Package 'tigre'

December 6, 2025

Version 1.65.0

Date 2021-08-04

Title Transcription factor Inference through Gaussian process Reconstruction of Expression

Author Antti Honkela, Pei Gao, Jonatan Ropponen, Miika-Petteri Matikainen, Magnus Rattray, Neil D. Lawrence

Maintainer Antti Honkela <antti.honkela@helsinki.fi>

Depends R (>= 2.11.0), BiocGenerics, Biobase

Imports methods, AnnotationDbi, gplots, graphics, grDevices, stats, utils, annotate, DBI, RSQLite

Suggests drosgenome1.db, puma, lumi, BiocStyle, BiocManager

Description The tigre package implements our methodology of Gaussian process differential equation models for analysis of gene expression time series from single input motif networks. The package can be used for inferring unobserved transcription factor (TF) protein concentrations from expression measurements of known target genes, or for ranking candidate targets of a TF.

License AGPL-3

URL https://github.com/ahonkela/tigre

BugReports https://github.com/ahonkela/tigre/issues

biocViews Microarray, TimeCourse, GeneExpression, Transcription, GeneRegulation, NetworkInference, Bayesian

git_url https://git.bioconductor.org/packages/tigre

git_branch devel

git_last_commit 4b818eb

git_last_commit_date 2025-10-29

Repository Bioconductor 3.23

Date/Publication 2025-12-05

Contents

`tigre-package` *tigre - Transcription factor Inference through Gaussian process Reconstruction of Expression*

Description

This package implements the method of Gao et al. (2008) and Honkela et al. (2010) for Gaussian process modelling single input motif regulatory systems with time-series expression data. The method can be used to rank potential targets of transcription factors based on such data.

Details

Package:	tigre
Type:	Package
Version:	1.12.0
Date:	2012-10-02
License:	A-GPL Version 3

For details of using the package please refer to the Vignette.

Author(s)

Antti Honkela, Pei Gao, Jonatan Ropponen, Miika-Petteri Matikainen, Magnus Rattray, Neil D. Lawrence

Maintainer: Antti Honkela <antti.honkela@hiit.fi>

References

A.~Honkela, P.~Gao, J.~Ropponen, M.~Rattray, and N.~D.~Lawrence. tigre: Transcription factor Inference through Gaussian process Reconstruction of Expression for Bioconductor. *Bioinformatics* 27(7):1026-1027, 2011. DOI: 10.1093/bioinformatics/btr057.

P.~Gao, A.~Honkela, M.~Rattray, and N.~D.~Lawrence. Gaussian process modelling of latent chemical species: applications to inferring transcription factor activities. *Bioinformatics* 24(16):i70–i75, 2008. DOI: 10.1093/bioinformatics/btn278.

A.~Honkela, C.~Girardot, E.~H. Gustafson, Y.-H. Liu, E.~E.~M. Furlong, N.~D. Lawrence, and M.~Rattray. Model-based method for transcription factor target identification with limited data. *Proc Natl Acad Sci USA* 107(17):7793-7798, 2010. DOI: 10.1073/pnas.0914285107.

See Also

puma

Examples

```
## Not run:
  # Load a mmgmos preprocessed fragment of the Drosophila developmental
  # time series
  data(drosophila_gpsim_fragment)

  # Get the target probe names
  library(annotate)
  aliasMapping <- getAnnMap("ALIAS2PROBE",
                    annotation(drosophila_gpsim_fragment))
  twi <- get('twi', env=aliasMapping)
  fbgnMapping <- getAnnMap("FLYBASE2PROBE",
                   annotation(drosophila_gpsim_fragment))
  targetProbe <- get('FBgn0035257', env=fbgnMapping)
```

```
  # Learn the model
  model <- GPLearn(drosophila_gpsim_fragment,
                   TF=twi, targets=targetProbe,
                   useGpdisim=TRUE, quiet=TRUE)

  # Plot it
  GPPlot(model, nameMapping=getAnnMap("FLYBASE",
                   annotation(drosophila_gpsim_fragment)))

## End(Not run)
```

`drosophila_gpsim_fragment` *Fragment of 12 time point Drosophila embryonic development microarray gene expression time series*

Description

Four genes from the 12 time point Drosophila embryonic development Affymetrix microarray gene expression data set by Tomancak et al. (2002).

The data has been processed using `mmgmos` from `puma` package and `processData`.

Usage

```
data(drosophila_gpsim_fragment)
```

Format

An `ExpressionTimeSeries` object with 3 repeats of the 12 time points for 4 probes.

Source

ftp://ftp.fruitfly.org/pub/embryo_tc_array_data/

References

Tomancak, P et al. Systematic determination of patterns of gene expression during Drosophila embryogenesis. *Genome Biol* 3:RESEARCH0088, 2002.

drosophila_mmgmos_fragment *Fragment of 12 time point Drosophila embryonic development microarray gene expression time series*

Description

Four genes from the 12 time point Drosophila embryonic development Affymetrix microarray gene expression data set by Tomancak et al. (2002).

The data has been processed using mmgmos from the puma package.

Usage

```
data(drosophila_mmgmos_fragment)
```

Format

A puma package exprReslt object with 3 repeats of the 12 time points for 4 probes.

Source

ftp://ftp.fruitfly.org/pub/embryo_tc_array_data/

References

Tomancak, P et al. Systematic determination of patterns of gene expression during Drosophila embryogenesis. *Genome Biol* 3:RESEARCH0088, 2002.

export.scores *Export results to an SQLite database*

Description

Exports the results to an SQLite database which can then be browsed with a result browser. The function will export log likelihoods, z-scores, model figures and gene aliases.

Usage

```
export.scores(scores, datasetName='', experimentSet='',
  databaseFile='database.sqlite', preprocData=NULL, models=NULL,
  figpath=NULL, aliasTypes=c("SYMBOL", "GENENAME", "ENTREZID"),
  datasetSource='', datasetDescription='',
  datasetSaveLocation='', datasetFigureFilename='',
  experimentTimestamp=as.character(Sys.Date()),
  figureDesc='', figurePrio=0, regulator=NULL)
```

Arguments

scores	The scoreList to export.
datasetName	Name of the dataset in the database.
experimentSet	Name of the experiment set in the database.
databaseFile	Filename of the database. New database is created if the file does not exist.
preprocData	Preprocessed data. This is required in order to generate models and figures and to calculate z-scores. Also, inserting aliases requires preprocessed data.
models	Learned models. If not given, the function will generate models if preprocessed data is available.
figpath	Figure path. If this is defined, the function will generate figures to the given path instead of inserting them to the database.
aliasTypes	Types of aliases that are inserted to the database.
datasetSource	Additional information that is inserted to the database if defined.
datasetDescription	Additional information that is inserted to the database if defined.
datasetSaveLocation	Additional information that is inserted to the database if defined.
datasetFigureFilename	Additional information that is inserted to the database if defined.
experimentTimestamp	Timestamp that is inserted to the database. The default value is current date in ISO-8601 format.
figureDesc	Additional information that is inserted to the database if defined.
figurePrio	Additional information that is inserted to the database if defined.
regulator	If defined, override the regulator name from scoreList.

Author(s)

Miika-Petteri Matikainen, Antti Honkela

See Also

GPRankTargets, GPRankTFs.

Examples

```
## Not run:
  # Load a mmgmos preprocessed fragment of the Drosophila developmental
  # time series
  data(drosophila_gpsim_fragment)

  # FBgn names of target genes
  targets <- c('FBgn0003486', 'FBgn0033188', 'FBgn0035257')
  # Load gene annotations
  library(annotate)
  aliasMapping <- getAnnMap("ALIAS2PROBE",
```

```
                annotation(drosophila_gpsim_fragment))

  # Get the probe identifier for TF 'twi'
  twi <- get('twi', env=aliasMapping)
  # Load alternative gene annotations
  fbgnMapping <- getAnnMap("FLYBASE2PROBE",
                 annotation(drosophila_gpsim_fragment))

  # Get the probe identifiers for target genes
  targetProbes <- mget(targets, env=fbgnMapping)

  # Rank the targets, filtering weakly expressed genes with average
  # expression z-score below 1.8
  scores <- GPRankTargets(drosophila_gpsim_fragment, TF=twi,
                          testTargets=targetProbes,
                          options=list(quiet=TRUE),
                          filterLimit=1.8)

  # Export data from scoreList and preprocessed data to a database
  export.scores(scores, datasetName='Drosophila',
                experimentSet='GPSIM/GPDISIM',
                database='database.sqlite',
                preprocData=drosophila_gpsim_fragment,
                aliasTypes=c('SYMBOL', 'GENENAME', 'FLYBASE', 'ENTREZID'))

## End(Not run)
```

`ExpressionTimeSeries-class` *Class to contain time series expression assays*

Description

Container for time series expression assays and experimental metadata. `ExpressionTimeSeries` class is derived from `ExpressionSet`, and requires fields `experiments` and `modeltime` in `phenoData`.

Extends

Directly extends class `ExpressionSet`.

Objects from the Class

`new("ExpressionTimeSeries")`

`new("ExpressionTimeSeries", phenoData = new("AnnotatedDataFrame"), featureData = new("AnnotatedDataFra experimentData = new("MIAME"), annotation = character(0), protocolData = phenoData[,integer(0)], exprs = new("matrix"), var.exprs = new("matrix"))`

This creates an `ExpressionTimeSeries` with `assayData` implicitly created to contain `exprs` and `var.exprs`.

`new("ExpressionTimeSeries", assayData = assayDataNew(exprs=new("matrix")), phenoData = new("AnnotatedDataFrame"), featureData = new("AnnotatedDataFrame"), experimentData = new("MIAME"), annotation = character(0), protocolData = phenoData[,integer(0)])`

This creates an `ExpressionTimeSeries` with `assayData` provided explicitly. In this form, the only required named argument is `assayData`.

`ExpressionTimeSeries` instances are usually created through `new("ExpressionTimeSeries", ...)`. Usually the arguments to `new` include `exprs` (a matrix of expression data, with features corresponding to rows and samples to columns), `var.exprs`, `phenoData`, `featureData`, `experimentData`, `annotation`, and `protocolData`. `phenoData`, `featureData`, `experimentData`, `annotation`, and `protocolData` can be missing, in which case they are assigned default values.

Slots

`assayData`**:** Inherited from `ExpressionSet`. The models in `gpsim` package assume that `exprs` contains absolute (i.e. non-logarithmic) expression values. The member `var.exprs` may contain variances of the values.

`phenoData`**:** Inherited from `ExpressionSet`. The following fields are required: `experiments` which contains integers from 1 to N with measurements from the same biological assay having the same number; `modeltime` which contains observation times in model units.

`featureData`**:** Inherited from `ExpressionSet`.

`experimentData`**:** Inherited from `ExpressionSet`.

`annotation`**:** Inherited from `ExpressionSet`.

`protocolData`**:** Inherited from `ExpressionSet`.

`.__classVersion__`**:** Inherited from `ExpressionSet`.

Methods

See also methods for `ExpressionSet`.

`var.exprs(object)`, `var.exprs(object)<- value` Access and set `var.exprs`

`initialize("ExpressionTimeSeries")` Object instantiation, used by new; not to be called directly by the user.

Author(s)

Antti Honkela, Jonatan Ropponen

See Also

`processData`, `processRawData`.

Examples

```
showClass("ExpressionTimeSeries")
```

expTransform *Constrains a parameter.*

Description

contains commands to constrain parameters to be positive via exponentiation or within a fixed interval via the sigmoid function.

Usage

```
expTransform(x, transform)
sigmoidTransform(x, transform)
boundedTransform(x, transform, bounds)
```

Arguments

x	input argument.
transform	type of transform, 'atox' maps a value into the transformed space (i.e. makes it positive). 'xtoa' maps the parameter back from transformed space to the original space. 'gradfact' gives the factor needed to correct gradients with respect to the transformed parameter.
bounds	a 2-vector of bounds of allowed values in boundedTransform

Value

Return value as selected by `tranform`

See Also

modelOptimise

Examples

```
# Transform unconstrained parameter -4 to a positive value
expTransform(-4, 'atox')

# Transform a bounded parameter in (1,3) to an unconstrained one
boundedTransform(2, 'xtoa', c(1, 3))
```

`generateModels` *Generating models with the given data*

Description

'generateModels' recreates models based on the parameters stored in a scoreList.

Usage

```
generateModels(preprocData, scores)
```

Arguments

`preprocData`	The preprocessed data to be used.
`scores`	A scoreList object containing data of the models to be generated.

Value

'generateModels' returns a list of the generated models.

Author(s)

Antti Honkela, Jonatan Ropponen

See Also

`GPLearn`, `GPRankTargets`, `GPRankTFs`, `scoreList`.

Examples

```
## Not run:
  # Load a mmgmos preprocessed fragment of the Drosophila developmental
  # time series
  data(drosophila_gpsim_fragment)

  # Get the target probe names
  targets <- c('FBgn0003486', 'FBgn0033188', 'FBgn0035257')
  library(annotate)
  aliasMapping <- getAnnMap("ALIAS2PROBE",
                  annotation(drosophila_gpsim_fragment))
  twi <- get('twi', env=aliasMapping)
  fbgnMapping <- getAnnMap("FLYBASE2PROBE",
                 annotation(drosophila_gpsim_fragment))
  targetProbes <- mget(targets, env=fbgnMapping)

  scores <- GPRankTargets(drosophila_gpsim_fragment, TF=twi,
                          testTargets=targetProbes,
                          options=list(quiet=TRUE),
                          filterLimit=1.8)
```

```
  models <- generateModels(drosophila_gpsim_fragment, scores)

## End(Not run)
```

GPLearn *Fit a GP model*

Description

Forms an optimized model of the desired genes. The function can form a model with GPsim or GPdisim and it's also possible to use initial parameters or fix parameters for future use. The genes can also be filtered based on ratios calculated from the expression values. The given data can also be searched for the data of specific genes.

Usage

```
GPLearn(preprocData, TF = NULL, targets = NULL,
  useGpdisim = !is.null(TF), randomize = FALSE, addPriors = FALSE,
  fixedParams = FALSE, initParams = NULL, initialZero = TRUE,
  fixComps = NULL, dontOptimise = FALSE,
  allowNegativeSensitivities = FALSE, quiet = FALSE,
  gpsimOptions = NULL, allArgs = NULL)
```

Arguments

preprocData	The preprocessed data to be used.
TF	The probe corresponding to the transcription factor (TF) mRNA if TF protein translation model is used, or NULL (default) if the translation model is not used.
targets	The target genes of the model.
useGpdisim	A logical value determining whether a model of translation is included. By default TRUE if TF is set, FALSE if TF is unset.
randomize	A logical value determining whether the parameters of the model are randomized before optimization.
addPriors	A logical value determining whether priors are added to the model.
fixedParams	A logical value determining whether the initial parameters are fixed.
initParams	The initial parameters for the model. In combination with fixedParams a value NA denotes parameters to learn.
initialZero	Assume a zero initial TF protein concentration, default = TRUE.
fixComps	The blocks of the kernel the parameters of which are to be fixed. To be used together with fixedParams and initParams.
dontOptimise	Just create the model, do not run optimisation.
allowNegativeSensitivities	Allow sensitivities to go negative. This is an experimental feature, and the negative values have no physical interpretation.

quiet	Suppress optimiser output.
gpsimOptions	Internal: additional options to pass to gp[di]simCreate.
allArgs	A list of arguments that can be used to override ones with the same name.

Value

Returns the optimized model.

Author(s)

Antti Honkela, Pei Gao, Jonatan Ropponen, Magnus Rattray, Neil D. Lawrence

See Also

GPRankTargets, GPRankTFs.

Examples

```
  # Load a mmgmos preprocessed fragment of the Drosophila developmental
  # time series
  data(drosophila_gpsim_fragment)

  # Get the target probe names
  library(annotate)
  aliasMapping <- getAnnMap("ALIAS2PROBE",
                    annotation(drosophila_gpsim_fragment))
  twi <- get('twi', env=aliasMapping)
  fbgnMapping <- getAnnMap("FLYBASE2PROBE",
                   annotation(drosophila_gpsim_fragment))
  targetProbe <- get('FBgn0035257', env=fbgnMapping)

  # Create the model but do not optimise (rarely needed...)
  model <- GPLearn(drosophila_gpsim_fragment,
                   TF=twi, targets=targetProbe,
                   useGpdisim=TRUE, quiet=TRUE,
                   dontOptimise=TRUE)
## Not run:
  # Create and learn the model
  model <- GPLearn(drosophila_gpsim_fragment,
                   TF=twi, targets=targetProbe,
                   useGpdisim=TRUE, quiet=TRUE)

## End(Not run)
```

GPModel-class *A container for gpsim models*

Description

The class is a container for the internal representation of models used by the gpsim package.

Objects from the Class

Objects can be created by calls of the form new("GPModel", model).

Slots

model: A model object used internally by the code of the gpsim package

type: Type of the model object

Methods

modelStruct(object), modelStruct(object)<- value Access and set the internal model structure

modelType(object) Access the internal type values

show(object) Informatively display object contents.

is.GPModel(object) Check if object is a GPModel.

initialize("GPModel") Object instantiation, used by new; not to be called directly by the user.

Author(s)

Antti Honkela, Jonatan Ropponen

See Also

GPLearn, GPRankTargets, GPRankTFs, generateModels, modelExtractParam, modelLogLikelihood.

Examples

```
showClass("GPModel")
```

GPPlot *Plot GP(DI)SIM models*

Description

Plots GP(DI)SIM models.

Usage

```
GPPlot(data, savepath = '', nameMapping = NULL, predt = NULL,
       fileOutput=FALSE, plotTime=NULL)
```

Arguments

data	The model to plot as returned by GPLearn.
savepath	The location in the file system where the images are saved.
nameMapping	The annotation used for mapping the names of the genes for the figures.
predt	The set of time points to use in plotting (default: the time interval covering the data).
fileOutput	Is the plot being saved to a file? If yes, do not open new interactive devices for each plot.
plotTime	The times of observations to use in the plot. Should usually not be changed!

Details

The function plots the fitted expression level of the transcription factor (if applicable), the inferred activity of the transcription factor, and the fitted expression level of the target(s).

Author(s)

Antti Honkela

See Also

GPLearn.

Examples

```
## Not run:
  # Load a mmgmos preprocessed fragment of the Drosophila developmental
  # time series
  data(drosophila_gpsim_fragment)

  # Get the target probe names
  library(annotate)
  aliasMapping <- getAnnMap("ALIAS2PROBE",
                    annotation(drosophila_gpsim_fragment))
```

```
  twi <- get('twi', env=aliasMapping)
  fbgnMapping <- getAnnMap("FLYBASE2PROBE",
                  annotation(drosophila_gpsim_fragment))
  targetProbe <- get('FBgn0035257', env=fbgnMapping)

  # Learn the model
  model <- GPLearn(drosophila_gpsim_fragment,
                  TF=twi, targets=targetProbe,
                  useGpdisim=TRUE, quiet=TRUE)

  # Plot it
  GPPlot(model, nameMapping=getAnnMap("FLYBASE",
         annotation(drosophila_gpsim_fragment)))

## End(Not run)
```

GPRankTargets *Ranking possible target genes or regulators*

Description

GPRankTargets ranks possible target genes by forming optimized models with a fixed transcription factor, a set of known target genes and targets to be tested. The transcription factor and the known targets are always included in the models while the tested targets are tested by including them in the models one at a time. The function determines itself whether to use GPSIM or GPDISIM based on the input arguments.

Usage

```
GPRankTargets(preprocData, TF = NULL, knownTargets = NULL,
              testTargets = NULL, filterLimit = 1.8,
              returnModels = FALSE, options = NULL,
              scoreSaveFile = NULL,
              datasetName = "", experimentSet = "")
GPRankTFs(preprocData, TFs, targets,
          filterLimit = 1.8, returnModels = FALSE, options = NULL,
          scoreSaveFile = NULL, datasetName = "", experimentSet = "")
```

Arguments

preprocData	The preprocessed data to be used.
TF	The transcription factor (TF) probe present in all models when TF protein translation model is used. Set to NULL (default) when translation model is not used.
knownTargets	The target genes present in all models.
testTargets	Target genes that are tested by including them in the models one at a time. Can be names of genes, or a set of indices in preprocData.

filterLimit	Genes with an average expression z-score above this figure are accepted after filtering. If this value is 0, all genes will be accepted.
returnModels	A logical value determining whether the function returns the calculated models.
options	A list of additional arguments to pass to GPLearn.
scoreSaveFile	Name of file to save the scores to after processing each gene.
TFs	The transcription factors that are tested by including them in the models one at a time.
targets	The target genes present in all models.
datasetName	For exporting the scores using export.scores: Name of the dataset in the database.
experimentSet	For exporting the scores using export.scores: Name of the experiment set in the database.

Details

The models are formed by calling GPLearn. If there is no value given to the transcription factor, a model without protein translation is used. Without protein translation model, some known targets are needed. If known targets are given, a model is first created with only the transcription factor and the known targets. The parameters extracted from this model are used as initial parameters of the models with test targets.

GPRankTFs is very similar to GPRankTargets, except it loops over candidate regulators, not candidate targets.

Value

The function returns a scoreList containing the genes, parameters and log-likelihoods of the models If returnModels is true, the function returns a list of the calculated models.

Author(s)

Antti Honkela, Jonatan Ropponen, Magnus Rattray, Neil D. Lawrence

See Also

GPLearn, scoreList, generateModels, export.scores.

Examples

```
## Not run:
  # Load a mmgmos preprocessed fragment of the Drosophila developmental
  # time series
  data(drosophila_gpsim_fragment)

  # Get the target probe names
  targets <- c('FBgn0003486', 'FBgn0033188', 'FBgn0035257')
  library(annotate)
  aliasMapping <- getAnnMap("ALIAS2PROBE",
                    annotation(drosophila_gpsim_fragment))
```

```
twi <- get('twi', env=aliasMapping)
fbgnMapping <- getAnnMap("FLYBASE2PROBE",
                annotation(drosophila_gpsim_fragment))
targetProbes <- mget(targets, env=fbgnMapping)

scores <- GPRankTargets(drosophila_gpsim_fragment, TF=twi,
                        testTargets=targetProbes,
                        options=list(quiet=TRUE),
                        filterLimit=1.8)

## End(Not run)
```

---

gpsimCreate *Create a GPSIM/GPDISIM model.*

Description

creates a model for single input motifs with Gaussian processes.

Usage

```
gpsimCreate(Ngenes, Ntf, times, y,
            yvar, options, genes=NULL, annotation=NULL)
gpdisimCreate(Ngenes, Ntf, times, y,
              yvar, options, genes=NULL, annotation=NULL)
```

Arguments

Ngenes	number of genes to be modelled in the system.
Ntf	number of proteins to be modelled in the system.
times	the time points where the data is to be modelled.
y	the values of each gene at the different time points.
yvar	the variances of each gene at the different time points.
options	options structure (optional).
genes	names of the probes the model is for
annotation	(optional) annotation for the probe names

Details

These functions are meant to be used through GPLearn.

Value

model	model structure containing default parameterisation.

See Also

modelExtractParam, modelOptimise, GPLearn.

Examples

```
## missing, see GPLearn
```

---

kernCompute *Compute the kernel given the parameters and X.*

Description

Compute the kernel given the parameters and X.

Usage

```
kernCompute(kern, x, x2)
kernDiagCompute(kern, x)
```

Arguments

kern	kernel structure to be computed.
x	first or only input data matrix (rows are data points) to the kernel computation.
x2	(optional) second input matrix to the kernel computation (forms the columns of the kernel).

Details

K <- kernCompute(kern, X) computes a kernel matrix for the given kernel type given an input data matrix.

K <- kernCompute(kern, X1, X2) computes a kernel matrix for the given kernel type given two input data matrices, one for the rows and one for the columns.

K <- kernDiagCompute(kern, X) computes the diagonal of a kernel matrix for the given kernel.

K <- *X*kernCompute(kern1, kern2, X) K <- *X*kernCompute(kern1, kern2, X1, X2) same as above, but for cross combinations of two kernels, kern1 and kern2.

Value

K	computed elements of the kernel structure.
Kd	vector containing computed diagonal elements of the kernel structure.

See Also

kernCreate

Examples

```
kern <- kernCreate(1, 'rbf')
K <- kernCompute(kern, as.matrix(3:8))
```

`kernCreate` *Initialise a kernel structure.*

Description

Initialise a kernel structure.

Usage

```
kernCreate(x, kernType, kernOptions=NULL)
```

Arguments

`x`	If list, array or matrix: input data values (from which kernel will later be computed). If scalar: input dimension of the design matrix (i.e. number of features in the design matrix).
`kernType`	Type of kernel to be created, some standard types are 'rbf', 'white', 'sim' and 'disim'. If a list of the form `list(type='cmpnd', comp=c('rbf', 'rbf', 'white'))` is used a compound kernel based on the sum of the individual kernels will be created. Parameters can be passed to kernels using type `list(type='parametric', options=list(opt=val), realType=...)`, where `realType` is the type that would be used otherwise.
`kernOptions`	(optional) list of kernel options

Details

`kern <- kernCreate(X, type)` input points and a kernel type.

`kern <- kernCreate(dim, type)` creates a kernel matrix structure given the dimensions of the design matrix and the kernel type.

The *KernParamInit functions perform initialisation specific to different types of kernels. They should not be called directly.

Value

`kern`	The kernel structure.

See Also

`kernDisplay`, `modelTieParam`.

Examples

```
# Create a multi kernel with two rbf blocks with bounded inverse widths
invWidthBounds <- c(0.5, 2)
kernType <- list(type="multi", comp=list())
for (i in 1:2)
  kernType$comp[[i]] <- list(type="parametric", realType="rbf",
                             options=list(isNormalised=TRUE,
                               inverseWidthBounds=invWidthBounds))
kern <- kernCreate(1, kernType)

# Tie the inverse with parameters of the component RBF kernels
kern <- modelTieParam(kern, list(tieWidth="inverseWidth"))
kernDisplay(kern)
```

---

kernDiagGradX *Compute the gradient of the kernel wrt X.*

Description

computes the gradient of the (diagonal of the) kernel matrix with respect to the elements of the design matrix given in X.

Usage

```
kernDiagGradX(kern, x)
kernGradX(kern, x, x2)
```

Arguments

kern	the kernel structure for which gradients are being computed.
x	if only argument: the input data in the form of a design matrix, if two arguments: row locations against which gradients are being computed.
x2	(optional) column locations against which gradients are being computed.

Value

gX	the gradients of the diagonal with respect to each element of X. The returned matrix has the same dimensions as X.
gX2	the returned gradients. The gradients are returned in a matrix which is numData x numInputs x numData. Where numData is the number of data points and numInputs is the number of input dimensions in X.

See Also

kernGradient

Examples

```
kern <- kernCreate(1, 'mlp')
g <- kernDiagGradX(kern, as.matrix(3:8))
```

`kernGradient` *Compute the gradient wrt the kernel parameters.*

Description

Compute the gradient wrt the kernel parameters.

Usage

```
kernGradient(kern, x, ...)
```

Arguments

`kern`	the kernel structure for which the gradients are being computed.
`x`	the input locations for which the gradients are being computed, specifically those associated with the rows of the kernel matrix if there are two arguments of input locations.
`...`	optional arguments including potentially: the input locations associated with the columns of the kernel matrix; matrix of partial derivatives of the function of interest with respect to the kernel matrix. With single input, the argument takes the form of a square matrix of dimension numData, where numData is the number of rows in x, with two input arguments the matrix should have the same number of rows as the first and the same number of columns as the second has rows.

Details

`g <- kernGradient(kern, x, partial)` `g <- *kernGradient(kern, x, partial)` computes the gradient of functions with respect to the kernel parameters. As well as the kernel structure and the input positions, the user provides a matrix PARTIAL which gives the partial derivatives of the function with respect to the relevant elements of the kernel matrix.

`g <- kernGradient(kern, x1, x2, partial_)` `g <- *kernGradient(kern, x1, x2, partial_)` computes the derivatives as above, but input locations are now provided in two matrices associated with rows and columns of the kernel matrix.

`g <- *X*kernGradient(kern1, kern2, x, partial)` `g <- *X*kernGradient(kern1, kern2, x1, x2, partial_)` same as above, but for cross combinations of two kernels, `kern1` and `kern2`.

Value

`g`	gradients of the function of interest with respect to the kernel parameters. The ordering of the vector should match that provided by the function kernExtractParam.

See Also

kernCompute, kernExtractParam.

Examples

```
kern <- kernCreate(1, 'rbf')
g <- kernGradient(kern, as.matrix(c(1, 4)), array(1, c(2, 2)))
```

---

lnDiffErfs *Helper function for computing the log of difference*

Description

Helper function for computing the log of difference

Usage

```
lnDiffErfs(x1, x2)
```

Arguments

x1	argument of the positive erf
x2	argument of the negative erf

Details

v <- lnDiffErfs(x1, x2) computes the log of the difference of two erfs in a numerically stable manner.

Value

v	list(c(log(abs(erf(x1) - erf(x2))), sign(erf(x1) - erf(x2))))

Examples

```
lnDiffErfs(100, 10)
```

modelDisplay *Display a model.*

Description

displays the parameters of the model/kernel and the model/kernel type to the console.

Usage

```
modelDisplay(model, ...)
```

Arguments

model	the model/kernel structure to be displayed.
...	(optional) indent level for the display.

See Also

modelExtractParam

Examples

```
# Load a mmgmos preprocessed fragment of the Drosophila developmental
# time series
data(drosophila_gpsim_fragment)

# The probe identifier for TF 'twi'
twi <- "143396_at"
# The probe identifier for the target gene
targetProbe <- "152715_at"

# Create the model, but do not optimise
model <- GPLearn(drosophila_gpsim_fragment,
                 TF=twi, targets=targetProbe,
                 useGpdisim=TRUE, quiet=TRUE,
                 dontOptimise=TRUE)

# Display the initial model
modelDisplay(model)
```

`modelExpandParam` *Update a model structure with new parameters or update the posterior processes.*

Description

Update a model structure or component with new parameters, or update the posterior processes.

Usage

```
modelExpandParam(model, params)
modelUpdateProcesses(model, predt=NULL)
```

Arguments

`model`	the model structure to be updated.
`params`	vector of parameters.
`predt`	(optional) a vector of times to infer the posterior at. By default this is 100 points spanning the time range of the data.

Details

`model <- modelExpandParam(model, param)` returns a model structure filled with the parameters in the given vector. This is used as a helper function to enable parameters to be optimised in, for example, the optimisation functions.

`model <- modelUpdateProcesses(model)` updates posterior processes of the given model.

Value

`model`	updated model structure.

See Also

`GPLearn`, `modelExtractParam`

Examples

```
## Not run:
  # Learn the model
  model <- GPLearn(...)
  params <- modelExtractParam(model, only.values=TRUE)
  params[1] <- 0
  new_model <- modelExpandParam(model, params)
  new_model <- modelUpdateProcesses(new_model)

## End(Not run)
```

modelExtractParam *Extract the parameters of a model.*

Description

Extract parameters from the model into a vector of parameters for optimisation.

Usage

```
modelExtractParam(model, only.values=TRUE, untransformed.values=FALSE)
```

Arguments

model	the model structure containing the parameters to be extracted.
only.values	include parameter names in the returned vector.
untransformed.values	return actual values, not transformed values used by the optimisers.

Value

param	vector of parameters extracted from the model.

See Also

modelExpandParam

Examples

```
# Load a mmgmos preprocessed fragment of the Drosophila developmental
# time series
data(drosophila_gpsim_fragment)

# The probe identifier for TF 'twi'
twi <- "143396_at"
# The probe identifier for the target gene
targetProbe <- "152715_at"

# Create the model, but do not optimise
model <- GPLearn(drosophila_gpsim_fragment,
                 TF=twi, targets=targetProbe,
                 useGpdisim=TRUE, quiet=TRUE,
                 dontOptimise=TRUE)

# Get the initial parameter values
params <- modelExtractParam(model, only.values=FALSE)
```

modelGradient *Model log-likelihood/objective error function and its gradient.*

Description

modeGradient gives the gradient of the objective function for a model. By default the objective function (modelObjective) is a negative log likelihood (modelLogLikelihood).

Usage

```
modelObjective(params, model, ...)
modelLogLikelihood(model)
modelGradient(params, model, ...)
```

Arguments

params	parameter vector to evaluate at.
model	model structure.
...	optional additional arguments.

Value

g	the gradient of the error function to be minimised.
v	the objective function value (lower is better).
ll	the log-likelihood value.

See Also

modelOptimise.

Examples

```
# Load a mmgmos preprocessed fragment of the Drosophila developmental
# time series
data(drosophila_gpsim_fragment)

# The probe identifier for TF 'twi'
twi <- "143396_at"
# The probe identifier for the target gene
targetProbe <- "152715_at"

# Create the model but do not optimise
model <- GPLearn(drosophila_gpsim_fragment,
                 TF=twi, targets=targetProbe,
                 useGpdisim=TRUE, quiet=TRUE,
                 dontOptimise=TRUE)
params <- modelExtractParam(model, only.values=FALSE)
ll <- modelLogLikelihood(model)
```

```
paramValues <- modelExtractParam(model, only.values=TRUE)
modelGradient(paramValues, model)
```

`modelTieParam` *Tie parameters of a model together.*

Description

groups of parameters of a model to be seen as one parameter during optimisation of the model.

Usage

```
modelTieParam(model, paramsList)
```

Arguments

`model`	the model for which parameters are being tied together.
`paramsList`	indices of parameteres to group together. The indices are provided in a list. Each element in the list contains a vector of indices of parameters that should be considered as one parameter. Each group of parameters in each cell should obviously be mutually exclusive. Alternatively, the specification may consist of strings, which are interpreted as regular expressions that are matched against the parameter names returned by `modelExtractParam` or `kernExtractParam`, as appropriate fot the current object.

Value

`model`	the model with the parameters grouped together.

See Also

`modelExtractParam`, `modelExpandParam`, `modelGradient`.

Examples

```
# Create a multi kernel with two rbf blocks with bounded inverse widths
invWidthBounds <- c(0.5, 2)
kernType <- list(type="multi", comp=list())
for (i in 1:2)
  kernType$comp[[i]] <- list(type="parametric", realType="rbf",
                             options=list(isNormalised=TRUE,
                               inverseWidthBounds=invWidthBounds))
kern <- kernCreate(1, kernType)

# Tie the inverse with parameters of the component RBF kernels
kern <- modelTieParam(kern, list(tieWidth="inverseWidth"))
kernDisplay(kern)
```

optimiDefaultConstraint *Returns function for parameter constraint.*

Description

returns the current default function for constraining a parameter.

Usage

```
optimiDefaultConstraint(constraint)
```

Arguments

constraint	the type of constraint you want to place on the parameter, options include 'positive' (gives an 'exp' constraint) and 'zeroone' (gives a 'sigmoid' constraint).

Value

val	a list with two components: 'func' for the name of function used to apply the constraint, and 'hasArgs' for a boolean flag if the function requires additional arguments.

See Also

expTransform, sigmoidTransform.

Examples

```
optimiDefaultConstraint('positive')
optimiDefaultConstraint('bounded')
```

plotTimeseries *Plot ExpressionTimeSeries data*

Description

Plots ExpressionTimeSeries data.

Usage

```
plotTimeseries(data, nameMapping = NULL)
```

Arguments

data	An ExpressionTimeSeries object.
nameMapping	The annotation used for mapping the names of the genes for the figures. By default, the SYMBOL annotation for the array is used, if available.

Details

The function plots the expression levels from an ExpressionTimeSeries object and the associated standard deviations. If the object includes multiple time series, they will be plotted in the same figure, but slightly shifted.

Author(s)

Antti Honkela

See Also

processData.

Examples

```
# Load a mmgmos preprocessed fragment of the Drosophila developmental
# time series
data(drosophila_gpsim_fragment)

# Plot the first two genes
plotTimeseries(drosophila_gpsim_fragment[1:2,])
```

---

processData *Processing expression time series*

Description

processData further processes time series data preprocessed by puma or lumi.

processRawData performs similar processing for other data.

Both functions return ExpressionTimeSeries objects that can be used as input for the functions GPLearn and GPRankTargets.

Usage

```
processData(data, times = NULL, experiments = NULL,
  do.normalisation = TRUE)
processRawData(rawData, times, experiments = NULL,
  is.logged = TRUE, do.normalisation = ifelse(is.logged, TRUE, FALSE))
```

Arguments

data	The preprocessed data from mmgMOS in the puma package (an exprReslt object) or from the lumi package (a LumiBatch object).
rawData	Raw data matrix to be used. Each row corresponds to a gene and each column to a data point.
times	Observation times of each data point. If unspecified or NULL, processData attempts to infer this from phenoData(data) field containing 'time' in the name.
experiments	The replicate structure of the data indicating which expression data points arise from which experiments. This should be an array in integers from 1 to N with length equal to the number of data points. By default all the data points are assumed to be from same replicate.
is.logged	Indicates whether the expression values are on log scale or not. Normalisation of non-logged data is unsupported.
do.normalisation	Indicates whether to perform the normalisation.

Details

The expression data (and percentiles, if available) are normalized by equalising the mean of log-expression in each time points. In processData, a normal distribution is then fitted into the data with distfit.

Value

An ExpressionTimeSeries object containing all provided information.

Author(s)

Antti Honkela, Jonatan Ropponen

See Also

GPLearn, GPRankTargets.

Examples

```
## Load a mmgmos preprocessed fragment of the Drosophila developmental
## time series
data(drosophila_mmgmos_fragment)

## Process the data (3 experiments containing 12 time points each)
drosophila_gpsim_fragment <- processData(drosophila_mmgmos_fragment,
  experiments=rep(1:3, each=12))
```

SCGoptim *Optimise the given function using (scaled) conjugate gradients.*

Description

Optimise the given function using (scaled) conjugate gradients.

Usage

```
optimiDefaultOptions()
SCGoptim(x, fn, grad, options, ...)
CGoptim(x, fn, grad, options, ...)
modelOptimise(model, options, ...)
```

Arguments

model	the model to be optimised.
x	initial parameter values.
fn	objective function to minimise
grad	gradient function of the objective
options	options structure like one returned by optimiDefaultOptions. The fields are interpreted as\ option[1] : number of iterations\ option[2] : interval for the line search\ option[3] : tolerence for x to terminate the loop\ option[4] : tolerence for fn to terminate the loop\ option$display : option of showing the details of optimisaton
...	extra arguments to pass to fn and grad

Value

options	an options structure
newParams	optimised parameter values
model	the optimised model.

See Also

modelObjective, modelGradient

Examples

```
## Not run to speed up package checks
# model <- GPLearn(..., dontOptimise=TRUE)
# options <- optimiDefaultOptions()
# model <- modelOptimise(model, options)
```

scoreList-class *Class "scoreList"*

Description

'scoreList' is an object which contain the genes, parameters, log-likelihoods and arguments of models. With the data in a scoreList item and the original data used for creating the models, the models can be reconstructed with the function 'generateModels'.

Objects from the Class

Objects can be created by calls of the form `scoreList(params, loglikelihoods, genes, modelArgs, knownTargets, TF, sharedModel)`.

Slots

`params`: The parameters of the models.

`loglikelihoods`: The log-likelihoods of the models.

`baseloglikelihoods`: The log-likelihoods of corresponding null models.

`genes`: The genes used in the models.

`modelArgs`: A list of arguments used to generate the models.

`knownTargets`: The list of known targets used in the ranking.

`TF`: The TF used in the ranking.

`sharedModel`: Shared model for known targets.

`datasetName`: Dataset name, used when exporting scores to a database.

`experimentSet`: Experiment set name, used when exporting scores to a database.

Methods

Class-specific methods:

`write.scores(object, ...)` Writes the log-likelihoods and null log-likelihoods. Accepts any options `write.table` does.

`genes(object)`, `genes(object)<- value` Access and set `genes`

`knownTargets(object)`, `knownTargets(object)<- value` Access and set `knownTargets`

`loglikelihoods(object)`, `loglikelihoods(object)<- value` Access and set `loglikelihoods`

`baseloglikelihoods(object)`, `baseloglikelihoods(object)<- value` Access and set `baseloglikelihoods`

`modelArgs(object)`, `modelArgs(object)<- value` Access and set `modelArgs`

`params(object)`, `params(object)<- value` Access and set `params`

`sharedModel(object)`, `sharedModel(object)<- value` Access and set `sharedModel`

`TF(object)`, `TF(object)<- value` Access and set `TF`

`datasetName(object)`, `datasetName(object)<- value` Access and set `datasetName`

`experimentSet(object)`, `experimentSet(object)<- value` Access and set `experimentSet`

Standard generic methods:

`object[(index)` Conducts subsetting of the scoreList.

`c(object, ...)` Concatenates scoreLists.

`length(object)` Returns the length of the list.

`show(object)` Informatively display object contents.

`sort(object, decreasing=FALSE)` Sort the list according to log-likelihood

Author(s)

Antti Honkela, Jonatan Ropponen

See Also

`GPRankTargets`, `GPRankTFs`, `generateModels`, `write.table`.

Examples

```
showClass("scoreList")
```

Index