These utility functions convert data of format divided by amino acids into list of format divided by ORFs, or convert data to other formats.

**Usage**

```r
class.reu13.df.to.list(reu13.df)
class.y.to.list(y)
class.n.to.list(n)
class.y.to.scuo(y)
class.seq.data.to.string(seq.data)
codon.low2up(x)
codon.up2low(x)
dna.low2up(x)
dna.up2low(x)
class.b.to.bVec(b)
class.bVec.to.b(bVec, aa.names, model = .CF$model[1])
```

**Arguments**

- `reu13.df` a list of reu13.df data frames divided by amino acids.
- `y` a list of y data frames divided by amino acids.
Converting Utility

n
- a list of n vectors divided by amino acids.

seq.data
- a vector of seq.data format.

x
- a codon or dna string, such as "ACG", "acg", or "A", "a".

b
- a b object.

bvec
- a bvec object.

aa.names
- a vector contains amino acid names for analysis.

model
- model fitted.

Details

convert.reu13.df.to.list(), convert.y.to.list(), and convert.n.to.list(): these utility functions take the inputs divided by amino acids and return the outputs divided by ORFs.

convert.y.scuo() converts y into scuo format.

convert.seq.data.to.string() converts seq.data into seq.string format.

codon.low2up() and codon.up2low() convert codon strings between lower or upper cases.

convert.bVec.to.b() and convert.b.to.bVec() convert objects b and bVec.

Value

All functions return the corresponding formats.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

AllDataFormats, rearrange.n(), rearrange.reu13.df(), rearrange.y(), and read.seq().

Examples

```r
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

reu13.list <- convert.reu13.df.to.list(ex.train$reu13.df)
y.list <- convert.y.to.list(ex.train$y)
n.list <- convert.n.to.list(ex.train$n)

scuo <- convert.y.to.scuo(ex.train$y)

seq.data <- read.seq(get.exp-path("seq_200.fasta"))
seq.string <- convert.seq.data.to.string(seq.data)

codon.low2up("acg")
```
CUB Model Approximation

Codon Usage Bias Approximation for ORFs without Expression

Description

This function provides codon usage bias approximation with observed ORFs but without any expressions.

Usage

cubappr(reu1SNdf, phi.Ninit, y, n, 
niter = 1000, burnin = 100, 
bInit = NULL, init.b.Scale = .CF.CONF$init.b.Scale, 
b.DrawScale = .CF.CONF$b.DrawScale, 
p.Init = NULL, p.nclass = .CF.CONF$p.nclass, 
p.DrawScale = .CF.CONF$p.DrawScale, 
phi.DrawScale.pred = .CF.CONF$phi.DrawScale.pred, 
model = .CF.CT$model[1], model.Phi = .CF.CT$model.Phi[1], 
adaptive = .CF.CT$adaptive[1], 
verbose = .CF.DP$verbose, 
iterThin = .CF.DP$iterThin, report = .CF.DP$report)

Arguments

reu13.df.obs a reu13.df object, ORFs information.
phi.Init a phi.Init object, temporarily initial of expression without measurement errors.
y a y object, codon counts.
n a n object, total codon counts.
nIter number of iterations after burn-in iterations.
burnin number of burn-in iterations.
bInit initial values for parameters b.
init.b.Scale for initial b if bInit = NULL.
b.DrawScale scaling factor for adaptive MCMC with random walks when drawing new b.
p.Init initial values for hyper-parameters.
p.nclass number of components for model.Phi = "logmixture".
phi.DrawScale.pred
scaling factor for adaptive MCMC with random walks when drawing new Phi of predicted set.

model  
model to be fitted, currently "roc" only.

model.Phi  
prior model for Phi, currently "lognormal".

adaptive  
adaptive method of MCMC for proposing new b and Phi.

verbose  
print iteration messages.

iterThin  
thinning iterations.

report  
number of iterations to report more information.

Details

Total number of MCMC iterations is burnin + nIter + 1, but the outputs may be thinned to (burnin + nIter) / iterThin + 1 iterations.

Temporary result dumping may be controlled by .CF.DP.

Value

A list contains three big lists of MCMC traces including: b.mat for mutation and selection coefficients of b, p.mat for hyper-parameters, and phi.mat for expected expression values Phi. All lists are of length (burnin + nIter) / iterThin + 1 and each element contains the output of each iteration.

All lists also can be binded as trace matrices, such as via do.call("rbind", b.mat) yielding a matrix of dimension number of iterations by number of parameters. Then, those traces can be analyzed further via other MCMC packages such as coda.

Note

Note that phi.Init need to be normalized to mean 1.

p.DrawScale may cause scaling prior if adaptive MCMC is used, and it can result in non-exits of equilibrium distribution.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

DataIO, DataConverting, cubfits() and cubpred().
CUB Model Fits

Examples

## Not run:
suppressMessages(library(cubfits, quietly = TRUE))
demo(roc.appr, 'cubfits', ask = F, echo = F)
## End(Not run)

---

CUB Model Fits  Codon Usage Bias Fits for Observed ORFs and Expression

Description

This function provides codon usage bias fits with observed ORFs and expressions which possibly contains measurement errors.

Usage

```
cubfits(reu13.df.obs, phi.Obs, y, n,
niter = 1000, burnin = 100,
bInit = NULL, init.b.Scale = .CF.CONF$init.b.Scale,
b.DrawScale = .CF.CONF$b.DrawScale,
p.Init = NULL, p.nclass = .CF.CONF$p.nclass,
p.DrawScale = .CF.CONF$p.DrawScale,
phi.Init = NULL, init.phi.Scale = .CF.CONF$init.phi.Scale,
phi.DrawScale = .CF.CONF$phi.DrawScale,
model = .CF.CT$model[1], model.Phi = .CF.CT$model.Phi[1],
adaptive = .CF.CT$adaptive[1],
verbose = .CF.DP$verbose,
iterThin = .CF.DP$iterThin, report = .CF.DP$report)
```

Arguments

- `reu13.df.obs` a `reu13.df` object, ORFs information.
- `phi.Obs` a `phi.Obs` object, expression with measurement errors.
- `y` a `y` object, codon counts.
- `n` a `n` object, total codon counts.
- `nIter` number of iterations after burn-in iterations.
- `burnin` number of burn-in iterations.
- `bInit` initial values for parameters `b`.
- `init.b.Scale` for initial `b` if `bInit = NULL`.
- `b.DrawScale` scaling factor for adaptive MCMC with random walks when drawing new `b`.
- `p.Init` initial values for hyper-parameters.
- `p.nclass` number of components for model.Phi = "logmixture".
CUB Model Fits

- `phi.Init`: initial values for `Phi`.
- `init.phi.Scale`: for initial `phi` if `phi.Init = NULL`.
- `model`: model to be fitted, currently "roc" only.
- `model.Phi`: prior model for `Phi`, currently "lognormal".
- `verbose`: print iteration messages.
- `iterThin`: thinning iterations.
- `report`: number of iterations to report more information.

**Details**

This function correctly and carefully implements a combining version of Shah and Gilchrist (2011) and Wallace et al. (2013).

Total number of MCMC iterations is `burnin + nIter + 1`, but the outputs may be thinned to `(burnin + nIter) / iterThin + 1` iterations.

Temporary result dumping may be controlled by `.CF.DP`.

**Value**

A list contains three big lists of MCMC traces including: `b.mat` for mutation and selection coefficients of `b`, `p.mat` for hyper-parameters, and `phi.mat` for expected expression values `Phi`. All lists are of length `(burnin + nIter) / iterThin + 1` and each element contains the output of each iteration.

All lists also can be binded as trace matrices, such as via `do.call("rbind", b.mat)` yielding a matrix of dimension number of iterations by number of parameters. Then, those traces can be analyzed further via other MCMC packages such as `coda`.

**Note**

Note that `phi.obs` need to be normalized to mean 1.

`p.DrawScale` may cause scaling prior if adaptive MCMC is used, and it can result in non-exits of equilibrium distribution.

**Author(s)**

Wei-Chen Chen `<wccsnow@gmail.com>`.

**References**

[https://github.com/snoweye/cubfits/](https://github.com/snoweye/cubfits/)


CUB Model Prediction

See Also
DataIO, DataConverting, cubappr() and cubpred().

Examples

```r
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))
demo(roc.train, 'cubfits', ask = F, echo = F)
## End(Not run)
```

CUB Model Prediction  Codon Usage Bias Prediction for Observed ORFs

Description
This function provides codon usage bias fits of training set which has observed ORFs and expressions possibly containing measurement errors, and provides predictions of testing set which has other observed ORFs but without expression.

Usage
```
cubpred(reu13.df.obs, phi.Obs, y, n,
   reu13.df.pred, y.pred, n.pred,
   nIter = 1000, burnin = 100,
   bInit = NULL, init.b.Scale = .CF.CONF$init.b.Scale,
   b.DrawScale = .CF.CONF$b.DrawScale,
   p.Init = NULL, p.nclass = .CF.CONF$p.nclass,
   p.DrawScale = .CF.CONF$p.DrawScale,
   phi.Init = NULL, init.phi.Scale = .CF.CONF$init.phi.Scale,
   phi.DrawScale = .CF.CONF$phi.DrawScale,
   phi.Init.pred = NULL,
   phi.DrawScale.pred = .CF.CONF$phi.DrawScale.pred,
   model = .CF.CT$model[1], model.Phi = .CF.CT$model.Phi[1],
   adaptive = .CF.CT$adaptive[1],
   verbose = .CF.DP$verbose,
   iterThin = .CF.DP$iterThin, report = .CF.DP$report)
```

Arguments
```
reu13.df.obs a reu13 df to be trained.
phi.Obs a phi.Obs to be trained.
y a y to be trained.
n a n to be trained.
reu13.df.pred a reu13 df to be predicted.
```
y.pred a y to be predicted.
n.pred a n to be predicted.
nIter number of iterations after burn-in iterations.
burnin number of burn-in iterations.
bInit initial values for parameters b.
init.b.Scale for initial b if bInit = NULL.
b.DrawScale scaling factor for adaptive MCMC with random walks when drawing new b.
p.Init initial values for hyper-parameters.
p.nclass number of components for model. Phi = "logmixture".
phi.Init initial values for Phi.
init.phi.Scale for initial phi if phi.Init = NULL.
phi.DrawScale scaling factor for adaptive MCMC with random walks when drawing new Phi.
phi.Init.pred initial values for Phi of predicted set.
phi.DrawScale.pred as phi.DrawScale but for predicted set.
model model to be fitted, currently "roc" only.
model.Ph phi prior model for Phi, currently "lognormal".
adaptive adaptive method of MCMC for proposing new b and Phi.
verbose print iteration messages.
iterThin thinning iterations.
report number of iterations to report more information.

Details
This function correctly and carefully implements an extension of Shah and Gilchrist (2011) and Wallace et al. (2013).

Total number of MCMC iterations is burnin + nIter + 1, but the outputs may be thinned to (burnin + nIter) / iterThin + 1 iterations.
Temporary result dumping may be controlled by .CF.DP.

Value
A list contains four big lists of MCMC traces including: b.Mat for mutation and selection coefficients of b, p.Mat for hyper-parameters, phi.Mat for expected expression values Phi, and phi.Mat.pred for predictive expression values Phi. All lists have (burnin + nIter) / iterThin + 1 elements, and each element contains the output of each iteration.
All lists also can be binded as trace matrices, such as via do.call("rbind", b.Mat) yielding a matrix of dimension number of iterations by number of parameters. Then, those traces can be analyzed further via other MCMC packages such as coda.
Note

Note that phi.obs need to be normalized to mean 1.
p.DrawScale may cause scaling prior if adaptive MCMC is used, and it can result in non-exits of equilibrium distribution.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/


See Also

DataIO, DataConverting, cubfits() and cubappr().

Examples

```r
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

demo(roc.pred, 'cubfits', ask = F, echo = F)

## End(Not run)
```

Data Formats

Description

Data formats used in cubfits.

Format

All are in simple formats as S3 default lists or data frames.

Details

- Format b:
  A named list A contains amino acids. Each element of the list A[[i]] is a list of elements coefficients (coefficients of log(mu) and Delta.t), coef.mat (matrix format of coefficients), and R (covariance matrix of coefficients). Note that coefficients and R are typically as in the output of vglm() of VGAM package. Also, coef.mat and R may miss in some cases. e.g. A[[i]]$coef.mat is the regression beta matrix of i-th amino acid.
Data Formats

• Format bvec:
  A vector simply contains all coefficients of a b object A. Note that this is probably only used
  inside MCMC or the output of vglm() of VGAM package.
  e.g. do.call("c", lapply(A, function(x) x$coefficients)).

• Format n:
  A named list A contains amino acids. Each element of the list A[[i]] is a vector containing
  total codon counts.
  e.g. A[[i]][j] is for j-th ORF of i-th amino acid names(A)[i].

• Format n.list:
  A named list A contains ORFs. Each element of the list A[[i]] is a named list of amino acid
  containing total count.
  e.g. A[[i]][[j]] contains total count of j-th amino acid in i-th ORF.

• Format phi.df:
  A data frame A contains two columns ORF and phi.value.
  e.g. A[i,] is for i-th ORF.

• Format reu13.df:
  A named list A contains amino acids. Each element is a data frame summarizing ORF and
  expression. The data frame has four to five columns including ORF, phi (expression), Pos
  (amino acid position), Codon (synonymous codon), and Codon.id (synonymous codon id, for
  computing only). Note that Codon.id may miss in some cases.
  e.g. A[[i]][17,] is the 17-th recode of i-th amino acid.

• Format reu13.list:
  A named list A contains ORFs. Each element is a named list A[[i]] contains amino acids.
  Each element of nested list A[[i]][[j]] is a position vector of synonymous codon.
  e.g. A[[i]][[j]][k] is the k-th synonymous codon position of j-th amino acid in the i-th
  ORF.

• Format scuo:
  A data frame of 8 named columns includes AA (amino acid), ORF, C1, ..., C6 where C*’s are
  for codon counts.

• Format seq.string:
  Default outputs of read.fasta() of seqinr package. A named list A contains ORFs. Each
  element of the list is a long string of a ORF.
  e.g. A[[i]][1] or A[[i]] is the sequence of i-th ORF.

• Format seq.data:
  Converted from seq.string format. A named list A contains ORFs. Each element of the list
  A[[i]] is a string vector. Each element of the vector is a codon string.
  e.g. A[[i]][j] is i-th ORF and j-th codon.

• Format phi.obs:
  A named vector A of observed expression values and possibly with measurement errors.
  e.g. A[i] is the observed phi value of i-th ORF.

• Format y:
  A named list A contains amino acids. Each element of the list A[[i]] is a matrix where ORFs
  are in row and synonymous codons are in column. The element of the matrix contains codon
  counts.
  e.g. A[[i]][j, k] is the count for i-th amino acid, j-th ORF, and k-th synonymous codon.
Datasets

- **Format y.list:**
  A named list `A` contains ORFs. Each element of the list `A[[i]]` is a named list `A[[i]][[j]]` contains amino acids. The element of amino acids list is a codon count vector.
  e.g. `A[[i]][[j]][k]` is the count for `i`-th ORF, `j`-th amino acid, and `k`-th synonymous codon.

**Author(s)**

Wei-Chen Chen <wccsnow@gmail.com>.

**References**

[https://github.com/snoweye/cubfits/](https://github.com/snoweye/cubfits/)

---

**Datasets**

**Datasets for Demonstrations**

**Description**

Examples of toy data to test and demonstrate cubfits.

**Usage**

```
Ex.
```

**Format**

All are in list formats.

**Details**

`bInit` contains two sets (`roc` and `rocnse`) of initial coefficients including mutation and selection parameters for 3 amino acids 'A', 'C', and 'D' in matrix format. Both sets are in `b` format.

`ex.train` contains a training set of 100 sequences including 3 `reu13.df` (codon counts in `reu13` data frame format divided by amino acids), 3 `y` (codon counts in simplified data frame format divided by amino acids), 3 `n` (total amino acid counts in vector format divided by amino acids), and `phi.Obs` (observed phi values in vector format).

`ex.test` contains a testing set of the other 100 sequences in the same format of `ex.train`.

**Author(s)**

Wei-Chen Chen <wccsnow@gmail.com>.

**References**

[https://github.com/snoweye/cubfits/](https://github.com/snoweye/cubfits/)
Estimate Phi

See Also

init.function(), cubfits(), cubpred(), and cubappr().

Examples

## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

str(bInit)
str(ex.test)
str(ex.train)

## End(Not run)

---

Estimate Phi Initialization of Phi (Generic)

Description

This generic function estimates Phi (expression value) either by posterior mean (PM) or by maximum likelihood estimator (MLE) depending on options set by init.function().

Usage

estimatePhi(fitlist, reu13.list, y.list, n.list,
E.Phi = .CF.OP$E.Phi, lower.optim = .CF.OP$lower.optim,
upper.optim = .CF.OP$upper.optim,
lower.integrate = .CF.OP$lower.integrate,
upper.integrate = .CF.OP$upper.integrate, control = list())

Arguments

fitlist an object of format b.
reu13.list an object of format reu13.list.
y.list an object of format y.list.
n.list an object of format n.list.
E.Phi potential expected value of Phi.
lower.optim lower bound to optim().
upper.optim upper bound to optim().
lower.integrate lower bound to integrate().
upper.integrate upper bound to integrate().
control control options to optim().
**Details**

estimatePhi() is a generic function first initialized by `init.function()`, then it estimates Phi accordingly. By default, `.CF.CT$init.Phi` sets the method PM for the posterior mean.

PM uses a flat prior and integrate() to estimate Phi. While, MLE uses optim() to estimate Phi which may have boundary solutions for some sequences.

**Value**

Estimated Phi for every sequence is returned.

**Author(s)**

Wei-Chen Chen `<wccsnow@gmail.com>`.

**References**

https://github.com/snoweye/cubfits/

**See Also**

`init.function()` and `fitMultinom()`.

**Examples**

```r
# Not run:
suppressMessages(library(cubfits, quietly = TRUE))
set.seed(1234)

# Convert data.
reu13.list <- convert.reu13.df.to.list(ex.test$reu13.df)
y.list <- convert.y.to.list(ex.test$y)
n.list <- convert.n.to.list(ex.test$n)

# Get phi.Init.pred
init.function(model = "roc")
fitlist <- fitMultinom(ex.train$reu13.df, ex.train$phi.Obs, ex.train$y, ex.train$n)
phi.Init.pred <- estimatePhi(fitlist, reu13.list, y.list, n.list,
                           E.Phi = median(ex.test$phi.Obs),
                           lower.optim = min(ex.test$phi.Obs) * 0.9,
                           upper.optim = max(ex.test$phi.Obs) * 1.1)

# End(Not run)
```
Fit Multinomial

**Description**

This generic function estimates $b$ (mutation (log(mu)) and selection (Delta.t) parameters) depending on options set by `init.function()`.

**Usage**

```r
fitmultinom(reu13.df, phi, y, n, phi.new = NULL, coefstart = NULL)
```

**Arguments**

- `phi`: an object of format `phi.Obs`.
- `y`: an object of format `y`.
- `n`: an object of format `n`.
- `phi.new`: an object of format `phi.Obs` for MCMC only.
- `coefstart`: initial value for $b$ (mutation (log(mu)) and selection (Delta.t) parameters) only used in `vglm()`.

**Details**

`fitMultinom()` fits a multinomial logistic regression via vector generalized linear model fitting, `vglm()`. By default, for each amino acids, the last codon (order by characters) is assumed as a based line, and other codons are compared to the based line relatively.

In MCMC, `phi.new` are new proposed expression values and used to propose new $b$. The `coefstart` is used to avoid randomization of estimating $b$ in `vglm()`, and speed up computation.

**Value**

A list of format `b` is returned which are modified from the returns of `vglm()`. Mainly, it includes `b$coefficient` (parameters in vector), `b$coef.mat` (parameters in matrix), and `b$R` (covariance matrix of parameters, *R* matrix in QR decomposition).

**Author(s)**

Wei-Chen Chen <wccsnow@gmail.com>.

**References**

[https://github.com/snoweye/cubfits/](https://github.com/snoweye/cubfits/)

See Also

`init.function()` and `estimatePhi()`.

Examples

```r
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))
set.seed(1234)

# Convert data.
reu13.list <- convert.reu13.df.to.list(ex.test$reu13.df)
y.list <- convert.y.to.list(ex.test$y)
n.list <- convert.n.to.list(ex.test$n)

# Get phi.Init.pred
init.function(model = "roc")
fitlist <- fitMultinom(ex.train$reu13.df, ex.train$phi.Obs, ex.train$y, ex.train$n)
phi.Init.pred <- estimatePhi(fitlist, reu13.list, y.list, n.list, 
E.Phi = median(ex.test$phi.Obs),
lower.optim = min(ex.test$phi.Obs) * 0.9,
upper.optim = max(ex.test$phi.Obs) * 1.1)

## End(Not run)
```

Description

These utility functions generate and summarize sequence strings into several useful formats such as `reu13.df`, `y`, and `n`, etc.

Usage

```r
gen.reu13.df(seq.string, phi.df = NULL, aa.names = .CF.GV$amino.acid, 
  split.S = TRUE, drop.X = TRUE, drop.MW = TRUE, 
  drop.1st.codon = TRUE)
gen.y(seq.string, aa.names = .CF.GV$amino.acid, 
  split.S = TRUE, drop.X = TRUE, drop.MW = TRUE)
gen.n(seq.string, aa.names = .CF.GV$amino.acid, 
  split.S = TRUE, drop.X = TRUE, drop.MW = TRUE)
gen.reu13.list(seq.string, aa.names = .CF.GV$amino.acid, 
  split.S = TRUE, drop.X = TRUE, drop.MW = TRUE, 
  drop.1st.codon = TRUE)
gen.phi.Obs(phi.df)
gen.scuo(seq.string, aa.names = .CF.GV$amino.acid, 
  split.S = TRUE, drop.X = TRUE, drop.MW = TRUE)
```
Generating Utility

Arguments

- `seq.string` a list of sequence strings.
- `phi.df` a `phi.df` object returned from `read.phi.df()`.
- `aa.names` a vector contains amino acid names for analysis.
- `split.S` split amino acid 'S' if any.
- `drop.X` drop amino acid 'X' if any.
- `drop.MW` drop amino acid 'M' and 'W' if any.
- `drop.1st.codon` if drop the first codon.

Details

These functions mainly take inputs of sequence strings `seq.string` or `phi.df` and turn them into corresponding format.

Value

The outputs are data structure in corresponding formats. See `AllDataFormats` for details.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

[https://github.com/snoweye/cubfits/](https://github.com/snoweye/cubfits/)

See Also

`AllDataFormats`, `read.seq()`, `read.phi.df()`, and `convert.seq.data.to.string()`.

Examples

```r
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

seq.data <- read.seq(get.expath("seq_200.fasta"))
phi.df <- read.phi.df(get.expath("phi_200.tsv"))
aa.names <- c("A", "C", "D")

# Read in from FASTA file.
seq.string <- convert.seq.data.to.string(seq.data)
reul3.df <- gen.reul3.df(seq.string, phi.df, aa.names)
reul3.list.new <- gen.reul3.list(seq.string, aa.names)
y <- gen.y(seq.string, aa.names)
n <- gen.n(seq.string, aa.names)
scko <- gen.scuo(seq.string, aa.names)

# Convert to list format.
reul3.list <- convert.reul3.df.to.list(reul3.df)
```
y.list <- convert.y.to.list(y)
n.list <- convert.n.to.list(n)

## End(Not run)

---

Initial Generic Functions

Initial Generic Functions of Codon Usage Bias Fits

Description

Initial generic functions for model fitting/approximation/prediction of `cubfits`.

Usage

```r
init.function(model = .CF.CT$model[1],
              type.p = .CF.CT$type.p[1],
              type.Phi = .CF.CT$type.Phi[1],
              model.Phi = .CF.CT$model.Phi[1],
              init.Phi = .CF.CT$init.Phi[1],
              init.fit = .CF.CT$init.fit[1],
              parallel = .CF.CT$parallel[1],
              adaptive = .CF.CT$adaptive[1])
```

Arguments

- `model` main fitted model.
- `type.p` proposal method for hyper-parameters.
- `type.Phi` proposal method for Phi (true expression values).
- `model.Phi` prior of Phi.
- `init.Phi` initial methods for Phi.
- `init.fit` how is coefficient initialed in `vglm()` of `VGAM`.
- `parallel` parallel functions.
- `adaptive` method for adaptive MCMC.

Details

This function mainly takes the options, find the according generic functions, and assign those functions to `.cubfitsEnv`. Those generic functions can be executed accordingly later within functions for MCMC or multinomial logistic regression such as `cubfits()`, `cubappr()`, and `cubpred()`. By default, those options are provided by `.CF.CT` which also leaves rooms for extensions of more complicated models and further optimizations.

It is supposed to call this function before running any MCMC or multinomial logistic regression. This function may affect `cubfits()`, `cubpred()`, `cubappr()`, `estimatePhi()`, and `fitMultinom()`.

- `model` is the main fitting model, currently only `roc` is fully supported.
**Initial Generic Functions**

- `type.p` is for proposing hyper-parameters in Gibb sampler. Currently, `lognormal_fix` is suggested where mean 1 is fixed for log normal distribution. Conjugated prior and flat prior exist and are easily available in this step.

- `type.Phi` is for proposing Phi (expression values) in the random walk chain updates. Only, `RW_Norm` is supported. Usually, the acceptance ratio can be adapted within 25% and 50% controlled by `CF.AC` if `adaptive = simple`.

- `model.Phi` is for the distribution of Phi. Typically, log normal distribution `lognormal` is assumed.

- `init.Phi` is a way to initial Phi. Posterior mean `PM` is recommended which avoid boundary values.

- `init.fit` is a way of initial coefficients to fit mutation and selection coefficients (log mu and Delta.t or omega) in `vglm()`. Option `current` means the b (log(mu) and Delta.t) of current MCMC iteration is the initial values, while `random` means `vglm()` provides the initial values.

- `parallel` is a way of parallel methods to speed up code. `lapply()` is used and no parallel; `mclapply()` of `parallel` is used and good for shared memory machines; `task.pull` means `task.pull()` of `pbdMPI` is used and good for heterogeneous machines; `pbdlapply()` of `pbdMPI` is used and good for homogeneous machines. Among those, `task.pull` is tested thoroughly and is the most reliable and efficient method.

- `adaptive` is a way for adaptive MCMC that propose better mixing distributions for random walks of Phi. The simple method is suggested and only the proposal distribution of Phi (`type.Phi = RW_Norm`) is adjusted gradually.

**Value**

Return an invisible object which is a list contain all generic functions according to the input options. All functions are also assigned in the `.cubfitsEnv` for later evaluations called by MCMC or multinomial logistic regression.

**Note**

Note that all options are taken default values from the global control object `.CF.CT`, so one can utilize/alter the object’s values to adjust those affected functions.

Note that phi.0bs should be scaled to mean 1 before applying to MCMC.

**Author(s)**

Wei-Chen Chen `<wccsnow@gmail.com>`.

**References**

[https://github.com/snoweye/cubfits/](https://github.com/snoweye/cubfits/)

**See Also**

`.CF.CT`, `.CF.CT.cubfits()`, `cubpred()`, and `cubappr()`.
Examples

```r
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))
set.seed(1234)

# Convert data.
reu13.list <- convert.reu13.df.to.list(ex.test$reu13.df)
y.list <- convert.y.to.list(ex.test$y)
n.list <- convert.n.to.list(ex.test$n)

# Get phi.Init.pred
init.function(model = "roc")
fitlist <- fitMultinom(ex.train$reu13.df, ex.train$phi.Obs, ex.train$y,
ex.train$n)
phi.Init.pred <- estimatePhi(fitlist, reu13.list, y.list, n.list,
    E.Phi = median(ex.test$phi.Obs),
    lower.optim = min(ex.test$phi.Obs) * 0.9,
    upper.optim = max(ex.test$phi.Obs) * 1.1)

## End(Not run)
```

Description

These utility functions read and write data of FASTA and phi.df formats.

Usage

```r
read.seq(file.name, forceDNAtolower = FALSE, convertDNAtoupper = TRUE)
write.seq(seq.data, file.name)

read.phi.df(file.name, header = TRUE, sep = "\t", quote = ""
write.phi.df(phi.df, file.name)

get.expath(file.name, path.root = ".//ex_data/", pkg = "cubfits")
```

Arguments

- `file.name` a file name to read or write.
- `forceDNAtolower` an option passed to read.fasta() of `seqinr` package.
- `convertDNAtoupper` force everything in upper case.
- `header` an option passed to read.table().
sep an option passed to `read.table()`.
quote an option passed to `read.table()`.
seq.data a `seq.data` object.
phi.df a `phi.df` object.
path.root root path for the file name relatively to the pkg.
pkg package name for the path of root.

Details

`read.seq()` and `write.seq()` typically read and write FASTA files (DNA ORFs or sequences).
`read.phi.df()` and `write.phi.df()` typically read and write phi.df files (expression values of ORFs or sequences).
`get.expath()` is only for demonstration returning a full path to the file.

Value

`read.seq()` returns an object of `seq.data` format which can be converted to `seq.string` format later via `convert.seq.data.to.string()`.
`read.phi.df()` returns an object of `phi.df` format which contains expression values.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

`convert.seq.data.to.string()`.

Examples

```r
### Not run:
suppressMessages(library(cubfits, quietly = TRUE))

seq.data <- read.seq(get.expath("seq_200.fasta"))
phi.df <- read.phi.df(get.expath("phi_200.tsv"))
aa.names <- c("A", "C", "D")

# Read in from FASTA file.
seq.string <- convert.seq.data.to.string(seq.data)

### End(Not run)
```
Mixed Normal Optimization

Description

Constrained optimization for mixed normal in 1D and typically for 2 components.

Usage

```r
mixnormerr.optim(X, K = 2, param = NULL)
dmixnormerr(x, param)
```

Arguments

- `X` a gene expression data matrix of dimension `N * R` which has `N` genes and `R` replicates.
- `K` number of components to fit.
- `x` vector of quantiles.
- `param` parameters of `mixnormerr`, typically the element `param` of the `mixnormerr.optim()` returning object.

Details

The function `mixnormerr.optim()` maximizes likelihood using `constrOptim()` based on the gene expression data `X` (usually in log scale) for `N` genes and `R` replicates (NA is allowed). The likelihood of each gene expression is a `K = 2` component mixed normal distribution \( \sum_k p_k N(\mu_k, \sigma_k^2 + \sigma_e^2) \) with measurement errors of the replicates \( N(0, \sigma_e^2) \).

The \( \sigma_k^2 \) is as the error of random component and the \( \sigma_e^2 \) is as the error of fixed component. Both are within a mixture model of two normal distributions.

The function `dmixnormerr()` computes the density of the mixed normal distribution.

`param` is a parameter list and contains five elements: `K` for number of components, `prop` for proportions, `mu` for centers of components, `sigma2` for variance of components, and `sigma2.e` for variance of measurement errors.

Value

`mixnormerr.optim()` returns a list containing three main elements `param` is the final results (MLEs), `param.start` is the starting parameters, and `optim.ret` is the original returns of `constrOptim()`.

Note

This function is limited for small `K`. An equivalent EM algorithm should be done in a more stable way for large `K`. 
Plotbin

Author(s)
Wei-Chen Chen <wccsnow@gmail.com>.

References
https://github.com/snoweye/cubfits/

See Also
print.mixnormerr(), simu.mixnormerr().

Examples

```r
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

### Get individual of phi.Obs.
GM <- apply(yassour[, -1], 1, function(x) exp(mean(log(x[x != 0]))) * sum(GM) * 15000)
phi.Obs.all <- yassour[, -1] / sum(GM)
phi.Obs.all[phi.Obs.all == 0] <- NA

### Run optimization.
X <- log(as.matrix(phi.Obs.all))
param.init <- list(K = 2, prop = c(0.95, 0.05), mu = c(-0.59, 3.11),
                  sigma2 = c(1.40, 0.59), sigma2.e = 0.03)
ret <- mixnormerr.optim(X, K = 2, param = param.init)
print(ret)

## End(Not run)
```

---

**Plotbin**  
*Plot Binning Results*

Description
Plot binning results to visualize the effects of mutation and selection along with expression levels empirically.

Usage

```r
prop.bin.roc(reu13.df, phi.Obs = NULL, nclass = 20)

plotbin(ret.bin, ret.model = NULL, main = NULL, xlab = "Production Rate (log10)", ylab = "Proportion",
        xlab = NULL, lty = 1, x.log10 = TRUE, stderr = FALSE, ...)
```
Arguments

- `reu13.df` a `reu13.df` object.
- `phi.Obs` a `phi.Obs` object.
- `nclass` number of binning classes across the range of `phi.Obs`.
- `ret.bin` binning results from `prop.bin.roc()`.
- `ret.model` model results from `prop.model.roc()`.
- `main` an option passed to `plot()`.
- `xlab` an option passed to `plot()`.
- `ylab` an option passed to `plot()`.
- `xlim` range of X-axis.
- `lty` line type if `ret.model` is provided.
- `x.log10` log10() transformation of X-axis.
- `stderr` plot stand error instead of stand deviation.
- ... options passed to `plot()`.

Details

The function `plotbin()` plots the binning results `ret.bin` returned from `prop.bin.roc()`. Fitted curves may be added if `ret.model` is provided which can be obtained from `prop.model.roc()`.

`plotaddmodel()` can append model later if `ret.model` is not provided to `plotbin()`.

Currently, only ROC model is supported. Colors are controlled by `.CF.PT`.

Value

A binning plot is drawn.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

`plotmodel()` and `prop.model.roc()`.

Examples

```r
## Not run:
demo(plotbin, 'cubfits', ask = F, echo = F)

## End(Not run)
```
Description

Plot model results to visualize the effects of mutation and selection along with expression levels. The model can be fitted by MCMC or multinomial logistic regression.

Usage

```r
prop.model.roc(bInit, phi.Obs.lim = c(0.01, 10), phi.Obs.scale = 1,
               nclass = 40, x.log10 = TRUE)

plotmodel(ret.model, main = NULL,
          xlab = "Production Rate (log10)", ylab = "Proportion",
          xlim = NULL, lty = 1, x.log10 = TRUE, ...)

plotaddmodel(ret.model, lty, u.codon = NULL, color = NULL,
             x.log10 = TRUE)
```

Arguments

- `bInit`: a `b` object.
- `phi.Obs.scale`: optional scaling factor.
- `nclass`: number of binning classes across the range of `phi.Obs`.
- `x.log10`: `log10()` transformation of X-axis.
- `ret.model`: model results from `prop.model.roc()`.
- `main`: an option passed to `plot()`.
- `xlab`: an option passed to `plot()`.
- `ylab`: an option passed to `plot()`.
- `xlim`: range of X-axis.
- `lty`: line type.
- `u.codon`: unique synonymous codon names.
- `color`: a color vector for unique codon, typically returns of the internal function `get.color()`.
- `...`: options passed to `plot()`.

Details

The function `plotmodel()` plots the fitted curves obtained from `prop.model.roc()`.

The function `plotaddmodel()` can append model curves to a binning plot provided unique synonymous codons and colors are given. This function is nearly for an internal call within `plotmodel()`, but is exported and useful for workflow.

Currently, only ROC model is supported. Colors are controlled by `.CF.PT`. 
Plotprxy

Predictive X-Y Plot

Description
This utility function provides a basic plot of production rates.

Usage

```r
plotprxy(x, y, x.ci = NULL, y.ci = NULL,
        log10.x = TRUE, log10.y = TRUE,
        add.lm = TRUE, add.one.to.one = TRUE, weights = NULL,
        add.legend = TRUE,
        xlim = NULL, ylim = NULL,
        xlab = "Predicted Production Rate (log10)",
        ylab = "Observed Production Rate (log10)",
        main = NULL)
```

Arguments

- **x**: expression values.
- **y**: expression values, of the same length of **x**.
- **x.ci**: confidence interval of **x**, of dimension `length(x) * 2`, for outliers labeling.
- **y.ci**: confidence interval of **y**, of dimension `length(y) * 2`, for outliers labeling.
- **log10.x**: `log10()` and mean transformation of **x** axis.
log10(y) log10() and mean transformation of y axis.
add.lm if add lm() fit.
add.one.to.one if add one-to-one line.
weights weights to lm().
add.legend if add default legend.
xlim limits of x-axis.
ylim limits of y-axis.
xlab an option passed to plot().
ylab an option passed to plot().
main an option passed to plot().

Details
As the usual X-Y plot where x and y are expression values.
If add.lm = TRUE and weights are given, then both ordinary and weighted least squares results will be plotted.

Value
A scatter plot with a fitted lm() line and R squared value.

Author(s)
Wei-Chen Chen <wccsnow@gmail.com>.

References
https://github.com/snoweye/cubfits/

See Also
plotbin() and plotmodel().

Examples
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

y.scuo <- convert.y.to.scuo(ex.train$y)
SCUO <- calc_scuo_values(y.scuo)$SCUO
plotprxy(ex.train$phi.Obs, SCUO)
## End(Not run)
Posterior Results of Yassour2009

Posterior Results of Yassour 2009 Yeast Experiment Dataset

Description

Output summarized from MCMC posterior results analyzing Yassour 2009 data.

Usage

```r
yassour.pm.fits
yassour.pm.appr
yassour.info
```

Format

These are lists containing several posterior means: \(E.\Phi i\) for expected expression, \(bInitList.roc\) for parameters, \(AA.prob\) for proportion of amino acids, \(sigmaW\) for standard error of measure errors, and \(gene.length\) for gene length.

Details

\(yassour.pm.fits\) and \(yassour.pm.appr\) are the MCMC output of with/without observed expression, respectively. Both contain posterior means of expected expressions and coefficient parameters: \(E.\Phi i\) and \(bInitList.roc\) are scaled results such that each MCMC iteration has mean 1 at \(E.\Phi i\).

\(yassour.info\) contains sequences information (Yeast): \(AA.prob\) and \(gene.length\) are summarized from corresponding genes in the analysis.

Note that some of genes may not have good quality of expression or sequence information, so those genes are dropped from \(yassour\) dataset.

References

https://github.com/snoweye/cubfits/

See Also

\(yassour\)

Examples

```r
## Not run:
str(yassour.pm.fits)
str(yassour.pm.appr)
str(yassour.pm.info)

## End(Not run)
```
Description

A Class `mixnormerr` is declared in `cubfits`, and this is the function to print and summary objects.

Usage

```r
## S3 method for class 'mixnormerr'
print(x, digits = max(4,getOption("digits") - 3), ...)
```

Arguments

- `x` an object with the class attributes.
- `digits` for printing out numbers.
- `...` other possible options.

Details

This is an useful function for summarizing and debugging.

Value

The results will cat or print on the STDOUT by default.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

[https://github.com/snoweye/cubfits/](https://github.com/snoweye/cubfits/)

See Also

`mixnormerr.optim()`.

Examples

```r
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

### Get individual of phi.Obs.
GMM <- apply(yassour[, -1], 1, function(x) exp(mean(log(x[x != 0]))) )
phi.Obs.all <- yassour[, -1] / sum(GM) * 15000
phi.Obs.all[phi.Obs.all == 0] <- NA
```
### Generate Randomized SCUO Index

**Description**

Generate randomized SCUO indices in log normal distribution, but provides original unchanged SCUO order.

**Usage**

```r
scuo.random(SCUO, phi.Obs = NULL, meanlog = .CF.PARAM$phi.meanlog, sdlog = .CF.PARAM$phi.sdlog)
```

**Arguments**

- **SCUO**: SCUO index returned from `calc_scuo_values()`.
- **phi.Obs**: optional object of format `phi.Obs`.
- **meanlog**: mean of log normal distribution.
- **sdlog**: std of log normal distribution.

**Details**

This function takes SCUO indices (outputs of `calc_scuo_values()`) computes the rank of them, generates log normal random variables, and replaces SCUO indices by those variables in the same rank orders. Typically, these random variables are used to replace expression values when either no expression is observed or for the purpose of model validation.

If `phi.Obs` is provided, the mean and std of log(`phi.Obs`) are used for log normal random variables. Otherwise, `meanlog` and `sdlog` are used.

The default `meanlog` and `sdlog` was estimated from `yassour` dataset.

**Value**

A vector of log normal random variables is returned.

**Author(s)**

Wei-Chen Chen <wccsnow@gmail.com>.
Rearrangement Utility

References

https://github.com/snoweye/cubfits/

See Also

calc_scuo_values(), yassour.

Examples

```r
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

### example dataset.
y.scuo <- convert.y.to.scuo(ex.train$y)
SCUO <- calc_scuo_values(y.scuo)$SCUO
plotprxy(ex.train$phi.Obs, SCUO)

### yassour dataset.
GM <- apply(yassour[, -1], 1, function(x) exp(mean(log(x[x != 0]))))
phi.Obs <- GM / sum(GM) * 15000
mean(log(phi.Obs))
sd(log(phi.Obs))
ret <- scuo.random(SCUO, meanlog = -0.441473, sdlog = 1.393285)
plotprxy(ret, SCUO)

## End(Not run)
```

Rearrangement Utility  Rearrange Data Structure by ORF Names

Description

These utility functions rearrange data in the order of ORF names.

Usage

```r
rearrange.reu13.df(reu13.df)
rearrange.y(y)
rearrange.n(n)
rearrange.phi.Obs(phi.Obs)
```

Arguments

- `reu13.df`: a list of `reu13.df` data frames divided by amino acids.
- `y`: a list of `y` data frames divided by amino acids.
- `n`: a list of `n` vectors divided by amino acids.
Details

These utility functions take inputs and return ordered outputs. It is necessary to rearrange data in a right order of ORF names which avoids subsetting data frame within MCMC and improve performance.

Value

The outputs are in the same format of inputs except the order of data is sorted by ORF names.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

AllDataFormats, convert.n.to.list(), convert.reu13.df.to.list(), and convert.y.to.list().

Examples

```r
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

reu13.df <- rearrange.reu13.df(ex.train$reu13.df)
y <- rearrange.y(ex.train$y)
n <- rearrange.n(ex.train$n)
phi.Obs <- rearrange.phi.Obs(ex.train$phi.Obs)

## End(Not run)
```

SCUO Index Function for Synonymous Codon Usage Order (SCUO) Index

Description

Calculate the Synonymous Codon Usage Order (SCUO) index for each gene. Used as a substitute for expression in cases of without expression measurements.

Usage

```r
calc_scuo_values(codon.counts)
```

Arguments

codon.counts an object of format scuo.
Selection on Codon Usage

Details
This function computes SCUO index for each gene. Typically, this method is completely based on entropy and information theory to estimate expression values of sequences according to their codon information.

Value
SCUO indices are returned.

Author(s)
Drew Schmidt.

References
http://www.tandfonline.com/doi/abs/10.1080/03081070500502967

See Also
scuo.random(), calc_cai_values(), calc_scu_values().

Examples
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))

y.scuo <- convert.y.to.scuo(ex.train$y)
SCUO <- calc_scu_values(y.scuo)$SCUO
plotprxy(ex.train$phi.0bs, SCUO, ylab = "SCUO (log10)"

## End(Not run)

Selection on Codon Usage

Function for Selection on Codon Usage (SCU)

Description
Calculate the average translational selection per transcript include mSCU and SCU (if gene expression is provided) for each gene.

Usage
calc_scu_values(b, y.list, phi.0bs = NULL)
Arguments

- `b` an object of format `b`.
- `y.list` an object of format `y.list`.
- `phi.Obs` an object of format `phi.Obs`, for SCU only.

Details

This function computes SCU and mSCU for each gene. Typically, this method is completely based on estimated parameters of mutation and selection such as outputs of MCMC or `fitMultinom()`.

Value

A list with two named elements SCU and mSCU are returned.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References


See Also

calc_scuo_values(), calc_cai_values().

Examples

```r
## Not run:
library(cubfits, quietly = TRUE)

b <- bInit$roc
phi.Obs <- ex.train$phi.Obs
y <- ex.train$y
y.list <- convert.y.to.list(y)
mscu <- calc_scu_values(b, y.list, phi.Obs)$mscu
plot(mscu, log10(phi.Obs), main = "Expression vs mSCU",
    xlab = "mSCU", ylab = "Expression (log10)"
)

### Compare with CAI with weights seqinr::cubtab$sc.
library(seqinr, quietly = TRUE)
w <- caitab$sc
names(w) <- codon.low2up(rownames( Caitab))
CAI <- calc_cai_values(y, y.list, w = w)$CAI

plot(mscu, CAI, main = "CAI vs mSCU",
    xlab = "mSCU", ylab = "CAI")

## End(Not run)
```
Description

These utility functions generate data for simulation studies including fake ORFs and expression values.

Usage

```r
simu.orf(n, bInit, phi.Obs = NULL, AA.prob = NULL, orf.length = NULL,
        orf.names = NULL, model = .CF.CT$model)
simu.phi.Obs(Phi, sigmaW.lim = 1)
simu.mixnormerr(n, param)
```

Arguments

- `n`: number of ORFs or sequences.
- `bInit`: parameters of mutation and selection of format `b`.
- `AA.prob`: proportion of amino acids.
- `orf.length`: lengths of ORFs.
- `orf.names`: names of ORFs.
- `model`: model to be simulated.
- `Phi`: expression values (potentially true expression).
- `sigmaW.lim`: std of measurement errors (between Phi and phi.Obs).
- `param`: as in `dmixnormerr()`

Details

- `simu.orf()` generates ORFs or sequences based on the `bInit` and `phi.Obs`.
  If `phi.Obs` is omitted, then standard log normal random variables are instead.
  If `AA.prob` is omitted, then uniform proportion is assigned.
  If `orf.length` is omitted, then 10 to 20 codons are randomly assigned.
  If `orf.names` is omitted, then "ORF1" to "ORFn" are assigned.
- `simu.phi.Obs()` generates `phi.Obs` by adding normal random errors to `Phi`, and errors have mean 0 and standard deviation `sigmaW.lim`.
- `simu.mixnormerr()` generates `Phi` according to the `param`, and adds normal random errors to `Phi`.  

Value

simu.orf() returns a list of format seq.data.
simu.phi.Obs() returns a vector of format phi.Obs.
simu.mixnormerr() returns a list contains three vectors of length n: one for expected gene expression Phi, one for observed gene expression phi.Obs, and one for the component id id.K.

Author(s)

Wei-Chen Chen <wccsnow@gmail.com>.

References

https://github.com/snoweye/cubfits/

See Also

read.seq(), read.phi.df(), write.seq(), write.phi.df(), and mixnormerr.optim().

Examples

```r
## Not run:
suppressMessages(library(cubfits, quietly = TRUE))
set.seed(1234)

data.frame with 6303 rows and 5 columns:
orf is for gene names in character, and
ypd.P1, ypd.R, ypd1.P1, and ypd1.R are gene expressions in positive double corresponding to 4 controlled Yeast experiments.
```
Details

The original data are available as the URL of the section of Source next. As the section of Examples next, data are selected from SD3.xls and reordered by ORF.

For further analysis, the Examples section also provides how to convert them to phi.Obs values either in geometric means or individually.

Source

http://www.pnas.org/content/early/2009/02/10/0812841106
http://www.pnas.org/content/vol0/issue2009/images/data/0812841106/DCSupplemental/SD3.xls


References


Examples

```r
# Not run:
### SD3.xls is available from the URL provided in the References.
da <- read.table("SD3.xls", header = TRUE, sep = "\t", quote = "", stringsAsFactors = FALSE)

### Select ORF, YPD0.1, YPD0.2, YPD15.1, YPD15.2.
da <- da[, c(1, 8, 9, 10, 11)]
colnames(da) <- c("ORF", "YPD0.1", "YPD0.2", "YPD15.1", "YPD15.2")

### Drop inappropriate values (NaN, NA, Inf, -Inf, and 0).
tmp <- da[, 2:5]
id.tmp <- rowSums(is.finite(as.matrix(tmp)) & tmp != 0) >= 3
tmp <- da[id.tmp, 1:5]
yassour <- tmp[order(tmp$ORF),]  # cubfits::yassour

### Get geometric mean of phi.Obs and scaling similar to Wallace (2013).
GM <- apply(yassour[, -1], 1, function(x) exp(mean(log(x[x != 0]))))
phi.Obs <- GM / sum(GM) * 15000

### Get individual of phi.Obs.
GM <- apply(yassour[, -1], 1, function(x) exp(mean(log(x[x != 0]))))
phi.Obs.all <- yassour[, -1] / sum(GM) * 15000
phi.Obs.all[phi.Obs.all == 0] <- NA

# End(Not run)
```
Index

+ Topic dataformats
  Data Formats, 18
+ Topic datasets
  Controls, 7
  Datasets, 20
  Posterior Results of Yasour2009, 36
  Yasour2009, 44
+ Topic main function
  CUB Model Approximation, 12
  CUB Model Fits, 14
  CUB Model Prediction, 16
+ Topic package
  cubfits-package, 2
+ Topic plotting
  Plotbin, 31
  Plotmodel, 33
  Plotprxy, 34
+ Topic summary
  Print, 37
+ Topic tool
  Codon Adaptation Index, 5
  Estimate Phi, 21
  Fit Multinomial, 23
  Initial Generic Functions, 26
  Randomize SCUO Index, 38
  SCUO Index, 40
  Selection on Codon Usage, 41
  Simulation Tool, 43
+ Topic utility
  Asymmetric Laplace Distribution, 3
  Converting Utility, 10
  Generating Utility, 24
  Input and Output Utility, 28
  Mixed Normal Optimization, 30
  Rearrangement Utility, 39
  .CF.AC, 27
  .CF.AC (Controls), 7
  .CF.CONF (Controls), 7
  .CF.CT, 26, 27
  .CF.CT (Controls), 7
  .CF.DP, 13, 15, 17
  .CF.DP (Controls), 7
  .CF.GV (Controls), 7
  .CF.OP (Controls), 7
  .CF.PARAM (Controls), 7
  .CF.PT, 32, 33
  .CF.PT (Controls), 7
  .CO.CT (Controls), 7
  .cubfitsEnv, 26, 27
  .cubfitsEnv (Controls), 7

AllDataFormats, 11, 25, 40
AllDataFormats (Data Formats), 18
asl.optim (Asymmetric Laplace Distribution), 3
Asymmetric Laplace Distribution, 3
b, 11–15, 17, 20, 21, 23, 27, 33, 42, 43
b (Data Formats), 18
bInit (Datasets), 20
bVec, 11
bVec (Data Formats), 18
calc_cai_values, 41, 42
calc_cai_values (Codon Adaptation Index), 5
calc_scu_values, 6, 41
calc_scu_values (Selection on Codon Usage), 41
calc_scu_values (SCUO Index), 40
Codon Adaptation Index, 5
codon.low2up (Converting Utility), 10
codon.up2low (Converting Utility), 10
Controls, 7
convert.b.to.bVec (Converting Utility), 10

46
INDEX

<table>
<thead>
<tr>
<th>Function/Package</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>convert.bVec.to.b</td>
<td>10</td>
</tr>
<tr>
<td>convert.n.to.list</td>
<td>40</td>
</tr>
<tr>
<td>convert.n.to.list</td>
<td>10</td>
</tr>
<tr>
<td>convert.reul3.df.to.list</td>
<td>40</td>
</tr>
<tr>
<td>convert.reul3.df.to.list</td>
<td>10</td>
</tr>
<tr>
<td>convert.seq.data.to.string</td>
<td>25, 29</td>
</tr>
<tr>
<td>convert.seq.data.to.string</td>
<td>10</td>
</tr>
<tr>
<td>convert.y.to.list</td>
<td>40</td>
</tr>
<tr>
<td>convert.y.to.list</td>
<td>10</td>
</tr>
<tr>
<td>convert.y.to.scuo</td>
<td>10</td>
</tr>
<tr>
<td>Cuberating Utility</td>
<td>10</td>
</tr>
<tr>
<td>Cub Model Approximation</td>
<td>12</td>
</tr>
<tr>
<td>Cub Model Fits</td>
<td>14</td>
</tr>
<tr>
<td>Cub Model Prediction</td>
<td>16</td>
</tr>
<tr>
<td>cuba.prpr</td>
<td>3, 9, 16, 18, 21, 26, 27</td>
</tr>
<tr>
<td>cuba.prpr (CUB Model Approximation)</td>
<td>12</td>
</tr>
<tr>
<td>cubfits</td>
<td>3, 9, 13, 18, 21, 26, 27</td>
</tr>
<tr>
<td>cubfits (CUB Model Fits)</td>
<td>14</td>
</tr>
<tr>
<td>cubfits</td>
<td>package</td>
</tr>
<tr>
<td>cubpred</td>
<td>3, 9, 13, 16, 21, 26, 27</td>
</tr>
<tr>
<td>cubpred (CUB Model Prediction)</td>
<td>16</td>
</tr>
<tr>
<td>dasl (Asymmetric Laplace Distribution)</td>
<td>3</td>
</tr>
<tr>
<td>dasla (Asymmetric Laplace Distribution)</td>
<td>3</td>
</tr>
<tr>
<td>Data Formats</td>
<td>18</td>
</tr>
<tr>
<td>DataConverting</td>
<td>13, 16, 18</td>
</tr>
<tr>
<td>DataConverting (Converting Utility)</td>
<td>10</td>
</tr>
<tr>
<td>DataGenerating (Generating Utility)</td>
<td>24</td>
</tr>
<tr>
<td>DataIO</td>
<td>13, 16, 18</td>
</tr>
<tr>
<td>DataIO (Input and Output Utility)</td>
<td>28</td>
</tr>
<tr>
<td>Datasets</td>
<td>20</td>
</tr>
<tr>
<td>dimxnormerr</td>
<td>43</td>
</tr>
<tr>
<td>dimxnormerr (Mixed Normal Optimization)</td>
<td>30</td>
</tr>
<tr>
<td>dna.low2up (Converting Utility)</td>
<td>10</td>
</tr>
<tr>
<td>dna.up2low (Converting Utility)</td>
<td>10</td>
</tr>
<tr>
<td>Estimate Phi</td>
<td>21</td>
</tr>
<tr>
<td>estimatePhi</td>
<td>24, 26</td>
</tr>
<tr>
<td>estimatePhi (Estimate Phi)</td>
<td>21</td>
</tr>
<tr>
<td>ex.test (Datasets)</td>
<td>20</td>
</tr>
<tr>
<td>ex.train (Datasets)</td>
<td>20</td>
</tr>
<tr>
<td>Fit Multinomial</td>
<td>23</td>
</tr>
<tr>
<td>fitMultinom</td>
<td>22, 26, 42</td>
</tr>
<tr>
<td>fitMultinom (Fit Multinomial)</td>
<td>23</td>
</tr>
<tr>
<td>gen.n (Generating Utility)</td>
<td>24</td>
</tr>
<tr>
<td>gen.phi.Obs (Generating Utility)</td>
<td>24</td>
</tr>
<tr>
<td>gen.reul3.df (Generating Utility)</td>
<td>24</td>
</tr>
<tr>
<td>gen.reul3.list (Generating Utility)</td>
<td>24</td>
</tr>
<tr>
<td>gen.scuo (Generating Utility)</td>
<td>24</td>
</tr>
<tr>
<td>gen.y (Generating Utility)</td>
<td>24</td>
</tr>
<tr>
<td>Generating Utility</td>
<td>24</td>
</tr>
<tr>
<td>get.expPath (Input and Output Utility)</td>
<td>28</td>
</tr>
<tr>
<td>init.function</td>
<td>3, 8, 9, 21–24</td>
</tr>
<tr>
<td>init.function (Initial Generic Functions)</td>
<td>26</td>
</tr>
<tr>
<td>Initial Generic Functions</td>
<td>26</td>
</tr>
<tr>
<td>Input and Output Utility</td>
<td>28</td>
</tr>
<tr>
<td>Mixed Normal Optimization</td>
<td>30</td>
</tr>
<tr>
<td>mixnormerr.optim</td>
<td>9, 37, 44</td>
</tr>
<tr>
<td>mixnormerr.optim (Mixed Normal Optimization)</td>
<td>30</td>
</tr>
<tr>
<td>n</td>
<td>11, 12, 14, 16, 17, 20, 23, 24, 39</td>
</tr>
<tr>
<td>n (Data Formats)</td>
<td>18</td>
</tr>
<tr>
<td>n.list</td>
<td>21</td>
</tr>
<tr>
<td>pasl (Asymmetric Laplace Distribution)</td>
<td>3</td>
</tr>
<tr>
<td>pasla (Asymmetric Laplace Distribution)</td>
<td>3</td>
</tr>
<tr>
<td>phi.df</td>
<td>25, 29</td>
</tr>
<tr>
<td>phi.df (Data Formats)</td>
<td>18</td>
</tr>
<tr>
<td>phi.Obs</td>
<td>12, 14, 16, 20, 23, 32, 38, 39, 42–44</td>
</tr>
<tr>
<td>phi.Obs (Data Formats)</td>
<td>18</td>
</tr>
<tr>
<td>plotaddmodel</td>
<td>32</td>
</tr>
<tr>
<td>plotaddmodel (Plotmodel)</td>
<td>33</td>
</tr>
<tr>
<td>Plotbin</td>
<td>31</td>
</tr>
<tr>
<td>plotbin</td>
<td>34, 35</td>
</tr>
<tr>
<td>plotbin (Plotbin)</td>
<td>31</td>
</tr>
<tr>
<td>Plotmodel</td>
<td>33</td>
</tr>
<tr>
<td>plotmodel</td>
<td>32, 35</td>
</tr>
<tr>
<td>plotmodel (Plotmodel)</td>
<td>33</td>
</tr>
<tr>
<td>Plotprxy</td>
<td>34</td>
</tr>
<tr>
<td>plotprxy (Plotprxy)</td>
<td>34</td>
</tr>
<tr>
<td>Posterior Results of Yassour2009</td>
<td>36</td>
</tr>
</tbody>
</table>
Print, 37
print.mixnormerr, 31
print.mixnormerr (Print), 37
prop.bin.roc, 32, 34
prop.bin.roc (Plotbin), 31
prop.model.roc, 32–34
prop.model.roc (Plotmodel), 33
qasl (Asymmetric Laplace Distribution), 3
qasla (Asymmetric Laplace Distribution), 3
Randomize SCUO Index, 38
rasl (Asymmetric Laplace Distribution), 3
rasla (Asymmetric Laplace Distribution), 3
read.phi.df, 25, 44
read.phi.df (Input and Output Utility), 28
read.seq, 11, 25, 44
read.seq (Input and Output Utility), 28
rearrange.n, 11
rearrange.n (Rearrangement Utility), 39
rearrange.phi.Obs (Rearrangement Utility), 39
rearrange.reul3.df, 11
rearrange.reul3.df (Rearrangement Utility), 39
rearrange.y, 11
rearrange.y (Rearrangement Utility), 39
Rearrangement Utility, 39
reu13.df, 10, 12, 14, 16, 20, 23, 24, 32, 39
reu13.df (Data Formats), 18
reu13.list, 21
reu13.list (Data Formats), 18
scuo, 11, 40
scuo (Data Formats), 18
SCUO Index, 40
scuo.random, 41
scuo.random (Randomize SCUO Index), 38
Selection on Codon Usage, 41
seq.data, 11, 29, 44
seq.data (Data Formats), 18
seq.string, 11, 25, 29
seq.string (Data Formats), 18
simu.mixnormerr, 31
simu.mixnormerr (Simulation Tool), 43
simu.orf (Simulation Tool), 43
simu.phi.Obs (Simulation Tool), 43
Simulation Tool, 43
write.phi.df, 44
write.phi.df (Input and Output Utility), 28
write.seq, 44
write.seq (Input and Output Utility), 28
y, 6, 10–12, 14, 16, 17, 20, 23, 24, 39
y (Data Formats), 18
y.list, 6, 21, 42
yassour, 36
yassour (Yassour2009), 44
yassour.info (Posterior Results of Yassour2009), 36
yassour.PM.appr (Posterior Results of Yassour2009), 36
yassour.PM.fits (Posterior Results of Yassour2009), 36
Yassour2009, 44