Package 'SynExtend'

June 5, 2026

Type Package

Title Tools for Comparative Genomics

Version 1.25.1

biocViews Genetics, Clustering, ComparativeGenomics, DataImport

Description A multitude of tools for comparative genomics, focused on large-scale analyses of biological data. SynExtend includes tools for working with syntenic data, clustering massive network structures, and estimating functional relationships among genes.

Depends R (>= 4.5.0), DECIPHER (>= 2.28.0)

Imports methods, Biostrings, S4Vectors, IRanges, utils, stats, parallel, graphics, grDevices, RSQLite, DBI

Suggests BiocStyle, knitr, igraph, markdown, rmarkdown, rtracklayer

License GPL-3

ByteCompile true

Encoding UTF-8

NeedsCompilation yes

VignetteBuilder knitr

URL https://github.com/npcooley/SynExtend

BugReports https://github.com/npcooley/SynExtend/issues/new/

LazyLoad true

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

git_branch devel

git_last_commit ff73121

git_last_commit_date 2026-05-25

Repository Bioconductor 3.24

Date/Publication 2026-06-04

Author Nicholas Cooley [aut, cre] (ORCID: <https://orcid.org/0000-0002-6029-304X>), Aidan Lakshman [aut, ctb] (ORCID: <https://orcid.org/0000-0002-9465-6785>), Adelle Fernando [ctb], Erik Wright [aut]

Maintainer Nicholas Cooley <npc19@pitt.edu>

Contents

`AAHitScoping` *Adjust the scope of kmer hits between feature and genome space.*

Description

This function is designed to work internally to functions within `SynExtend` so it works on relatively simple atomic vectors and has little overhead checking.

Usage

```
AAHitScoping(hitlist,
             fstrand1,
             fstart1,
             fstop1,
             fstrand2,
             fstart2,
             fstop2)
```

Arguments

`hitlist`	A list containing matrices produced by `SearchIndex`.
`fstrand1`	An integer vector of 0s and 1s describing the strand of features.
`fstart1`	Integer; a vector of left bounds of features.
`fstop1`	Integer; a vector of right bounds of features.
`fstrand2`	An integer vector of 0s and 1s describing the strand of features.
`fstart2`	Integer; a vector of left bounds of features.
`fstop2`	Integer; a vector of right bounds of features.

Details

`AAHitScoping` converts the hits returned by `SearchIndex` from feature-to-feature context genome-to-genome context.

Value

A list of matrices.

Author(s)

Nicholas Cooley <npc19@pitt.edu>

See Also

NucleotideOverlap, SummarizePairs

Examples

```
# a contrived example
x <- list(matrix(c(1L, 3L, 1L, 3L,
                   5L, 7L, 5L, 7L),
                 nrow = 4L,
                 ncol = 2L),
          matrix(c(2L, 4L, 2L, 4L),
                 nrow = 4L,
                 ncol = 1L))

# Feature 1 (query): forward strand, genome positions 100-199
# Feature 2 (query): reverse strand, genome positions 300-449
vals01 <- c(0L, 1L)   # 0 = forward, 1 = reverse
vals02 <- c(100L, 300L)
vals03 <- c(199L, 449L)

## Subject features mirror the same layout
vals04 <- c(0L, 1L)
vals05 <- c(500L, 700L)
vals06 <- c(599L, 849L)

result <- AAHitScoping(hitlist  = x,
                       fstrand1 = vals01,
                       fstart1 = vals02,
                       fstop1 = vals03,
                       fstrand2 = vals04,
                       fstart2 = vals05,
                       fstop2 = vals06)
```

`ApproximateBackground` *Return the approximate background alignment score for a series of paired sequences.*

Description

This function is designed to work internally to SummarizePairs so it works on relatively simple atomic vectors and has little overhead checking.

Usage

```
ApproximateBackground(p1,
                      p2,
                      code1,
                      code2,
                      mod1,
                      mod2,
                      aa1,
                      aa2,
                      nt1,
```

```
                    nt2,
                    register1,
                    register2,
                    aamat,
                    ntmat)
```

Arguments

p1	Integer; references positions within nt1 or aa1.
p2	Integer; references positions within nt2 or aa2.
code1	Logical; specifies whether the position referenced by p1 is reported as a coding sequence.
code2	Logical; specifies whether the position referenced by p2 is reported as a coding sequence.
mod1	Logical; specifies whether the position referenced by p1 can be translated without complaint by translate.
mod2	Logical; specifies whether the position referenced by p2 can be translated without complaint by translate.
aa1	AAStringSet.
aa2	AAStringSet.
nt1	DNAStringSet.
nt2	DNAStringSet.
register1	Integer; a vector that maps which positions in aa1 are the translations of that particular index in nt1. NAs identify positions that are not translated.
register2	Integer; a vector that maps which positions in aa2 are the translations of that particular index in nt2. NAs identify positions that are not translated.
aamat	A substitution matrix for amino acids.
ntmat	A substitution matrix for nucleotides.

Details

ApproximateBackground generates approximate background alignment scores for sets of sequences.

Value

A vector of numerics.

Author(s)

Nicholas Cooley <npc19@pitt.edu>

See Also

NucleotideOverlap, SummarizePairs, FindSynteny

Examples

```
fas <- system.file("extdata", "50S_ribosomal_protein_L2.fas", package="DECIPHER")
dna <- readDNAStringSet(fas)
aa <- translate(dna)

s1 <- sample(x = length(dna),
             size = 30,
             replace = FALSE)
s2 <- s1[1:15]
s1 <- s1[16:30]

mat1 <- DECIPHER:::.getSubMatrix("PFASUM50")
mat2 <- DECIPHER:::.nucleotideSubstitutionMatrix(2L, -1L, 1L)

aa1 <- aa2 <- alphabetFrequency(aa)
aa1 <- aa2 <- aa1[, colnames(mat1)]
aa1 <- aa2 <- aa1 / rowSums(aa1)

nt1 <- nt2 <- alphabetFrequency(dna)
nt1 <- nt2 <- nt1[, colnames(mat2)]
nt1 <- nt2 <- nt1 / rowSums(nt1)

x <- ApproximateBackground(p1 = s1,
                           p2 = s2,
                           code1 = rep(TRUE, length(s1)),
                           code2 = rep(TRUE, length(s2)),
                           mod1 = rep(TRUE, length(s1)),
                           mod2 = rep(TRUE, length(s2)),
                           aa1 = aa1,
                           aa2 = aa2,
                           nt1 = nt1,
                           nt2 = nt2,
                           register1 = seq(length(dna)),
                           register2 = seq(length(dna)),
                           aamat = mat1,
                           ntmat = mat2)
```

`BlastSeqs` *Run BLAST queries from R*

Description

Wrapper to run BLAST queries using the commandline BLAST tool directly from R. Can operate on an `XStringSet` or a `FASTA` file.

This function requires the BLAST+ commandline tools, which can be downloaded from https://blast.ncbi.nlm.nih.gov/Bla

Usage

```
BlastSeqs(seqs, BlastDB,
              blastType=c('blastn', 'blastp', 'tblastn', 'blastx', 'tblastx'),
              extraArgs='', verbose=TRUE)
```

Arguments

`seqs`	Sequence(s) to run BLAST query on. This can be either an `XStringSet` or a path to a FASTA file.
`BlastDB`	Character; path to FASTA file in a pre-built BLAST Database. These can be built using either `MakeBlastDb` from R or the commandline `makeblastdb` function from BLAST+. For more information on building BLAST DBs, see here.
`blastType`	Character; type of BLAST query to run. See 'Details' for more information on available types.
`extraArgs`	Character; additional arguments to be passed to the BLAST query executed on the command line. This should be a single string. (Optional)
`verbose`	Logical; should output be displayed? (Optional, default `TRUE`)

Details

BLAST implements multiple types of search. Available types are the following:

- `blastn`: Nucleotide sequences against database of nucleotide sequences
- `blastp`: Protein sequences against database of protein sequences
- `tblastn`: Protein sequences against translated database of nucleotide sequences
- `blastx`: Translated nucleotide sequences against database of protein sequences
- `tblastx`: Translated nucleotide sequences against translated database of nucleotide sequences

Different BLAST queries require different inputs. The function will throw an error if the input data does not match expected input for the requested query type.

Input sequences for `blastn`, `blastx`, and `tblastx` should be nucleotide data.

Input sequences for `blastp` and `tblastn` should be amino acid data.

Database for `blastn`, `tblastn`, `tblastx` should be nucleotide data.

Database for `blastp` and `blastx` should be amino acid data.

Value

Returns a data frame (data.frame) of results of the BLAST query.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

See Also

`MakeBlastDb`

Examples

```
#
```

BlockByRank *Return simple summaries of blocks of candidate pairs.*

Description

This function is designed to work internally to SummarizePairs so it works on relatively simple atomic vectors and has little overhead checking. All arguments must be the same length.

Usage

```
BlockByRank(index1,
            partner1,
            index2,
            partner2)
```

Arguments

index1	Integer; references the contigs containing candidate feature partners.
partner1	Integer; references the candidate feature partners by row position in the source DataFrame.
index2	Integer; references the contigs containing candidate feature partners.
partner2	Integer; references the candidate feature partners by row position in the source DataFrame.

Details

BlockByRank uses the diagonal rank to identify where runs of candidate features are present in sequential blocks. In cases where a candidate feature is part of two competing blocks it is assigned to the larger.

Value

A list with named elements absblocksize and blockidmap.

Author(s)

Nicholas Cooley <npc19@pitt.edu>

See Also

NucleotideOverlap, SummarizePairs, FindSynteny

Examples

```
data("init_pairs", package = "SynExtend")
x <- paste(init_pairs$p1, init_pairs$p2, sep = "_")
x <- do.call(rbind, strsplit(x = x, split = "_", fixed = TRUE))
x <- matrix(data = as.integer(x), nrow = nrow(x))
y <- BlockByRank(index1 = x[, 2],
                 partner1 = x[, 3],
                 index2 = x[, 5],
                 partner2 = x[, 6])
```

BuiltInEnsembles *Pretrained EvoWeaver Ensemble Models*

Description

EvoWeaver has best performance with an ensemble method combining individual evidence streams. This data file provides pretrained models for ease of use. Two groups of models are provided: 1. Models trained on the KEGG MODULES dataset 2. Models trained on the CORUM dataset

These models are used internally if the user does not provide their own model, and aren't explicitly designed to be accessed by the user.

See the examples for how to train your own ensemble model.

Usage

```
data("BuiltInEnsembles")
```

Format

The data contain a named list of objects of class `glm`. This list currently has two entries: `"KEGG"` and `"CORUM"`

Examples

```
## Training own ensemble method to avoid using built-ins
## defaults to built-ins when an ensemble isn't provided
set.seed(333L)
exData <- get(data("ExampleStreptomycesData"))
ew <- EvoWeaver(exData$Genes[seq_len(8L)], MySpeciesTree=exData$Tree, NoWarn=TRUE)
datavals <- predict(ew, NoPrediction=TRUE, Verbose=interactive())
datavals <- datavals[datavals[,1] != datavals[,2],]

# Picking random numbers for demonstration purposes
actual_values <- sample(rep(c(1,0), length.out=nrow(datavals)))
datavals[,'y'] <- actual_values
myModel <- glm(y~., datavals[,-c(1,2)], family='binomial')

predictionPW <- EvoWeaver(exData$Genes[9:10], MySpeciesTree=exData$Tree, NoWarn=TRUE)
predict(predictionPW,
          PretrainedModel=myModel, Verbose=interactive())[2,,drop=FALSE]
```

CheckAgainstReport *Pull an assembly from the NCBI FTP site.*

Description

This function is designed to work internally to functions within `SynExtend` so it works on relatively simple atomic vectors and has little overhead checking.

Usage

```
CheckAgainstReport(FTP_ADDRESS,
                   CHECK_ADDRESS,
                   RETRY = 5L)
```

Arguments

FTP_ADDRESS	Character; the ftp address of an ncbi assembly.
CHECK_ADDRESS	Character; the ftp address of an ncbi assembly report.
RETRY	Integer; the number of times to retry an assembly download should it not pull correctly.

Details

On occasion, readDNAStringSet fails to completely pull assemblies from the ncbi ftp site. It is not clear why, though it is infrequent but replicable at large scale. CheckAgainstReport checks the captured DNAStringSet against the reported assembly size and string widths.

Value

A DNAStringSet.

Author(s)

Nicholas Cooley <npc19@pitt.edu>

See Also

readDNAStringSet

Examples

```
#
```

CIDist_NullDist *Simulated Null Distributions for CI Distance*

Description

Simulated values of Clustering Information Distance for random trees with 4 to 200 shared leaves.

Usage

```
data("CIDist_NullDist")
```

Format

A matrix CI_DISTANCE_INTERNAL with 197 columns and 13 rows.

Details

Each column of the matrix corresponds to the distribution of distances between random trees with the given number of leaves. This begins at `CI_DISTANCE_INTERNAL[,1]` corresponding to 4 leaves, and ends at `CI_DISTANCE_INTERNAL[,197]` corresponding to 200 leaves. Distances begin at 4 leaves since there is only one unrooted tree with 1, 2, or 3 leaves (so the distance between any given tree with less than 4 leaves is always 0).

Each row of the matrix corresponds to statistics for the given simulation set. The first row gives the minimum value, the next 9 give quantiles in `c(1%, 5%, 10%, 25%, 50%, 75%, 90%, 95%, 99%)`, and the last three rows give the `max`, `mean`, and `sd` (respectively).

Source

Datafiles obtained from the TreeDistData package, published as part of Smith (2020).

References

Smith, Martin R. *Information theoretic generalized Robinson–Foulds metrics for comparing phylogenetic trees.* Bioinformatics, 2020. **36**(20):5007-5013.

Examples

```
data(CIDist_NullDist)
```

`CreateDecoys` *Generating Decoy Alignments for Feature Pairs*

Description

A function for generating a set of decoy pairwise alignments from a database of genome sequences and an accompanying gene calls object. For each pair of genomes, one feature set is reversed by default prior to alignment, producing spurious matches that serve as a null distribution against which real `PairSummaries` results can be calibrated or filtered.

Usage

```
CreateDecoys(DataBase01,
             GeneCalls,
             K_val_01,
             K_val_02,
             K_val_03,
             DefaultTranslationTable = "11",
             NT_limit = NULL,
             AA_limit = NULL,
             TRY_limit = NULL,
             DecoyMethod = c("reverse", "none"),
             Verbose = FALSE)
```

Arguments

DataBase01	Either a connection object to a DECIPHER-compatible SQLite database, or a character string giving the path to such a database on disk. The database must contain a Seqs table with nucleotide sequences whose identifiers correspond to the names of GeneCalls. If amino acid sequences for a given identifier are absent will be translated on the fly and written back to the database. If a path string is supplied, the RSQLite package must be installed.
GeneCalls	A named list of gene call objects, where each name is an identifier matching an entry in the Seqs table of DataBase01. Must contain at least two entries. Each element is expected to have the fields Range, Coding, Strand, and Translation_Table, as produced by SquaregffBy or FindGenes.
K_val_01	A positive integer specifying the k-mer width used to compute nucleotide frequency profiles (oligonucleotideFrequency) for k-mer distance calculations. This value is also stored as the KmerSize attribute of the returned object.
K_val_02	A positive integer specifying the k-mer width passed to IndexSeqs when building the inverted index over amino acid feature sequences. Controls the sensitivity of the amino acid search step.
K_val_03	A positive integer specifying the k-mer width passed to IndexSeqs when building the inverted index over nucleotide feature sequences. Controls the sensitivity of the nucleotide search step.
DefaultTranslationTable	A character string of length 1 specifying the NCBI genetic code identifier to use when translating coding sequences whose translation table is not recorded in GeneCalls. Defaults to "11" (the bacterial, archaeal, and plant plastid code). Must be a valid identifier accepted by Biostrings::getGeneticCode.
NT_limit	Either NULL (the default) or a positive integer specifying the cumulative number of nucleotide-aligned decoy pairs at which iteration stops early. When NULL, no nucleotide-count limit is applied. When used concurrently with AA_limit the first limit reached triggers the early exit, leaving the second un-respected.
AA_limit	Either NULL (the default) or a positive integer specifying the cumulative number of amino-acid-aligned decoy pairs at which iteration stops early. When NULL, no amino acid count limit is applied. When used concurrently with NT_limit the first limit reached triggers the early exit, leaving the second un-respected.
TRY_limit	Either NULL (the default) or a numeric or integer of length 1 specifying the maximum number of genome-pair comparisons to attempt. When NULL, all pairs in the upper triangle of GeneCalls are attempted. Useful for generating a fixed-size decoy set from a large multi-genome collection without iterating over every possible pair.
DecoyMethod	A character string controlling how the decoy is constructed for each genome pair. Must be one of "reverse" (the default; the feature sequences of a randomly chosen genome in each pair are reversed before alignment, producing alignments expected to be spurious) or "none" (no transformation is applied, yielding alignments between unmodified sequences). Partial matching is supported.
Verbose	Logical indicating whether to print a status message, display a progress bar, and print the elapsed time upon completion. Defaults to FALSE.

Details

For each ordered pair of genomes (i, j) in the upper triangle of GeneCalls (or up to TRY_limit pairs), the function:

1. Retrieves nucleotide feature sequences from DataBase01 via FeaturesFromDF. Amino acid sequences are retrieved from the AAs table if present, or translated via GetTranslatedFeatures and written back to the database for reuse.
2. Randomly assigns which genome's sequences are treated as the "subject" and which as the "pattern". Depending on DecoyMethod, the subject sequences are reversed (character-level reversal of the XStringSet) before search and alignment, ensuring that any matches found are unlikely to reflect true homology.
3. Builds inverted k-mer indices over the subject nucleotide and amino acid sequences using IndexSeqs with K_val_03 and K_val_02 respectively, then searches the pattern sequences against those indices using SearchIndex.
4. For each set of search hits, computes k-mer distances (normalised Euclidean distance between K_val_01-mer frequency profiles), hit statistics (TotalMatch, MaxMatch, UniqueMatches), pairwise alignments via AlignPairs, local and approximate global PID and alignment score, a positional consensus score, and a background-corrected score using per-genome residue composition and the PFASUM50 (amino acid) or a $2/-1/1$ match/mismatch/gap (nucleotide) substitution matrix.
5. Nucleotide-level results for any pair also represented in the amino acid results are discarded to avoid redundancy.
6. Iteration halts early if AA_limit, NT_limit, or TRY_limit is reached, whichever comes first.

The returned object has the same column schema as PairSummaries objects produced by SummarizePairs, making the two directly comparable for score distribution analysis and threshold calibration. All Block_UID values are assigned as sequential integers; no syntenic block structure is inferred because decoy pairs have no context from the perspective of the search scheme.

Note that this function does not implement the memory-pool management present in SummarizePairs and NucleotideOverlap, and may therefore consume more memory when applied to large genome collections.

Value

A data.frame of class c("data.frame", "PairSummaries") with one row per decoy feature pair. Column definitions match those returned by SummarizePairs:

p1 Character. Name of the subject feature (potentially reversed, depending on DecoyMethod).

p2 Character. Name of the pattern feature.

Consensus Numeric. Mean positional consensus score across k-mer hits for the pair, re-scaled so that values near 1 indicate hits at similar relative positions in both sequences and values near 0 indicate maximal positional disagreement.

p1featurelength Integer. Length of the subject feature in nucleotides.

p2featurelength Integer. Length of the pattern feature in nucleotides.

blocksize Integer. Always 1 for decoy pairs; no syntenic block context is available.

KDist Numeric. Normalised Euclidean distance between the K_val_01-mer frequency profiles of the two features.

TotalMatch Integer. Total number of nucleotide positions covered by k-mer hits for the pair.

MaxMatch Integer. Length of the largest single k-mer hit for the pair, in nucleotides.

UniqueMatches Integer. Number of distinct k-mer hits for the pair.

Local_PID Numeric. Fraction of matched positions within the aligned region (local percent identity).

Local_Score Numeric. Alignment score normalised by alignment length.

Approx_Global_PID Numeric. Fraction of matched positions normalised by the length of the longer feature (approximate global percent identity).

Approx_Global_Score Numeric. Alignment score normalised by the length of the longer feature.

Alignment Character. Alphabet used for alignment: `"AA"` for amino acid or `"NT"` for nucleotide.

Block_UID Integer. A sequential unique identifier assigned to each row; carries no syntenic meaning for decoy pairs.

Delta_Background Numeric. The difference between the approximate global alignment score and the expected score under a background model derived from genome-wide residue composition.

The returned object retains the following attributes:

GeneCalls The `GeneCalls` list supplied to the function.

KmerSize The value of `K_val_01` used during the run.

Author(s)

Nicholas Cooley <Nicholas.Cooley@ul.ie>

See Also

- SummarizePairs
- NucleotideOverlap
- EvaluatePairs

Examples

```
library(DBI)
data("genecalls")
tmp01 <- system.file("extdata",
                     "example_db.sqlite",
                     package = "SynExtend")
tmp02 <- tempfile()
file.copy(from = tmp01,
          to = tmp02)

drv <- dbDriver("SQLite")
conn01 <- dbConnect(drv = drv,
                    tmp02)

x <- CreateDecoys(DataBase01 = conn01,
                  GeneCalls = genecalls,
                  K_val_01 = 5,
                  K_val_02 = 5,
                  K_val_03 = 10,
                  DefaultTranslationTable = "11",
                  TRY_limit = 1,
                  DecoyMethod = c("reverse", "none"),
                  Verbose = FALSE)
```

DecisionTree-class *Decision Trees for Random Forests*

Description

DecisionTree objects comprising random forest models generated with RandForest.

Usage

```
## S3 method for class 'DecisionTree'
as.dendrogram(object, ...)

## S3 method for class 'DecisionTree'
plot(x, plotGain=FALSE, ...)
```

Arguments

object	an object of class DecisionTree to convert to class dendrogram.
x	an object of class DecisionTree to plot.
plotGain	Logical; Determines if the Gini gain (for classification) or decrease in sum of squared error (for regression) should be plotted for each decision point of the tree. If FALSE, only plots the variable threshold for each decision point.
...	For plot, further arguments passed to plot.dendrogram and text. Arguments prefixed with "text." (e.g., text.cex) will be passed to text, and all other arguments are passed to plot.dendrogram. For as.dendrogram, ... is further arguments for consistency with the generic definition.

Details

These methods help work with DecisionTree objects, which are returned as part of RandForest. Coercion to dendrogram objects creates a 'dendrogram' corresponding to the structure of the decision tree. Each internal node possesses the standard attributes present in a 'dendrogram' object, along with the following extra attributes:

- variable: which variable was used to split at this node.
- thresh: cutoff for partitioning points; values less than thresh are assigned to the left node, and those greater than to the right node.
- gain: change in the metric to maximize. For classification trees this is the Gini Gain, and for regression trees this is the decrease in sum of squared error.

Plotting allows for extra arguments to be passed to plot and text. Arguments prefixed with 'text' are passed to text, which controls the labeling of internal nodes. Common arguments used here are text.cex, text.adj, text.srt, and text.col. All other arguments are passed to plot.dendrogram. For example, col='blue' would change the dendrogram color to blue, whereas text.col='blue' would change the interior node labels to blue (but not the dendrogram itself).

Value

as.dendrogram returns an object of class 'dendrogram'. plot returns NULL invisibly.

Warning

These functions can be quite slow for large decision trees. Usage is discouraged for trees with more than 100 internal nodes.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

See Also

RandForest

Examples

```
set.seed(199L)
n_samp <- 100L
AA <- rnorm(n_samp, mean=1, sd=5)
BB <- rnorm(n_samp, mean=2, sd=3)
CC <- rgamma(n_samp, shape=1, rate=2)
err <- rnorm(n_samp, sd=0.5)
y <- AA + BB + 2*CC + err

d <- data.frame(AA,BB,CC,y)
train_i <- 1:90
test_i <- 91:100
train_data <- d[train_i,]
test_data <- d[test_i,]

rf_regr <- RandForest(y~., data=train_data, rf.mode="regression", max_depth=5L)
if(interactive()){
  # Visualize one of the decision trees
  plot(rf_regr[[1]])
}

dend <- as.dendrogram(rf_regr[[1]])
plot(dend)
```

---

dendrapply *Apply a Function to All Nodes of a Dendrogram*

Description

Apply function FUN to each node of a dendrogram recursively. When y <- dendrapply(x, fn), then y is a dendrogram of the same graph structure as x and for each node, y.node[j] <- FUN(x.node[j], ...) (where y.node[j] is an (invalid!) notation for the j-th node of y). Also provides flexibility in the order in which nodes are evaluated.

NOTE: This man page is for the dendrapply function defined in the **SynExtend** package. See ?stats::dendrapply for the default method (defined in the **stats** package).

Usage

```
dendrapply(X, FUN, ...,
           how = c("pre.order", "post.order"))
```

Arguments

X	An object of class "dendrogram".
FUN	An R function to be applied to each dendrogram node, typically working on its attributes alone, returning an altered version of the same node.
...	potential further arguments passed to FUN.
how	Character; one of c("pre.order", "post.order"), or an unambiguous abbreviation. Determines if nodes should be evaluated according to a preorder (default) or postorder traversal. See details for more information.

Details

"pre.order" preserves the functionality of the previous dendrapply. For each node n, FUN is applied first to n, then to n[[1]] (and any children it may have), then n[[2]] and its children, etc. Notably, each node is evaluted *prior to any* of its children (i.e., "top-down").

"post.order" allows for calculations that depend on the children of a given node. For each node n, FUN is applied first to *all* children of n, then is applied to n itself. Notably, each node is evaluated *after all* of its children (i.e., "bottom-up").

Value

Usually a dendrogram of the same (graph) structure as X. For that, the function must be conceptually of the form FUN <- function(X) { attributes(X) <- .....; X }, i.e., returning the node with some attributes added or changed.

If the function provided does not return the node, the result is a nested list of the same structure as X, or as close as can be achieved with the return values. If the function should only be applied to the leaves of X, consider using rapply instead.

Warning

dendrapply identifies leaf nodes as nodes such that attr(node, 'leaf') == TRUE, and internal nodes as nodes such that attr(node, 'leaf') %in% c(NULL, FALSE). If you modify or remove this attribute, dendrapply may perform unexpectedly.

Note

The prior implementation of dendrapply was recursive and inefficient for dendrograms with many non-leaves. This version is no longer recursive, and thus should no longer cause issues stemming from insufficient C stack size (as mentioned in the 'Warning' in dendrogram).

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

See Also

as.dendrogram, lapply for applying a function to each component of a list.

rapply is particularly useful for applying a function to the leaves of a dendrogram, and almost always be used when the function does not need to be applied to interior nodes due to significantly better performance.

Examples

```
require(graphics)

## a smallish simple dendrogram
dhc <- as.dendrogram(hc <- hclust(dist(USArrests), "ave"))
(dhc21 <- dhc[[2]][[1]])

## too simple:
dendrapply(dhc21, function(n) utils::str(attributes(n)))

## toy example to set colored leaf labels :
local({
  colLab <<- function(n) {
      if(is.leaf(n)) {
        a <- attributes(n)
        i <<- i+1
        attr(n, "nodePar") <- c(a$nodePar, list(lab.col = mycols[i], lab.font = i%%3))
      }
      n
  }
  mycols <- grDevices::rainbow(attr(dhc21,"members"))
  i <- 0
 })
dL <- dendrapply(dhc21, colLab)
op <- par(mfrow = 2:1)
 plot(dhc21)
 plot(dL) ## --> colored labels!
par(op)

## Illustrating difference between pre.order and post.order
dend <- as.dendrogram(hclust(dist(seq_len(4L))))

f <- function(x){
  if(!is.null(attr(x, 'leaf'))){
    v <- as.character(attr(x, 'label'))
  } else {
    v <- paste0(attr(x[[1]], 'newattr'), attr(x[[2]], 'newattr'))
  }
  attr(x, 'newattr') <- v
  x
}

# trying with default, note character(0) entries
preorder_try <- dendrapply(dend, f)
dendrapply(preorder_try, \(x){ print(attr(x, 'newattr')); x })

## trying with postorder, note that children nodes will already
## have been populated, so no character(0) entries
postorder_try <- dendrapply(dend, f, how='post.order')
dendrapply(postorder_try, \(x){ print(attr(x, 'newattr')); x })
```

`DisjointSet` *Return single linkage clusters from* `PairSummaries` *objects.*

Description

Takes in a PairSummaries object and return a list of identifiers organized into single linkage clusters.

Usage

```
DisjointSet(Pairs,
            Verbose = FALSE)
```

Arguments

Pairs	A PairSummaries object.
Verbose	Logical indicating whether to print progress bars and messages. Defaults to FALSE.

Details

Takes in a PairSummaries object and return a list of identifiers organized into single linkage clusters.

Value

Returns a list of character vectors representing IDs of sequence features, typically genes.

Author(s)

Nicholas Cooley <npc19@pitt.edu>

See Also

FindSynteny, Synteny-class, SummarizePairs, FindSets

Examples

```
data("init_pairs", package = "SynExtend")

x <- DisjointSet(Pairs = init_pairs,
                 Verbose = TRUE)
```

DPhyloStatistic *D-Statistic for Binary States on a Phylogeny*

Description

Calculates if a presence/absence pattern is random, Brownian, or neither for a binary trait with respect to a given phylogeny.

Usage

```
DPhyloStatistic(dend, PAProfile, NumIter = 1000L)
```

Arguments

dend	An object of class dendrogram
PAProfile	A vector representing presence/absence of binary traits. See Details for information on supported input types.
NumIter	Integer; Number of iterations to simulate for random permutation analysis.

Details

This function implements the D-Statistic for binary traits on a phylogeny, as introduced in Fritz and Purvis (2009). The statstic is the following ratio:

$$\frac{D_{obs} - D_b}{D_r - D_b}$$

Here D_{obs} is the D value for the input data, D_b is the value under simulated Brownian evolution, and D_r is the value under random permutation of the input data. The D value measures the sum of sister clade differences in a phylogeny weighted by branch lengths. A score close to 1 indicates phylogenetically random distribution, and a score close to 0 indicates the trait likely evolved under Brownian motion. Scores can fall outside this range; these scores are only intended as benchmark points on the scale. See the Value section or the original paper cited in References for more information.

The input parameter PAProfile supports a number of formatting options:

- Character vector, where each element is a label of the dendrogram. Presence in the character vector indicates presence of the trait in the corresponding label.
- Integer vector of length equivalent to the number of leaves, comprised of 0s and 1s. 0 indicates absence in the corresponding leaf, and 1 indicates presence.
- Logical vector of length equivalent to number of leaves. FALSE indicates absence in the corresponding leaf, and TRUE indicates presence.

See Examples for a demonstration of each case.

Value

Returns a numerical value with the following cases:

- Value less than 0: the trait is more phylogenetically concentrated than expected by chance ("extremely clumped")
- Value close to 0: the trait is as phylogenetically concetrated as expected if it had evolved by Brownian motion
- Value close to 1: the trait is as phylogenetically concetrated as expected under a random distribution
- Value greater than 1: the trait is less phylogenetically concentrated than expected under a random distribution ("overdispersed")

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

References

Fritz S.A. and Purvis A. *Selectivity in Mammalian Extinction Risk and Threat Types: a New Measure of Phylogenetic Signal Strength in Binary Traits.* Conservation Biology, 2010. **24**(4):1042-1051.

Examples

```
############################################################
### Replicating results from Table 1 in original paper ###
############################################################

# creates a dendrogram with 16 leaves and branch lengths all 1
distMat <- suppressWarnings(matrix(seq_len(17L), nrow=16, ncol=16))
testDend <- as.dendrogram(hclust(as.dist(distMat)))
testDend <- dendrapply(testDend, \(x){
                    attr(x, 'height') <- attr(x, 'height') / 2
                    return(x)
                  })
attr(testDend[[1]], 'height') <- attr(testDend[[2]], 'height') <- 3
attr(testDend, 'height') <- 4
plot(testDend)

set.seed(123)

# extremely clumped (should be close to -2.4)
DPhyloStatistic(testDend, as.character(1:8))

# clumped Brownian (should be close to 0)
DPhyloStatistic(testDend, as.character(c(1,2,5,6,10,12,13,14)))

# random (should be close to 1.0)
DPhyloStatistic(testDend, as.character(c(1,4:6,10,13,14,16)))

# overdispersed (should be close to 1.9)
DPhyloStatistic(testDend, as.character(seq(2,16,by=2)))

##########################################
### Different ways to create PAProfiles ###
##########################################

allLabs <- as.character(labels(testDend))

# All these ways create a PAProfile with
# presence in members 1:4
# and absence in members 5:16

# numeric vector:
c(rep(1,4), rep(0, length(allLabs)-4))

# logical vector:
c(rep(TRUE,4), rep(FALSE, length(allLabs)-4))

# character vector:
allLabs[1:4]
```

`EstimateExoLabel` *Estimate ExoLabel Disk Consumption*

Description

Estimate the total disk consumption for `ExoLabel`.

Usage

```
EstimateExoLabel(num_v, avg_degree=2,
                 is_undirected=TRUE,
                 num_edges=num_v*avg_degree,
                 node_name_length=10L)
```

Arguments

num_v	Integer; approximate number of total unique nodes in the network.
avg_degree	Numeric; average degree of nodes in the network (i.e., the average number of neighbors for each node)
is_undirected	Logical; indicates whether edges are undirected (TRUE) or directed (FALSE). Undirected edges consume twice as much disk space internally because they need to be recorded twice.
num_edges	Integer; approximate total number of edges in the network.
node_name_length	Integer; approximate average length of each node name, in characters.

Details

This function provides a rough estimate of the total disk space required to run ExoLabel for a given input network. Only one of avg_degree and num_edges must be provided. The function prints out the estimated size of the original edgelist files, the estimated disk space and RAM to be consumed by ExoLabel, and the approximate ratio of disk space relative to the original file.

node_name_length specifies the average length of the node names–since the names themselves must be stored on disk, this contributes to the overall size. For relatively short node names (1-16 characters) this has a negligible impact on overall disk consumption, though it may impact the worst-case RAM consumption. Expected RAM consumption is determined by the average prefix length a random pair of vertex labels have in common, and should be closer to the minimum usage in most scenarios (see ExoLabel for more details).

Value

Invisibly returns a vector of length six, showing the estimated RAM consumption, estimated input edgelist file size, estimated disk consumption using in-place sort (use_fast_sort=FALSE), estimated disk consumption using fast sort (use_fast_sort=TRUE), estimated final file size, and ratio of the input file size to total ExoLabel disk usage. All values denote bytes.

Note

Estimating the average node label size is challenging, and unfortunately it does have a relatively large effect on the estimated edgelist file size. This function should be used for **rough** estimations of sizing, not absolute values. Errors in estimation of rough node name size will have a larger impact on edgelist file estimation than on the ExoLabel disk usage, so users can have higher confidence in estimated ExoLabel consumption.

Author(s)

Aidan Lakshman <AHL27@pitt.edu>

See Also

ExoLabel

Examples

```
# 100,000 nodes, average degree 2
EstimateExoLabel(num_v=100000, avg_degree=2)

# 10,000 nodes, 50,000 edges
EstimateExoLabel(num_v=10000, num_edges=50000)
```

EstimRearrScen *Estimate Genome Rearrangement Scenarios with Double Cut and Join Operations*

Description

Take in a Synteny object and return predicted rearrangement events.

Usage

```
EstimRearrScen(SyntenyObject, NumRuns = -1,
               Mean = FALSE, MinBlockLength = -1,
               Verbose = TRUE)
```

Arguments

SyntenyObject	Synteny object, as obtained from running FindSynteny. Expected input is unichromosomal sequences, though multichromosomal sequences are supported.
NumRuns	Numeric; The number of scenarios to simulate. The default value of -1 corresponds to $\sqrt{b}$ scenarios, where b is the number of unique breakpoints in the Synteny object.
Mean	Logical; Indicates whether to return the mean (TRUE) or minimum (FALSE) number of inversions and transpositions found across all runs.
MinBlockLength	Numeric; Minimum size of syntenic blocks to use for analysis. The default value of -1 accepts all blocks. Set to a larger value to ignore sections of short mutations that could be the result of SNPs or other small-scale mutations.
Verbose	Logical; If TRUE, displays a progress bar and prints the time difference upon completion.

Details

EstimRearrScen is an implementation of the Double Cut and Join (DCJ) method for analyzing large scale mutation events.

The DCJ model is commonly used to model genome rearrangement operations. Given a genome, we can create a connected graph encoding the order of conserved genomic regions. Each syntenic region is split into two nodes, with one encoding the beginning and one encoding the end (beginning and end defined relative to the direction of transcription). Each node is then connected to the two nodes it is adjacent to in the genome.

For example, given a genome with 3 syntenic regions $a - b - c$ such that b is transcribed in the opposite direction relative to a, c, our graph would consist of nodes and edges $a1 - a2 - b2 - b1 - c1 - c2$.

Given two genomes, we derive syntenic regions between the two samples and then construct two of these graph structures. A DCJ operation is one that cuts two connections of a common color and creates two new edges. The goal of the DCJ model is to rearrange the graph of the first genome into the second genome using DCJ operations. The DCJ distance is defined as the minimum number of DCJ operations to transform one graph into another.

It can be easily shown that inversions can be performed with a single DCJ operation, and block interchanges/order rearrangements can be performed with a sequence of two DCJ operations. DCJ distance defines a metric space, and prior work has demonstrated algorithms for fast computation of the DCJ distance.

However, DCJ distance inherently incentivizes inversions over block interchanges due to the former requiring half as many DCJ operations. This is a strong assumption, and there is no evidence to support gene order rearrangements occuring half as often as gene inversions.

This implementation incentivizes minimum number of **events** rather than number of DCJs. As the search space is large and multiple sequences of events can be equally parsimonious, this algorithm computes multiple scenarios with random sequences of operations to try to find the minimum. The `Mean` parameter controls if the function returns the best found solution (`Mean=FALSE`) or the mean number of events from all solutions (`Mean=TRUE`).

Value

An *NxN* matrix of lists with the same shape as the input Synteny object. This is wrapped into a `GenRearr` object for pretty printing.

The diagonal corresponds to total sequence length of the corresponding genome.

In the upper triangle, entry `[i,j]` corresponds to the percent hits between genome `i` and genome `j`. In the lower triangle, entry `[i,j]` contains a List object with 5 properties:

- `$Inversions` and `$Transpositions` contain the (Mean/min) number of estimated inversions and transpositions (resp.) between genome `i` and genome `j`.
- `$pct_hits` contains percent hits between the genomes.
- `$Scenario` shows the sequence of events corresponding to the minimum rearrangement scenario found. See below for details.
- `$Key` provides a mapping between syntenic blocks and genome positions. See below for details.

The `print.GenRearr` method prints this data out as a matrix, with the diagonal showing the number of chromosomes and the lower triangle displaying `xI,yT`, where `x,y` the number of inversions and transpositions (resp.) between the corresponding entries.

The `$Scenario` entry describes a sequences of steps to rearrange one genome into another, as found by this algorithm. The goal of the DCJ model is to rearrange the second genome into the first. Thus, with `N` syntenic regions total, we can arbitrarily choose the syntenic blocks in genome 1 to be ordered `1,2,...,N`, and then have genome 2 numbers relative to that.

As an example, suppose genome 1 has elements `A B E(r) G` and genome 2 has elements `E B(r) A(r) G`, with X(r) denoting block X has reversed direction of transcription. We can then arbitrarily assign blocks to numbers such that genome 1 is `(1 2 3 4)` and genome 2 is `(3 -2 -1 4)`, where a negative indicates reversed direction of transcription relative to the corresponding syntenic block in genome 1.

Each entry in `$Scenario` details an operation, the result after that operation, and the number of blocks involved in the operation. If we reversed the middle two entries of genome 2, the entry in `$Scenario` would be:

`inversion: 3 1 2 4 { 2 }`

Here we inverted the whole block `(-2 -1)` into `(1 2)`. We could then finish the rearrangement by performing a transposition to move block 3 between 2 and 4. The entries of `$Scenario` in this case would be the following:

`Original: 3 -2 -1 4`

`inversion: 3 1 2 4 { 2 }`

`block interchange: 1 2 3 4 { 3 }`

Step 1 is the original state of genome 2, step 2 inverts 2 elements to arrive at `(3 1 2 4)`, and then step 3 moves one element to arrive at `(1 2 3 4)`.

It is important to note that the numbered genomic regions in `$Scenario` are not genes, they are blocks of conserved syntenic regions between the genomes. These blocks may not match up with the original blocks from the Synteny object, since some are combined during pre-processing to expedite calculations.

`$Key` is a mapping between these numbered regions and the original genomic regions. This is a 5 column matrix with the following columns (in order):

1. `start1`: Nucleotide position for the first nucleotide in of the syntenic region on genome 1.
2. `start2`: Same as `start1`, but for genome 2
3. `length`: Length of block, in nucleotides
4. `rel_direction_on_2`: 1 if the blocks have the same transcriptonal direction on both genomes, and 0 if the direction is reversed in genome 2
5. `index1`: Label of the genetic region used in `$Scenario` output

Author(s)

Aidan Lakshman (<ahl27@pitt.edu>)

References

Friedberg, R., Darling, A. E., & Yancopoulos, S. (2008). Genome rearrangement by the double cut and join operation. *Bioinformatics*, 385-416.

See Also

`FindSynteny`

`Synteny`

Examples

```
db <- system.file("extdata", "Influenza.sqlite", package="DECIPHER")
synteny <- FindSynteny(db)
synteny

rearrs <- EstimRearrScen(synteny)

rearrs          # view whole object
rearrs[[2,1]]   # view details on Genomes 1 and 2
```

EvaluatePairs *Evaluating and Filtering Candidate Feature Pairs Against Decoy Alignments*

Description

A function for evaluating candidate genomic feature pairs in a PairSummaries object by comparing them against a set of decoy alignments. Decoys may be generated on the fly from a supplied database, or provided as a pre-built PairSummaries object. An optional evaluation method appends a model-derived or cluster-derived score to each pair. When FDRCriteria is supplied, pairs are filtered to a user-specified false discovery rate based on the ranking of a chosen column; otherwise the augmented object is returned unfiltered.

Usage

```
EvaluatePairs(InputPairs,
              DataBase01,
              StaticDecoys,
              DecoyScalar = 0.5,
              EvaluationMethod = c("none",
                                   "kmeans",
                                   "glm",
                                   "lm"),
              Verbose = FALSE,
              FDRCriteria = c("Delta_Background" = 0.001),
              ...)
```

Arguments

InputPairs	An object of class PairSummaries, typically produced by SummarizePairs. This is the set of candidate feature pairs to be evaluated.
DataBase01	Optional. Either a connection object to a DECIPHER-compatible SQLite database, or a character string giving the path to such a database on disk. When supplied and StaticDecoys is absent, decoy alignments are generated on the fly via CreateDecoys using attributes inherited from InputPairs (GeneCalls, KmerSize, DefaultTranslationTable). If a path string is supplied, the RSQLite package must be installed. At least one of DataBase01 or StaticDecoys must be provided when FDRCriteria is not NULL.
StaticDecoys	Optional. A pre-built PairSummaries object to be used as the decoy set in place of on-the-fly generation. Takes precedence over DataBase01 when both are supplied. Must be of class PairSummaries.
DecoyScalar	A numeric value strictly greater than zero and less than or equal to 1 controlling the size of the decoy set relative to InputPairs. The number of decoy rows sampled or generated is ceiling(nrow(InputPairs) * DecoyScalar). When generating decoys on the fly, this ceiling is passed as AA_limit to CreateDecoys; if fewer decoys are available than requested, all available decoys are used. Defaults to 0.5.
EvaluationMethod	A character string specifying the method used to append a criteria_value column to the combined candidate and decoy data. Must be one of:

"none" No model is fitted and no criteria_value column is appended. The function returns the augmented PairSummaries object directly (after any FDR filtering). This is the default.

"glm" A quasi-binomial generalised linear model is fitted with Response as the outcome and Consensus, FeatureDiff, MaxMatch, KDist, Local_PID, Approx_Global_PID, and Delta_Background as predictors. Fitted values from the model are stored in criteria_value.

"lm" A linear model is fitted using the same predictors as "glm", with Delta_Background as the response. Fitted values are stored in criteria_value.

"kmeans" K-means clustering is applied across the predictor columns (without a Response column). The optimal number of clusters is selected by fitting a one-site binding curve to the within-cluster sum of squares elbow and scaling by SelectScalar (default 3, adjustable via ...). Cluster assignments are stored in criteria_value and cluster centres are stored as the centers attribute of the returned object.

Partial matching is supported.

Verbose — Logical indicating whether to print status messages and the elapsed time upon completion. Defaults to FALSE.

FDRCriteria — Either NULL or a named numeric vector of length 1 specifying the false discovery rate threshold to apply. The name must match a column of the combined evaluation data (e.g. c("Delta_Background" = 0.001)). Rows are ranked in decreasing order of the named column; the cumulative proportion of decoy rows in the ranking is computed, and all rows at or beyond the point where that proportion first reaches the threshold value are discarded. Only non-decoy rows in the retained set are returned. Requires decoy alignments to be present (either via DataBase01 or StaticDecoys). Set to NULL to skip filtering. Defaults to c("Delta_Background" = 0.001).

... — Additional named arguments passed to internal helpers. Currently recognised optional arguments include K_val_02 (k-mer width for amino acid index construction in on-the-fly decoy generation; defaults to 6) and K_val_03 (k-mer width for nucleotide index construction; defaults to 10). For EvaluationMethod = "kmeans", MaxK (maximum number of clusters to test; default 15) and SelectScalar (scaling factor applied to the estimated elbow; default 3) are also accepted.

Details

The function proceeds in three stages.

Stage 1 — Decoy assembly. If DataBase01 is supplied without StaticDecoys, decoy alignments are generated by calling CreateDecoys with arguments inherited from the attributes of InputPairs. If StaticDecoys is supplied, it is used directly. In both cases, the decoy set is downsampled to ceiling(nrow(InputPairs) * DecoyScalar) rows (or all available rows if fewer exist). Candidate pairs receive a Response value of 1; decoy pairs receive 0. If neither source of decoys is supplied, all rows are assigned Response = 1 and the function proceeds without a null distribution.

Stage 2 — Evaluation. A derived predictor FeatureDiff is computed as the absolute difference in feature lengths divided by the length of the longer feature, giving a normalised measure of length asymmetry. MaxMatch is similarly normalised by the sum of the two feature lengths. Depending on EvaluationMethod, a model or clustering procedure is applied to the combined data and a criteria_value column is appended.

Stage 3 — FDR filtering. When FDRCriteria is not NULL, pairs are ranked in decreasing order of the named column and a cumulative FDR is computed as the running proportion of decoy rows. All

rows at and beyond the rank position where the FDR first meets or exceeds the threshold are discarded. Decoy rows are removed from the result unconditionally; only candidate rows that survive the threshold are returned.

Value

An object of class `c("data.frame", "PairSummaries")` containing the candidate pairs that survived evaluation and, if `FDRCriteria` was not `NULL`, FDR filtering. Columns include all columns from `InputPairs` plus:

Response Integer. `1` for candidate pairs and `0` for decoy pairs. Decoy rows are always removed before the object is returned when `FDRCriteria` is not `NULL`; when `FDRCriteria` is `NULL` the combined candidate-and-decoy data is returned and this column is present.

criteria_value Numeric or integer. Present only when `EvaluationMethod` is not `"none"`. Contains fitted values from the GLM or LM, or integer cluster assignments from k-means.

The returned object retains the following attributes inherited from `InputPairs`:

GeneCalls The named list of gene calls.

KmerSize The k-mer size used in the upstream `SummarizePairs` call.

DefaultTranslationTable The translation table identifier used upstream.

When `EvaluationMethod = "kmeans"`, the returned object additionally carries a `centers` attribute containing the k-means cluster centre matrix.

Author(s)

Nicholas Cooley <Nicholas.Cooley@ul.ie>

See Also

- SummarizePairs
- CreateDecoys
- NucleotideOverlap

Examples

```
library(DBI)
data("init_pairs")
tmp01 <- system.file("extdata",
                     "example_db.sqlite",
                     package = "SynExtend")
tmp02 <- tempfile()
file.copy(from = tmp01,
          to = tmp02)

drv <- dbDriver("SQLite")
conn01 <- dbConnect(drv = drv,
                    tmp02)

x <- EvaluatePairs(InputPairs = init_pairs,
                   DataBase01 = conn01)
```

EvoWeaver *EvoWeaver: Identifying Gene Functional Associations from Coevolutionary Signals*

Description

EvoWeaver is an S3 class with methods for predicting functional association using protein or gene data. EvoWeaver implements multiple algorithms for analyzing coevolutionary signal between genes, which are combined into overall predictions on functional association. For details on predictions, see `predict.EvoWeaver`.

Usage

```
EvoWeaver(ListOfData, MySpeciesTree=NULL, NoWarn=FALSE)

## S3 method for class 'EvoWeaver'
SpeciesTree(ew, Verbose=TRUE, ...)
```

Arguments

`ListOfData`	A list of gene data, where each entry corresponds to information on a particular gene. List must contain either dendrograms or vectors, and cannot contain a mixture. If list is composed of dendrograms, each dendrogram is a gene tree for the corresponding entry. If list is composed of vectors, vectors should be numeric or character vectors denoting the genomes containing that gene.
`MySpeciesTree`	An object of class `'dendrogram'` representing the overall species tree for the list provided in `ListOfData`.
`NoWarn`	Logical; If `FALSE`, displays warnings corresponding to which algorithms are unavailable for given input data format (see Details for more information).
`ew`	An object of class `EvoWeaver`.
`Verbose`	Logical; If `TRUE`, displays output when calculating reference tree.
`...`	Further arguments passed to `SuperTree` for inferring a reference tree.

Details

EvoWeaver expects input data to be a list. All entries must be one of the following cases:

1. `ListOfData[[i]] = c('ID#1', 'ID#2', ..., 'ID#k')`
2. `ListOfData[[i]] = c('g1_d1_s1_p1', 'g2_d2_s2_p2', ..., 'gk_dk_sk_pk')`
3. `ListOfData[[i]] = dendrogram(...)`

In (1), each ID#i corresponds to the unique identifier for genome #i. For entry #j in the list, the presence of 'ID#i' means genome #i has an ortholog for gene/protein #j.

Case (2) is the same as (1), just with the formatting of names slightly different. Each entry is of the form `g_d_p`, where `g` is the unique identifier for the genome, `d` is which chromosome the ortholog is located, `s` indicates whether the gene is on the forward or reverse strand, and `p` is what position the ortholog appears in on that chromosome. `p` must be a `numeric`. `s` must be `0` or `1`, corresponding to whether the gene is on the forward or reverse strand. Whether `0` denotes forward or reverse is inconsequential as long as the scheme is consistent. `g,d` can be any value as long as they don't contain an underscore (`'_'`).

Case (3) expects gene trees for each gene, with labeled leaves corresponding to each source genome. If `ListOfData` is in this format, taking `labels(ListOfData[[i]])` should produce a character vector that matches the format of one of the previous cases.

See the Examples section for illustrative examples.

Whenever possible, provide a full set of `dendrogram` *objects with leaf labels in form (2). This will allow the most algorithms to run. What follows is a more detailed description of which inputs allow which algorithms.*

EvoWeaver requires input of scenario (3) to use distance matrix methods, and requires input of scenario (2) (or (3) with leaves labeled according to (2)) for gene organization analyses. Sequence Level methods require dendrograms with sequence information included as the `state` attribute in each leaf node.

Note that ALL entries must belong to the same category–a combination of character vectors and dendrograms is not allowed.

Prediction of a functional association network is done using `predict(EvoWeaverObject)`. See `predict.EvoWeaver` for more information.

The `SpeciesTree` function takes in an object of class `EvoWeaver` and returns a species tree. If the object was not initialized with a species tree, it calculates one using `SuperTree`. The species tree for a EvoWeaver object can be set with `attr(ew, 'speciesTree') <- ...`.

Value

Returns a EvoWeaver object.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

See Also

`predict.EvoWeaver`, `ExampleStreptomycesData`, `BuiltInEnsembles`, `SuperTree`

Examples

```
# I'm using gene to mean either a gene or protein

## Imagine we have the following 4 genomes:
## (each letter denotes a distinct gene)
##    Genome 1: a b c d
##    Genome 2: d c e
##    Genome 3: b a e
##    Genome 4: a e

## We have 5 total genes: (a,b,c,d,e)
##    a is present in genomes 1, 3, 4
##    b is present in genomes 1, 3
##    c is present in genomes 1, 2
##    d is present in genomes 1, 2
##    e is present in genomes 2, 3, 4

## Constructing a EvoWeaver object according to (1):
l <- list()
l[['a']] <- c('1', '3', '4')
l[['b']] <- c('1', '3')
```

```
l[['c']] <- c('1', '2')
l[['d']] <- c('1', '2')
l[['e']] <- c('2', '3', '4')

## Each value of the list corresponds to a gene
## The associated vector shows which genomes have that gene
pwCase1 <- EvoWeaver(l)

## Constructing a EvoWeaver object according to (2):
##  Here we need to add in the genome, chromosome, direction, and position
##  As we only have one chromosome,
##  we can just set that to 1 for all.
##  Position can be identified with knowledge, or with
##  FindGenes(...) from DECIPHER.

## In this toy case, genomes are small so it's simple.
l <- list()
l[['a']] <- c('a_1_0_1', 'c_1_1_2', 'd_1_0_1')
l[['b']] <- c('a_1_1_2', 'c_1_1_1')
l[['c']] <- c('a_1_1_3', 'b_1_0_2')
l[['d']] <- c('a_1_0_4', 'b_1_0_1')
l[['e']] <- c('b_1_0_3', 'c_1_0_3', 'd_1_0_2')

pwCase2 <- EvoWeaver(l)

## For Case 3, we just need dendrogram objects for each
# l[['a']] <- dendrogram(...)
# l[['b']] <- dendrogram(...)
# l[['c']] <- dendrogram(...)
# l[['d']] <- dendrogram(...)
# l[['e']] <- dendrogram(...)

## Leaf labels for these will be the same as the
##  entries in Case 1.
```

`EvoWeaver-GOPreds` *Gene Organization Predictions for EvoWeaver*

Description

`EvoWeaver` incorporates four classes of prediction, each with multiple methods and algorithms. Co-localization (Coloc) methods examine conservation of relative location and relative orientation of genetic regions within the genome.

`predict.EvoWeaver` currently supports three Coloc methods:

- `'GeneDistance'`
- `'MoransI'`
- `'OrientationMI'`

Format

None.

Details

All distance matrix methods require an `EvoWeaver` object initialized with gene locations using a four number code. See `EvoWeaver` for more information on input data types.

The built-in `GeneDistance` examines relative location of genes within genomes as evidence of interaction. For a given pair of genes, the score is given by $\sum_G e^{1-|dI_G|}$, where G the set of genomes and dI_G the difference in index between the two genes in genome G. Using gene index instead of number of base pairs avoids bias introduced by gene and genome length. If a given gene is found multiple times in the same genome, the maximal score across all possible pairings for that gene is used. The score for a pair of gene groups is the mean score of all gene pairings across the groups.

`MoransI` measures the extent to which gene distances are preserved across a phylogeny. This function uses the same initial scoring scheme as `GeneDistance`. The raw scores are passed into `MoranI` to calculate spatial autocorrelation. "Space" is taken as e^{-C}, where C is the Cophenetic distance matrix calculated from the species tree of the inputs. As such, this method requires a species tree as input, which can be calculated from a set of gene trees using `SuperTree`.

`OrientationMI` uses mutual information of the relative orientation of each pair of genes. Conservation of relative orientation between gene pairs has been shown to imply functional association in prior work. This algorithm requires that the `EvoWeaver` object is initialized with a four number code, with the third number either `0` or `1`, denoting whether the gene is on the forward or reverse strand. The mutual information is calculated as:

$$\sum_{x \in X} \sum_{y \in Y} (-1)^{(x!=y)} P_{(X,Y)}(x,y) \ \log \left(\frac{P_{(X,Y)}(x,y)}{P_X(x) P_Y(y)} \right)$$

Here $X = Y = \{0, 1\}$, x is the direction of the gene with lower index, y is the direction of the gene with higher index, and $P_{(T)}(t)$ is the probability of $T = t$. Note that this is a weighted MI as introduced by Beckley and Wright (2021). The mutual information is augmented by the addition of a single pseudocount to each value, and normalized by the joint entropy of X, Y. P-values are calculated using Fisher's Exact Test on the contingency table.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

References

Beckley, Andrew and E. S. Wright. *Identification of antibiotic pairs that evade concurrent resistance via a retrospective analysis of antimicrobial susceptibility test results*. The Lancet Microbe, 2021. **2**(10): 545-554.

Korbel, J. O., et al., *Analysis of genomic context: prediction of functional associations from conserved bidirectionally transcribed gene pairs*. Nature Biotechnology, 2004. **22**(7): 911-917.

Moran, P. A. P., *Notes on Continuous Stochastic Phenomena*. Biometrika, 1950. **37**(1): 17-23.

See Also

`EvoWeaver`

`predict.EvoWeaver`

EvoWeaver Phylogenetic Profiling Predictors

EvoWeaver Phylogenetic Structure Predictors

EvoWeaver Sequence Level Predictors

Examples

```
exData <- get(data("ExampleStreptomycesData"))
ew <- EvoWeaver(exData$Genes[seq_len(10L)],
                MySpeciesTree = exData$Tree,
                NoWarn = TRUE)

## GeneDistance: co-localization based on relative gene index
gd <- predict(ew,
              Method = "GeneDistance",
              Verbose = FALSE)
head(gd)

## MoransI: phylogenetically-corrected co-localization
## (requires a species tree, which is stored in the EvoWeaver object)
mi <- predict(ew,
              Method = "MoransI",
              Verbose = FALSE)
head(mi)

## OrientationMI: mutual information of relative strand orientation
oi <- predict(ew,
              Method = "OrientationMI",
              Verbose = FALSE)
head(oi)
```

`EvoWeaver-PPPreds` *Phylogenetic Profiling Predictions for EvoWeaver*

Description

`EvoWeaver` incorporates four classes of prediction, each with multiple methods and algorithms. Phylogenetic Profiling (PP) methods examine conservation of gain/loss events within orthology groups using phylogenetic profiles constructed from presence/absence patterns.

`predict.EvoWeaver` currently supports ten PP methods:

- `'ExtantJaccard'`
- `'Hamming'`
- `'GLMI'`
- `'PAPV'`
- `'CorrGL'`
- `'ProfDCA'`
- `'Behdenna'`
- `'GLDistance'`
- `'PAJaccard'`
- `'PAOverlap'`

Format

None.

Details

Most PP methods are compatible with a EvoWeaver object initialized with any input type. See EvoWeaver for more information on input data types.

When Method='Ensemble' or Method="PhylogeneticProfiling", EvoWeaver uses methods GLMI, GLDistance, PAJaccard, and PAOverlap.

These methods use presence/absence (P/A) profiles, which are binary vectors such that 1 implies the corresponding genome has that particular gene, and 0 implies the genome does not have that particular gene.

Methods Hamming and ExtantJaccard use Hamming and Jaccard distance (respectively) of P/A profiles to determine overall score.

GLMI uses mutual information of gain/loss (G/L) vectors to determine score, employing a weighting scheme such that concordant gains/losses give positive information, discordant gains/losses give negative information, and events that do not co-occur with a gain/loss in the other gene group give no information.

PAJaccard calculates the centered Jaccard index of P/A profiles, where each clade with identical extant patterns is collapsed to a single leaf.

PAOverlap calculates the proportion of time in the ancestry that both genes cooccur relative to the total time each individual gene occurs, based on ancestral states inferred with Fitch parsimony.

PAPV calculates a p-value for P/A profiles using Fisher's Exact Test. The returned score is provided as 1-p_value so that larger scores indicate more significance, and smaller scores indicate less significance. This rescaling is consistent with the other similarity metrics in EvoWeaver. This can be used with ExtantJaccard, Hamming, or GLMI to weight raw scores by statistical significance.

ProfDCA uses the direct coupling analysis algorithm introduced by Weigt et al. (2005) to determine direct information between P/A profiles. This approach has been validated on P/A profiles in Fukunaga and Iwasaki (2022), though the implementation in EvoWeaver forsakes the persistent contrasive divergence method in favor of the the algorithm from Lokhov et al. (2018) for increased speed and exact solutions. Note that this algorithm is still extremely slow relative to the other methods despite the aforementioned runtime improvements.

Behdenna implements the method detailed in Behdenna et al. (2016) to find statistically significant interactions using co-occurence of gain/loss events mapped to ancestral states on a species tree. This method requires a species tree as input. If the EvoWeaver object is initialized with dendrogram objects, SuperTree will be used to infer a species tree.

GLDistance uses a similar method to Behdenna. This method uses Fitch Parsimony to infer where events were gained or lost on a species tree, and then looks for distance between these gain/loss events. Unlike Behdenna, this method takes into account the types of events (ex. gain/gain and loss/loss are treated differently than gain/loss). This method requires a species tree as input. If the EvoWeaver object is initialized with dendrogram objects, SuperTree will be used to infer a species tree.

CorrGL infers where events were gained or lost on a species tree as in method GLDistance, then uses a Pearson's correlation coefficient weighted by p-value to infer similarity.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

References

Behdenna, A., et al., *Testing for Independence between Evolutionary Processes.* Systematic Biology, 2016. **65**(5): p. 812-823.

Chung, N.C, et al., *Jaccard/Tanimoto similarity test and estimation methods for biological presence-absence data.* BMC Bioinformatics, 2019. **20**(S15).

Date, S.V. and E.M. Marcotte, *Discovery of uncharacterized cellular systems by genome-wide analysis of functional linkages.* Nature Biotechnology, 2003. **21**(9): p. 1055-1062.

Fukunaga, T. and W. Iwasaki, *Inverse Potts model improves accuracy of phylogenetic profiling.* Bioinformatics, 2022.

Lokhov, A.Y., et al., *Optimal structure and parameter learning of Ising models.* Science advances, 2018. **4**(3): p. e1700791.

Pellegrini, M., et al., *Assigning protein function by comparative genome analysis: Protein phylogenetic profiles.* Proceedings of the National Academy of Sciences, 1999. **96**(8) p. 4285-4288

Weigt, M., et al., *Identification of direct residue contacts in protein-protein interaction by message passing.* Proceedings of the National Academy of Sciences, 2009. **106**(1): p. 67-72.

See Also

EvoWeaver

predict.EvoWeaver

EvoWeaver Phylogenetic Structure Predictors

EvoWeaver Gene Organization Predictors

EvoWeaver Sequence Level Predictors

Examples

```
exData <- get(data("ExampleStreptomycesData"))
ew <- EvoWeaver(exData$Genes[seq_len(10L)],
                MySpeciesTree = exData$Tree,
                NoWarn = TRUE)

## ExtantJaccard: Jaccard similarity of presence/absence profiles at extant leaves
ej <- predict(ew,
              Method = "ExtantJaccard",
              Verbose = FALSE)
head(ej)

## GLMI: mutual information of gain/loss vectors
gl <- predict(ew,
              Method = "GLMI",
              Verbose = FALSE)
head(gl)

## PAJaccard: centered Jaccard index with conserved clades collapsed
pj <- predict(ew,
              Method = "PAJaccard",
              Verbose = FALSE)
head(pj)

## PAOverlap: proportion of shared ancestry based on Fitch parsimony
po <- predict(ew,
              Method = "PAOverlap",
              Verbose = FALSE)
head(po)

## GLDistance: distance between inferred ancestral gain/loss events
```

```
## (requires a species tree)
gld <- predict(ew,
               Method = "GLDistance",
               Verbose = FALSE)
head(gld)
```

EvoWeaver-PSPreds *Phylogenetic Structure Predictions for EvoWeaver*

Description

EvoWeaver incorporates four classes of prediction, each with multiple methods and algorithms. Phylogenetic Structure (PS) methods examine conservation of overall evolutionary rates within orthology groups using distance matrices constructed from each gene tree.

predict.EvoWeaver currently supports three PS methods:

- 'RPMirrorTree'
- 'RPContextTree'
- 'TreeDistance'

Format

None.

Details

All distance matrix methods require a EvoWeaver object initialized with dendrogram objects. See EvoWeaver for more information on input data types.

The RPMirrorTree method was introduced by Pazos et al. (2001). This method builds distance matrices using a nucleotide substitution model, and then calculates coevolution between gene families using the Pearson correlation coefficient of the upper triangle of the two corresponding matrices.

Experimental analysis has shown data in the upper triangle is heavily redundant and rapidly overwhelms available system memory. Previous work has incorporated dimensionality reduction such as Singular Value Decomposition (SVD) to reduce the dimensionality of the data, but this prevents parallelization of the data and doesn't solve memory issues (since SVD takes as input the entire matrix with columns corresponding to upper triangle values). EvoWeaver instead uses a seeded random projection following Achlioptas (2001) to reduce the dimensionality of the data in a reproducible and parallel-compatible way. We also utilize Spearman's ρ, which outperforms Pearson's r following dimensionality reduction.

Subsequent work by Pazos et al. (2005) and Sato et al. (2005, 2006) found multiple ways to improve predictions from the initial MirrorTree method. These methods incorporate additional phylogenetic context, and are thus called ContextTree methods. These improvements include correcting for overall evolutionary rate using a species tree and/or using projection vectors. The built-in RPContextTree method implements a species tree correction, and weights the resulting score by the normalized Hamming distance of the presence/absence profiles. This can correct for gene trees with low overlap that achieve spuriously high scores via random projection. Additional correction measures are implemented in the MTCorrection argument.

The TreeDistance method uses phylogenetic tree distance to quantify differences between gene trees. This method implements a number of metrics and groups them together to improve overall runtime. The default tree distance method is normalized Robinson-Foulds distance due to its lower

computational complexity. Other methods can be specified using the `TreeMethods` argument, which expects a character vector containing one or more of the following:

- `"RF"`: Robinson-Foulds Distance
- `"CI"`: Clustering Information Distance
- `"JRF"`: Jaccard-Robinson-Foulds Distance
- `"Nye"`: Nye Similarity
- `"KF"`: Kuhner-Felsenstein Distance
- `"all"`: All of the above methods

See the links above for more information and references. All of these metrics are accessible using the `PhyloDistance` method. Method `"JRF"` defaults to a `k` value of 4, but this can be specified further if necessary using the `JRFk` input parameter. Higher values of `k` approach the value of Robinson-Foulds distance, but these have a negligible impact on performance so use of the default parameter is encouraged for simplicity. Multiple metrics can be specified.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

References

Achlioptas, Dimitris. *Database-friendly random projections.* Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, 2001. p. 274-281.

Pazos, F. and A. Valencia, *Similarity of phylogenetic trees as indicator of protein–protein interaction.* Protein Engineering, Design and Selection, 2001. **14**(9): p. 609-614.

Pazos, F., et al., *Assessing protein co-evolution in the context of the tree of life assists in the prediction of the interactome.* J Mol Biol, 2005. **352**(4): p. 1002-15.

Sato, T., et al., *The inference of protein-protein interactions by co-evolutionary analysis is improved by excluding the information about the phylogenetic relationships.* Bioinformatics, 2005. **21**(17): p. 3482-9.

Sato, T., et al., *Partial correlation coefficient between distance matrices as a new indicator of protein-protein interactions.* Bioinformatics, 2006. **22**(20): p. 2488-92.

See Also

`EvoWeaver`

`predict.EvoWeaver`

EvoWeaver Phylogenetic Profiling Predictors

EvoWeaver Gene Organization Predictors

EvoWeaver Sequence Level Predictors

`PhyloDistance`

Examples

```
set.seed(1986)

# Build a contrived example of toy gene trees with shared leaf labels.
nGenomes  <- 8L
nGenes    <- 6L
```

```
genomeIDs <- paste0("g",
                    seq_len(nGenomes))

geneList <- vector("list", nGenes)
names(geneList) <- paste0("gene",
                          seq_len(nGenes))
for (i in seq_len(nGenes)) {
  dm <- as.dist(matrix(runif(nGenomes^2L,
                             0.1,
                             1.0),
                       nrow = nGenomes,
                       ncol = nGenomes))
  tr <- as.dendrogram(hclust(dm))
  ## Assign genome IDs as leaf labels
  tr <- dendrapply(tr, function(node) {
    if (is.leaf(node)) {
      idx <- as.integer(attr(node, "label"))
      attr(node, "label") <- genomeIDs[idx]
    }
    node
  })
  geneList[[i]] <- tr
}

ew <- EvoWeaver(geneList,
                NoWarn = TRUE)

# RPMirrorTree: MirrorTree with random-projection dimensionality reduction
rp <- predict(ew,
              Method = "RPMirrorTree",
              Verbose = FALSE)
head(rp)

# TreeDistance: pairwise Robinson-Foulds distance between gene trees
td <- predict(ew,
              Method = "TreeDistance",
              Verbose = FALSE)
head(td)
```

`EvoWeaver-SLPreds` *Sequence Level Predictions for EvoWeaver*

Description

`EvoWeaver` incorporates four classes of prediction, each with multiple methods and algorithms. Sequence Level (SL) methods examine conservation of patterns in sequence data, commonly exhibited due to physical interactions between proteins.

`predict.EvoWeaver` currently supports three SL methods:

- `'SequenceInfo'`
- `'GeneVector'`
- `'Ancestral'`

Format

None.

Details

Sequence Level methods require a `EvoWeaver` object initialized with `dendrogram` objects and sequence information stored in the leaves. See `EvoWeaver` for more information on input data types.

When `Method='Ensemble'` or `Method="SequenceLevel"`, EvoWeaver uses methods `SequenceInfo` and `GeneVector`. The argument `useDNA` switches between interpreting sequences as DNA or AA sequences.

The `SequenceInfo` method looks at mutual information between sites in a multiple sequence alignment (MSA). This approach extends prior work in Martin et al. (2005). Each site from the first gene group is paired with the site from the second gene group that maximizes their mutual information.

The `GeneVector` method uses the natural vector encoding method introduced in Zhao et al. (2022). This encodes each gene sequences as a 92-dimensional vector, with the following entries:

$$N(S) = (n_A, n_C, n_G, n_T, \quad \mu_A, \mu_C, \mu_G, \mu_T, \quad D_2^A, D_2^C, D_2^G, D_2^T, \quad n_{AA}, n_{AC}, \ldots, n$$

Here n_X is the raw total count of nucleotide X (or di/trinucleotide). For single nucleotides, we also calculate μ_X, the mean location of nucleotide X, and D_2^X, the second moment of the location of nucleotide X. The overall natural vector for a Cluster of Orthologous Genes (COG) is calculated as the normalized mean vector from the natural vectors of all component gene sequences. Interaction scores are computed using Pearson's R between each COG's natural vector. These di/trinucleotide counts are by default excluded, but can be included using the `extended=TRUE` argument. Using the extended counts has shown minimal increased accuracy at the cost of slower runtime in benchmarking.

The `Ancestral` method calculates coevolution by looking at correlation of residue mutations near the leaves of each respective gene tree.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

References

Martin, L. C., Gloor, G. B., Dunn, S. D. & Wahl, L. M, *Using information theory to search for co-evolving residues in proteins*. Bioinformatics, 2005. **21**(4116-4124).

Zhao, N., et al., *Protein-protein interaction and non-interaction predictions using gene sequence natural vector*. Nature Communications Biology, 2022. **5**(652).

See Also

`EvoWeaver`

`predict.EvoWeaver`

EvoWeaver Phylogenetic Profiling Predictors

EvoWeaver Phylogenetic Structure Predictors

EvoWeaver Gene Organization Predictors

Examples

```
set.seed(1986)
nGenomes  <- 8L
nGenes    <- 4L
seqLen    <- 30L      # make this a little simple
genomeIDs <- paste0("g", seq_len(nGenomes))

# generate a random amino-acid string of fixed length
randAA <- function(n) {
  aas <- strsplit("ARNDCQEGHILKMFPSTWYV", "")[[1L]]
  aas <- paste(sample(aas, n, replace = TRUE), collapse = "")
  return(aas)
}

geneList <- vector("list", nGenes)
names(geneList) <- paste0("gene", seq_len(nGenes))

for (i in seq_len(nGenes)) {
  dm <- as.dist(matrix(runif(nGenomes^2L,
                             0.1,
                             1.0),
                       nrow = nGenomes,
                       ncol = nGenomes))
  tr <- as.dendrogram(hclust(dm))

  ## Attach genome IDs as leaf labels and toy sequences as 'state'
  tr <- dendrapply(tr, function(node) {
    if (is.leaf(node)) {
      idx <- as.integer(attr(node, "label"))
      attr(node, "label") <- genomeIDs[idx]
      attr(node, "state") <- randAA(seqLen)
    }
    node
  })
  geneList[[i]] <- tr
}

ew <- EvoWeaver(geneList,
                NoWarn = TRUE)

# SequenceInfo: mutual information between sites in a multiple sequence alignment
si <- predict(ew,
              Method = "SequenceInfo",
              Verbose = FALSE)
head(si)

# GeneVector: Pearson correlation of natural-vector encodings
gv <- predict(ew,
              Method = "GeneVector",
              Verbose = FALSE)
head(gv)
```

EvoWeb *EvoWeb: Predictions from EvoWeaver*

Description

EvoWeb objects can be returned from predict.EvoWeaver.

This class wraps the simMat object with some other diagnostic information intended to help interpret the output of EvoWeaver predictions.

Format

An object of class "EvoWeb", which inherits from "simMat".

Details

predict.EvoWeaver returns a EvoWeb object, which bundles some methods to make formatting and printing of results slightly nicer. This currently only implements a plot function.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

See Also

predict.EvoWeaver

simMat

plot.EvoWeb

Examples

```
##############
## Prediction with built-in model and data
###############

exData <- get(data("ExampleStreptomycesData"))

# Subset isn't necessary but is faster for a working example
ew <- EvoWeaver(exData$Genes[1:10])

# default return value is a data.frame (recommended for most users)
evoweb <- predict(ew, Method='ExtantJaccard', ReturnDataFrame=FALSE)

# print out results as an adjacency matrix
print(evoweb)

# print out results as a pairwise data.frame
as.data.frame(evoweb)
```

ExampleStreptomycesData

Example EvoWeaver Input Data from Streptomyces *Species*

Description

Data from *Streptomyces* species to test EvoWeaver functionality.

Usage

```
data("ExampleStreptomycesData")
```

Format

The data contain two elements, `Genes` and `Tree`. `Genes` is a list of presence/absence vectors in the input required for `EvoWeaver`. `Tree` is a species tree used for additional input.

Details

This dataset contains a number of Clusters of Orthologous Genes (COGs) and a species tree for use with EvoWeaver. This dataset showcases an example using EvoWeaver with a list of vectors. Entries in each vector are formatted correctly for use with co-localization prediction. Each COG `i` contains entries of the form `a_b_c`, indicating that the gene was found in genome `a` on chromosome `b`, and was at the `c`'th location. The original dataset is comprised of 301 unique genomes.

See Also

`EvoWeaver`

Examples

```
exData <- get(data("ExampleStreptomycesData"))
ew <- EvoWeaver(exData$Genes[seq_len(2L)], MySpeciesTree=exData$Tree, NoWarn=TRUE)
predict(ew, Method="PAJaccard")
```

ExoLabel *ExoLabel: Out-of-Memory Fast Label Propagation*

Description

Detects communities in networks with Fast Label Propagation using disk space to drastically reduce memory overhead.

Usage

```
ExoLabel(edgelistfiles,
              outfile=tempfile(tmpdir=tempfiledir),
              mode=c("undirected", "directed"),
              add_self_loops=FALSE,
              attenuation=TRUE,
              ignore_weights=FALSE,
              iterations=0L,
              return_table=FALSE,
              use_fast_sort=TRUE,
              verbose=interactive(),
              sep='\t',
              header=FALSE,
              tempfiledir=tempdir())
```

Arguments

edgelistfiles	Character; vector of files to be processed. Each entry should be a machine-interpretable path to an edgelist file. Plaintext and gzip-compressed files are currently supported. See Details for expected format.
outfile	Character; file to write final clusters to. Can be set to a vector of filepaths to run multiple clusterings (see "Multiple Clusterings").
mode	Character; specifies whether edges should be interpreted as undirected (default) or directed. If interpreted as directed, each edge V1 V2 is interpreted as $V_1 \rightarrow V_2$. Can be "undirected", "directed", or an unambiguous abbreviation.
add_self_loops	Logical or Numeric; determines if a self-loop cutoff should be added to the network. A self-loop cutoff of value w requires that at least one incoming edge has weight w in order to assign the node to that cluster (See "Self-Loops" for more information). If TRUE, adds self-loop cutoffs of weight 1.0 to all vertices. If set to numeric value w, adds self-loop cutoffs of weight w to all nodes. Can also be set to a vector when running multiple clusterings (see "Multiple Clusterings").
attenuation	Logical or Numeric; determines if label-hop attenuation should be used. If TRUE, uses attenuation to prevent single clusters from dominating results. Can also be set to a numeric to influence the strength of attenuation (larger values produce larger clusters). See "Attenuation" for more information on this parameter. Can also be set to a vector when running multiple clusterings (see "Multiple Clustering").
ignore_weights	Logical; determines if weights should be ignored. If TRUE, all edges will be treated as an edge of weight 1. Must be set to TRUE if any of edgelistfiles are two-column tables (start->end only, lacking a weights column).
iterations	Integer; maximum number of times to process each node. If set to zero or NULL, automatically uses the square root of the max node degree. See "Algorithm Convergence" for more information.
return_table	Logical; determines how the result of clustering is returned. If FALSE (default), returns a character vector corresponding to the path of outfile. If TRUE, parses outfile using read.table and returns the result (not recommended for very large graphs).
use_fast_sort	Logical; determines how files should be sorted. If FALSE, ExoLabel will perform file sorting functions in-place. If TRUE, ExoLabel will perform its file sorting functions using a second temporary file. This is much faster than the in-place sort, but consumes twice the amount of disk space. The relative disk consumption is about the same size as the input graph for use_fast_sort=FALSE, and about double the size of the input graph for use_fast_sort=TRUE (see "Memory Consumption" and the last paragraph of "Warning" below). Set to FALSE if you're worried about disk utilization.
verbose	Logical; determines if status messages (output, progress, etc.) should be displayed while running. Output messages are reduced if running in non-interactive mode.
sep	Character; expected character that separates entries on a line in each file in edgelistfiles. Defaults to tab, as would be expected in a .tsv formatted file. Set to ',' for a .csv file. Also determines the separator used in the output table.
header	Logical or Integer; determines if the first line of edgelist files should be skipped. If logical, TRUE skips the first line of each file.. If set to an integer n, skips the first n lines. Negative values are treated as 0, and decimals are coerced to integer.

`tempfiledir` Character; vector corresponding to the location where temporary files used during execution should be stored. These temporary files are deleted after ExoLabel finishes running.

Details

ExoLabel identifies communities (clusters) in graph/network structures using a variant of Fast Label Propagation, as proposed by Traag and Subelj (2023).

However, very large graphs require too much RAM for processing on some machines. In a graph containing billions of nodes and edges, loading the entire structure into RAM is rarely feasible. ExoLabel uses disk space for storing representations of graphs. While this is slower than computing on RAM, it allows ExoLabel to scale to graphs of enormous size while only using a comparatively small amount of memory. See "Memory Consumption" for details on the total disk/memory consumption of ExoLabel.

ExoLabel expects a set of edgelist files, provided as a vector of filepaths. Each entry in the file is expected to be in the following format:

`VERTEX1<sep>VERTEX2<sep>WEIGHT<linesep>`

This line defines a single edge between vertices `VERTEX1` and `VERTEX2` with weight `WEIGHT`. `VERTEX1` and `VERTEX2` are strings corresponding to vertex names, `WEIGHT` is a numeric value that can be interpreted as a `double`. The separator `<sep>` corresponds to the argument `sep` (defaulting to tab for `.tsv` format), and `linesep` is the newline value `'\n'`.

If `ignore_weight=TRUE`, the file can be formatted as:

`VERTEX1<sep>VERTEX2<linesep>`

Note that the `VERTEX1<sep>VERTEX2<sep>WEIGHT` format is still accepted for `ignore_weight=FALSE`, but the weights will be ignored. Also note that only positive weights are recorded; negative and zero-weighted edges are ignored.

Value

Returns a list object with the parameters and result of the clustering. If using multiple clusterings, the return value is a list of lists, with each entry corresponding to the single-clustering case. This list has three entries, `parameters`, `graph_stats`, and `results`.

`parameters` is a named vector with the values of `add_self_loops`, `attenuation`, and `iterations` used for the clustering.

`graph_stats` is a named numeric vector containing the number of nodes and edges in the input graph.

`results` differs depending on the value of `return_table`.

If `return_table=TRUE`, `results` is a `data.frame` object with two columns. The first column contains the name of each vertex, and the second column contains the cluster it was assigned to.

If `return_table=FALSE`, `results` is a character vector of length 1. This vector contains the path to the file to which the clusters were written. The file is formatted as a `.tsv`, with each line containing two tab separated columns (vertex name, assigned cluster). Clusters are numbered from one to the total number of clusters.

Self-Loops

Label Propagation algorithms are susceptible to a large number of small weights outcompeting small numbers of strong edges. While self-loops can be added to mitigate this problem, they fail to scale to larger networks because noise can scale quadratically, whereas self-loops are constants.

The standard intepretation of self-loops adds a self-loop edge with fixed weight w to each node, essentially requiring any node's neighboring communities to have at least weight w to propagate. In a setting like orthology detection, spurious similarity scores will eventually outweigh both true similarities and the self-loop edges with increasing graph size.

To combat this, we treat self-loop values as a "self-loop cutoff" rather than a fixed value. Self-loop cutoffs are a value w' such that all neighboring communities must have at least one edge of weight w' in order to propagate. With this usage, even if a node has many neighbors in the same community with spurious similarities, it must have at least one neighbor in that community with a strong similarity in order for the node to join that community. This approach scales better with the size of graphs compared to the traditional usage of self-loops.

As an example, consider a node N not yet assigned to a community with 10 neighbors. Neighbors 1-9 are in community 1 with weight 0.1, and neighbor 10 is in community 2 with weight 0.8. Community 1 thus has total weight 0.9, and community 2 has weight 0.8. In the context of orthology detection, values below 0.2 are likely to be spurious. With a standard self-loop of 0.4, N would still be assigned to community 1, despite these being likely spurious. However, with a *self-loop cutoff* of 0.4, N would be assigned to community 2 because no edge in community 1 is at least 0.4.

Iterations

One of the main issues of Label Propagation algorithms is that they can fail to converge. Consider an unweighted directed graph with four nodes connected in a loop. That is, `A->B`, `B->C`, `C->D`, `D->A`. If `A,C` are in cluster 1 and `B,D` are in cluster 2, this algorithm could keep processing all the nodes in a loop and never converge. To solve this issue, we introduce an additional measure for convergence controlled by `iterations`. If `iterations=x`, then we only allow the algorithm to process each node `x` times. Once a given node has been seen `x` times, it is no longer updated. This can be manually specified, but defaults to the square root of the largest node indegree.

Attenuation

ExoLabel also incorporates label-hop attenuation to reduce the chance of a single massive cluster dominating results, as inspired by Leung et al. (2009). In short, as a particular label propagates to other nodes, its subsequent contribution diminishes. The farther a particular label is from its original source, the less its contribution. The degree to which its contribution diminishes scales dynamically based on the proportion of nodes that update on each cycle. Each node's attenuated weight is calculated as $w' = w(1 - (pd)^a)$, with w the node weight, p the proportion of nodes that changed label in the previous iteration, d the distance from the initial label, and a the attenuation power (as controlled by `attenuation`).

Passing a value of `FALSE` (equivalent to 0.0) disables attenuation entirely rather than returning all singleton clusters.

The default values of `TRUE` for `attenuation` (equivalent to 1.0) recovers the original implementation provided in Leung et al. (2009).

Multiple Clusterings

Reading in the graph object takes a large portion of the processing time. This leads to a lot of duplicated effort when trying to cluster the same network under alternative parameter settings.

Multiple clusterings on the same network are supported by passing vectors of input to `outfile` and `add_self_loops` or `attenuation`. If the length of `outfile` is greater than 1, `add_self_loops` and `attenuation` can each be set to either a single value or a vector of the same length as `outfile`. For a single value, the same parameter value will be used across all clusterings. For multiple values, the corresponding value will be used in each clustering. See "Examples" for example usage.

Note that the order to process each node is randomly initialized, so multiple runs on the same parameters may produce different results if a random seed is not set.

Warning

While this algorithm can scale to very large graphs, it does have some internal limitations. First, nodes must be comprised of no more than 255 characters. This limitation is provided to decrease memory overhead and improve runtime. This behavior is controlled by the definition of `MAX_NODE_NAME_SIZE` in `src/OnDiskLP.c`.

Second, nodes are indexed using 44-bit unsigned integers. This means that the maximum possible number of nodes available is $2^{40} - 1$, which is about 17.5 trillion. This is because ExoLabel compresses weights and node labels into a single 64-bit integer to decrease disk consumption during sorting. Weights are rescaled with $w' = log_2(w + 1)$, and the resulting value is transformed into a floating point number with a 16-bit mantissa and 4-bit exponent. This representation maintains a maximum error in precision of less than 0.05%, but does result in absolute errors getting larger as weights increase in size. For a point of reference, the error in representation is less than 0.00004 for weights in [0,1] and less than 10.5 for weights in [65,000, 70,000]. This error should be undetectable outside of extremely niche scenarios.

Third, this algorithm uses disk space to store large objects. As such, please ensure you have sufficient disk space for the graph you intend to process. While there are safeguards in the code itself, unhandleable errors can occur when the OS runs out of space. Use `EstimateExoLabel` to estimate the disk consumption of your graph, and see "Memory Consumption" for more details on how the total disk/memory consumption is calculated. Note that using `use_fast_sort=TRUE` will double the maximal disk consumption of the algorithm.

Memory Consumption

Let v be the number of unique nodes, d the average indegree of nodes, and l the average length of node labels. Note that the number of edges e is equivalent to dv.

Specific calculations for memory/disk consumption are detailed below. In summary, the absolute worst case memory consumption is roughly $(24l + 46)v$ bytes, and the maximum disk consumption during computation is $16dv$ bytes (or $32dv$ bytes if `use_fast_sort=TRUE`). In practice, the RAM consumption is closer to $46v$ bytes. The final table consumes $(2 + l + \log_{10} v)v$ bytes on disk.

ExoLabel builds a trie to keep track of vertex names. Each internal node of the trie consumes 24 bytes, and each leaf node consumes 28 bytes. The lowest possible RAM consumption of the trie (if every label is length l and shares the same prefix of length $l - 1$) is roughly $28v$ bytes, and the maximum RAM consumption (if no two node labels have any prefix in common) is $(24l + 28)v$ bytes. We can generalize this to estimate the total memory consumption as roughly $(24(l-p)+28)v$, where p is the average length of common prefix between any two node labels.

ExoLabel also uses a number of internal caches to speed up read/writes from files. These caches take around 200MB of RAM in total irrespective of graph size. Note that this calculation does not include the RAM required for R itself. It also uses an internal queue for processing nodes, which consumes roughly $10v$ bytes, and an internal index of size $8v$ bytes.

As for disk space, ExoLabel transforms the graph into a CSR-compressed network, which is split across two files: a neighbors list, and a weights list. CSR compressions also require an index, which is stored directly in the trie structure. The two files consume a total of 12 bytes per outgoing edge, for a total disk consumption of $12vd$ bytes. However, the initial reading of the edges requires 16 bytes per edge, resulting in a maximum disk consumption of $16dv$. If `use_fast_sort=TRUE`, this edge reading maximally consumes 32 bytes per edge (a maximum disk consumption of $32dv$). Note that undirected edges are stored as two directed edges, which doubles the disk consumption.

The final table returned contains vertex names and cluster numbers in human-readable format. Each line is of the format VERTEX<sep>CLUSTER, where <sep> is the argument passed to sep. Each line consumes at most $l + 2 + \log_{10} v$ bytes. In the worst case, the number of clusters is equal to the number of vertices, which have at most $\log_{10} v$ digits. The average number of digits is close to the number of digits of the largest number due to how the number of digits scales with numbers. The extra two bytes are for the sep and newline characters. Thus, the total size of the file is at most $(2+l+\log_{10} v)v$ bytes. We remove all intermediate files prior to outputting clusters, so in practical cases this should be smaller than intermediate disk consumption.

Author(s)

Aidan Lakshman <AHL27@pitt.edu>

References

Traag, V.A., and L. Subelj. *Large network community detection by fast label propagation.* Sci. Rep., 2023. **13**(2701). https://doi.org/10.1038/s41598-023-29610-z

Leung, X.Y.I., et al., *Towards real-time community detection in large networks*. Phys. Rev. E, 2009. **79**(066107). https://doi.org/10.1103/PhysRevE.79.066107

See Also

EstimateExoLabel

Examples

```
## Build an example edgelist file
num_verts <- 20L
num_edges <- 20L
all_verts <- sample(letters, num_verts)
all_edges <- vapply(seq_len(num_edges),
      \(i) paste(c(sample(all_verts, 2L),
                   as.character(round(runif(1),3))),
                 collapse='\t'),
                    character(1L))
edgefile <- tempfile()
if(file.exists(edgefile)) file.remove(edgefile)
writeLines(all_edges, edgefile)

## Run ExoLabel
res_file <- ExoLabel(edgefile)
clustering <- read.delim(res_file$result, header=FALSE)
colnames(clustering) <- c("Vertex", "Cluster")
clustering


## Can also return the result directly if the network is small enough
res <- ExoLabel(edgefile, return_table=TRUE)
print(res)


############################
### Multiple Clustering ###
############################
## Run with multiple add_self_loops values
tfs <- replicate(3, tempfile())
```

```
p2 <- ExoLabel(edgefile, tfs,
                add_self_loops=c(0,0.5,1),
                return_table = TRUE)
```

ExtractBy *Extract and organize* DNAStringSets*.*

Description

Return organized DNAStringSets based on three currently supported object combinations. First return a single DNAStringSet of feature sequences from a DFrame of genecalls and a DNAStingSet of the source assembly. Second return a list of DNAStringSets of predicted pairs from a PairSummaries object and a character string of the location of a DECIPHER SQLite database. Third return a list of DNAStringSets of predicted single linkage communities from a PairSummaries object, a character string of the location of a DECIPHER SQLite database, and a list of identifiers generated by DisjointSet.

Usage

```
ExtractBy(x,
          y,
          z,
          Verbose = FALSE)
```

Arguments

x	A PairSummaries object, or if y is a DNAStringSet, a DFrame of gene calls such as one generated by gffToDataFrame.
y	A character vector of length 1 indicating the location of a DECIPHER SQLite database. Or, if x is a DFrame, a DNAStringSet of the assembly the gene calls are called from.
z	Optional; a list of identifiers generated by DisjointSet. Or any list built along a similar format with identifiers paired to the PairSummaries object.
Verbose	Logical indicating whether to print progress bars and messages. Defaults to FALSE.

Details

All sequences are forced into the same direction based on the Strand column supplied by either the gene calls DFrame specified by x, or the GeneCalls attribute of the PairSummaries object specified by y.

Value

Return a DNAStringSet, or list of DNAStringSets arranged depending upon the objects supplied. See description.

Author(s)

Nicholas Cooley <npc19@pitt.edu>

See Also

FindSynteny, Synteny-class, SummarizePairs, DisjointSet

Examples

```
data("init_pairs", package = "SynExtend")
tmp01 <- system.file("extdata",
                     "example_db.sqlite",
                     package = "SynExtend")


# extract the pairs - DBI based connections currently not supported...
Sets <- ExtractBy(x = init_pairs,
                  y = tmp01,
                  Verbose = TRUE)
```

FastQFromSRR *Get Sequencing Data from the SRA*

Description

Get sequencing data from the SRA.

Usage

```
FastQFromSRR(SRR,
             ARGS = list("--gzip" = NULL,
                         "--skip-technical" = NULL,
                         "--readids" = NULL,
                         "--read-filter" = "pass",
                         "--dumpbase" = NULL,
                         "--split-3" = NULL,
                         "--clip" = NULL),
             KEEPFILES = FALSE)
```

Arguments

SRR	A character vector of length 1 representing an SRA Run Accession, such as one that would be passed to the prefetch, fastq-dump, or fasterq-dump functions in the SRAToolkit.
ARGS	A list representing key and value sets used to construct the call to fastq-dump, multi-argument values are passed to paste directly and should be structured accordingly.
KEEPFILES	Logical indicating whether or not keep the downloaded fastq files outside of the R session. If TRUE, downloaded files will be moved to R's working directory with the default names assigned by fastq-dump. If FALSE - the default, they are removed and only the list of QualityScaledDNAStringSets returned by the function are retained.

Details

FastQFromSRR is a barebones wrapper for fastq-dump, it is set up for convenience purposes only and does not add any additional functionality. Requires a functioning installation of the SRAtoolkit.

Value

A list of QualityScaledDNAStringSets. The composition of this list will be determined by fastq-dump's splitting arguments.

Author(s)

Nicholas Cooley <npc19@pitt.edu>

Examples

```
x <- "ERR10466327"
y <- FastQFromSRR(SRR = x)
```

FeaturesFromDF *Return a DNAStringSet of features*

Description

Given a DNAStringSet of genomic sequences, and a DataFrame created by SquaregffBy containing the corresponding genecalls for the sequence, return a DNAStringSet representing the features in the DataFrame.

Usage

```
FeaturesFromDF(Genome,
               GeneCalls,
               Index = "1")
```

Arguments

Genome	A DNAStringSet.
GeneCalls	A DataFrame created by SquaregffBy containing information on feature locations in the DNAStringSet supplied to Genome.
Index	A character of length 1 setting the first identifier position for the names of the returned DNAStringSet.

Details

Given a DataFrame with the following feature columns:

- Index - integers indicating specific DNAStrings within the DNAStringSet
- Strand - integers (limited to 0 and 1) specifying the strandedness of the feature, 0 for the + strand, and 1 for the - strand

- Range - an IRangesList populated by subfeature boundaries the given feature; either the feature bounds themselves for non-coding features, or the CDS bounds (these can be out of phase!) for coding features

DataFrames with these columns are auto-generated by SquaregffBy, but can be generated by users if necessary. With both this DataFrame and a DNAStringSet containing the genomic sequences associated with the genecalls, this function returns a DNAStringSet of the nucleotide sequences for the features present in the DataFrame. Features with subfeatures are stitched together, and features on the negative strand are reverseComplemented.

Value

A DNAStringSet of features described in a given set of genomic sequences.

Author(s)

Nicholas Cooley <Nicholas.Cooley@ul.ie>

See Also

- SquaregffBy
- SummarizePairs
- NucleotideOverlap
- FrameDownward

Examples

```
library(DBI)
data("genecalls")
tmp01 <- system.file("extdata",
                     "example_db.sqlite",
                     package = "SynExtend")
tmp02 <- tempfile()
file.copy(from = tmp01,
          to = tmp02)

drv <- dbDriver("SQLite")
conn01 <- dbConnect(drv = drv,
                    tmp02)
x <- SearchDB(dbFile = conn01,
              identifier = "1",
              nameBy = "description")

y <- FeaturesFromDF(Genome = x,
                    GeneCalls = genecalls[[1]],
                    Index = "1")
```

FindSets *Find all single linkage clusters in an undirected pairs list.*

Description

Take in a pair of vectors representing the columns of an undirected pairs list and return the single linkage clusters.

Usage

```
FindSets(p1,
         p2,
         Verbose = FALSE)
```

Arguments

p1	Column 1 of a pairs matrix or list.
p2	Column 2 of a pairs matrix or list.
Verbose	Logical indicating whether or not to display a progress bar and print the time difference upon completion.

Details

FindSets uses a version of the union-find algorithm to collect single linkage clusters from a pairs list. Currently meant to be used inside a wrapper function, but left exposed for user convenience.

Value

A two column matrix with the first column being input nodes, and the second the node representing a single linkage cluster.

Author(s)

Nicholas Cooley <npc19@pitt.edu>

See Also

SummarizePairs

Examples

```
set.seed(1986)
m <- cbind(as.integer(sample(30, size = 25,
                             replace = TRUE)),
           as.integer(sample(35, size = 25,
                             replace = TRUE)))

Levs <- unique(c(m[, 1],
                 m[, 2]))
m <- cbind("1" = as.integer(factor(x = m[, 1L],
                                   levels = Levs)),
           "2" = as.integer(factor(x = m[, 2L],
```

```
                                    levels = Levs)))
z <- FindSets(p1 = m[, 1],
              p2 = m[, 2])
```

`FitchParsimony` *Calculate ancestral states using Fitch Parsimony*

Description

Ancestral states for binary traits can be inferred from presence/absence patterns at the tips of a dendrogram using Fitch Parsimony. This function works for an arbitrary number of states on bifurcating `dendrogram` objects.

Usage

```
FitchParsimony(dend, num_traits, traits_list,
                  initial_state=rep(0L,num_traits),
                  fill_ambiguous=TRUE)
```

Arguments

`dend`	An object of class `'dendrogram'`
`num_traits`	Integer; The number of traits to inferred.
`traits_list`	A list of character vectors, where the `i`'th entry corresponds to the leaf labels that have the trait `i`.
`initial_state`	Integer; The state assumed for the root node. Set to `NULL` to disable autofilling the root state.
`fill_ambiguous`	Logical; Determines if states that remain ambiguous after completion of the algorithm should be filled in randomly.

Details

Fitch Parismony allows for fast inference of ancestral states of binary traits. The algorithm proceeds in three steps.

First, traits are inferred upwards based on child nodes. If the child nodes have the same state (`1/1` or `0/0`), then the parent node is also set to that state. If the states are different, the parent node is set to `2`, denoting an ambiguous entry. If one child is ambiguous and the other is not, the parent is set to the non-ambiguous entry.

Second, traits are inferred downward to attempt to fill in ambiguous entries. If a node is not ambiguous but its child is, the child's state is set to the parent state. If specified, the root node's state is set to `initial_state` prior to this step.

Third, traits that remain ambiguous are optionally filled in (only if `fill_ambiguous` is set to `TRUE`). This proceeds by randomly setting ambiguous traits to either `1` or `0`.

The result is stored in the `FitchState` attribute within each node.

Value

A dendrogram with attribute `FitchState` set for each node, where this attribute is a binary vector of length `num_traits`.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

References

Fitch, Walter M. *Toward defining the course of evolution: minimum change for a specific tree topology.* Systematic Biology, 1971. **20**(4): p. 406-416.

Examples

```
d <- as.dendrogram(hclust(dist(USArrests), "ave"))
labs <- labels(d)

# Defining some presence absence patterns
set.seed(123L)
pa_1 <- sample(labs, 15L)
pa_2 <- sample(labs, 20L)

# inferring ancestral states
fpd <- FitchParsimony(d, 2L, list(pa_1, pa_2))

# Checking a state
attr(fpd[[1L]], 'FitchState')

# Visualizing the results for the first pattern
# Tips show P/A patterns, edges show gain/loss (green/red)
fpd <- dendrapply(fpd, \(x){
  ai <- 1L
  s <- attr(x, 'FitchState')
  l <- list()

  if(is.leaf(x)){
    # coloring tips based presence/absence
    l$col <- ifelse(s[ai]==1L, 'green', 'red')
    l$pch <- 19
    attr(x, 'nodePar') <- l
  } else {
    # coloring edges based on gain/loss
    for(i in seq_along(x)){
      sc <- attr(x[[i]], 'FitchState')
      if(s[ai] != sc[ai]){
        l$col <- ifelse(s[ai] == 1L, 'red', 'green')
      } else {
        l$col <- 'black'
      }
      attr(x[[i]], 'edgePar') <- l
    }
  }

  x
}, how='post.order')
plot(fpd, leaflab='none')
```

FrameDownward *Adjust a DataFrame of genecalls*

Description

Reframe a DataFrame of genecalls into an explicitly square representation of the features.

Usage

```
FrameDownward(genecalls)
```

Arguments

genecalls	A DataFrame object created by SquaregffBy.

Details

Given a DataFrame with a Range column containing an IRangesList, return a data.frame the length of the unlisted IRangesList. Each row represents the coordinates of a subfeature as described by the contents of the IRangesList. Key and SubKey columns can be used to trace child features back to their parents.

Value

A data.frame of of feature positions.

Author(s)

Nicholas Cooley <Nicholas.Cooley@ul.ie>

See Also

- SquaregffBy
- FindSynteny
- NucleotideOverlap

Examples

```
library(rtracklayer)
grange_obj <- import(con = system.file("extdata",
                                       "GCF_023585725.1_ASM2358572v1_genomic.gff.gz",
                                       package = "SynExtend"),
                     format = "gff")
ImportedGFF <- SquaregffBy(gff_object = grange_obj,
                           verbose = TRUE)
res <- FrameDownward(ImportedGFF)
```

genecalls *Example genecall data*

Description

A list of DataFrames.

Usage

```
data("genecalls", package = "SynExtend")
```

Format

A list of DataFrames.

Details

A list of genecalls generated by the extdata.R script contained in SynExtend's inst/scripts folder. This object contains succinct data for runnable function examples.

Examples

```
data("genecalls", package = "SynExtend")
```

Generic *Model for predicting PID based on k-mer statistics*

Description

Though the function PairSummaries provides an argument allowing users to ask for alignments, given the time consuming nature of that process on large data, models are provided for predicting PIDs of pairs based on k-mer statistics without performing alignments.

Usage

```
data("Generic")
```

Format

The format is an object of class "glm".

Details

A model for predicting the PID of a pair of sequences based on the k-mers that were used to link the pair.

Examples

```
data(Generic)
```

GetTranslatedFeatures *Translate Nucleotide Features to Amino Acid Sequences*

Description

A lightweight function for translating a set of nucleotide feature sequences into amino acid sequences. Only features that are flagged as coding and whose total coding length is in phase will be translated. Multiple genetic codes may be applied in a single call when different features carry different translation table annotations. The original feature order is preserved in the returned `AAStringSet`.

Usage

```
GetTranslatedFeatures(Nucs,
                      GeneCalls,
                      DefaultTranslationTable = "11")
```

Arguments

`Nucs`	A `DNAStringSet` of nucleotide feature sequences to be translated, typically produced by `FeaturesFromDF`. Names are expected to follow the internal `SynExtend` naming convention (underscore-delimited fields, with the feature index in the third position) so that output ordering can be restored after grouping by genetic code.
`GeneCalls`	A `DataFrame` of gene calls corresponding to the features in `Nucs`. Must contain at least the following columns: `Coding` Logical vector indicating whether each feature is a coding sequence. `Translation_Table` Character vector of NCBI genetic code identifiers (e.g. `"11"`), one per feature. `NA` values are replaced by `DefaultTranslationTable` for features that are both coding and in-frame. `Range` A list of `IRanges` or `GRanges` objects giving the genomic coordinates of each feature. The sum of widths for each element is used to determine whether the feature length is a multiple of three.
`DefaultTranslationTable`	A character string of length 1 specifying the NCBI genetic code identifier to use for coding, in-frame features whose `Translation_Table` entry is `NA`. Defaults to `"11"` (the bacterial, archaeal, and plant plastid code). Must be a valid identifier accepted by `Biostrings::getGeneticCode`.

Details

Translation is restricted to features satisfying all three conditions: the feature is flagged as coding (`GeneCalls$Coding == TRUE`), its total nucleotide length is a multiple of three (sum of `width(GeneCalls$Range)` modulo 3 equals zero), and a translation table is available after substituting `DefaultTranslationTable` for any `NA` entries.

Features meeting these criteria are grouped by their genetic code identifier and translated together using `Biostrings::translate` with `if.fuzzy.codon = "solve"`. When more than one genetic code is present, the per-code results are concatenated and reordered to match the original position order of `Nucs`, relying on the feature index encoded in the third underscore-delimited field of each sequence name.

Features that are non-coding or not in-frame are silently excluded; the returned AAStringSet therefore contains fewer sequences than Nucs whenever such features are present.

Value

An AAStringSet containing the translated amino acid sequences for all coding, in-frame features in Nucs, in the same relative order as their source sequences appear in Nucs. Names are inherited from the corresponding entries in Nucs.

Author(s)

Nicholas Cooley <Nicholas.Cooley@ul.ie>

See Also

- FeaturesFromDF
- SummarizePairs
- CreateDecoys

Examples

```
library(DBI)
data("genecalls")
tmp01 <- system.file("extdata",
                     "example_db.sqlite",
                     package = "SynExtend")
tmp02 <- tempfile()
file.copy(from = tmp01,
          to = tmp02)

drv <- dbDriver("SQLite")
conn01 <- dbConnect(drv = drv,
                    tmp02)
x <- SearchDB(dbFile = conn01,
              identifier = "1",
              nameBy = "description")

y <- FeaturesFromDF(Genome = x,
                    GeneCalls = genecalls[[1]],
                    Index = "1")

z <- GetTranslatedFeatures(Nucs = y,
                           GeneCalls = genecalls[[1]],
                           DefaultTranslationTable = "11")
```

HitConsensus — *Return a numeric measure of whether kmer hits linking two genomic features are in linearly similar locations in both features.*

Description

This function is designed to work internally to SummarizePairs so it works on relatively simple atomic vectors and has little overhead checking.

Usage

```
HitConsensus(gene1left,
             gene2left,
             gene1right,
             gene2right,
             strand1,
             strand2,
             hit1left,
             hit1right,
             hit2left,
             hit2right)
```

Arguments

gene1left	Integer; feature bound positions in nucleotide space.
gene2left	Integer; feature bound positions in nucleotide space.
gene1right	Integer; feature bound positions in nucleotide space.
gene2right	Integer; feature bound positions in nucleotide space.
strand1	Logical; is feature 1 on the positive or negative strand
strand2	Logical; is feature 2 on the positive or negative strand
hit1left	Integer; kmer hit bound positions in nucleotide space.
hit1right	Integer; kmer hit bound positions in nucleotide space.
hit2left	Integer; kmer hit bound positions in nucleotide space.
hit2right	Integer; kmer hit bound positions in nucleotide space.

Details

HitConsensus calculates whether the distances between the bounds of a kmer hit and the feature bounds are different between the features linked by the kmer.

Value

A vector of numerics.

Author(s)

Nicholas Cooley <npc19@pitt.edu>

See Also

NucleotideOverlap, SummarizePairs, FindSynteny

Examples

```
# no current example...
```

init_pairs *Example genecall data*

Description

An object of class LinkedPairs

Usage

```
data("init_pairs", package = "SynExtend")
```

Format

An object of class LinkedPairs.

Details

An object of class PairSummaries generated by the extdata.R script contained in SynExtend's inst/scripts folder. This object contains succinct data for runnable function examples.

Examples

```
data("init_pairs", package = "SynExtend")
```

LinkedPairs *Tables of where syntenic hits link pairs of genes*

Description

Syntenic blocks describe where order is shared between two sequences. These blocks are made up of exact match hits. These hits can be overlayed on the locations of sequence features to clearly illustrate where exact sequence similarity is shared between pairs of sequence features.

Usage

```
## S3 method for class 'LinkedPairs'
print(x,
      quote = FALSE,
      right = TRUE,
      ...)
```

Arguments

x	An object of class LinkedPairs.
quote	Logical indicating whether to print the output surrounded by quotes.
right	Logical specifying whether to right align strings.
...	Other arguments for print.

Details

Objects of class LinkedPairs are stored as square matrices of list elements with dimnames derived from the dimnames of the object of class "Synteny" from which it was created. The diagonal of the matrix is only filled if OutputFormat "Comprehensive" is selected in NucleotideOverlap, in which case it will be filled with the gene locations supplied to GeneCalls. The upper triangle is always filled, and contains location information in nucleotide space for all syntenic hits that link features between sequences in the form of an integer matrix with named columns. "QueryGene" and "SubjectGene" correspond to the integer rownames of the supplied gene calls. "QueryIndex" and "SubjectIndex" correspond to "Index1" and "Index2" columns of the source synteny object position. Remaining columns describe the exact positioning and size of extracted hits. The lower triangle is not filled if OutputFormat "Sparse" is selected and contains relative displacement positions for the 'left-most' and 'right-most' hit involved in linking the particular features indicated in the related line up the corresponding position in the upper triangle.

The object serves only as a simple package for input data to the PairSummaries function, and as such may not be entirely user friendly. However it has been left exposed to the user should they find this data interesting.

Value

An object of class "LinkedPairs".

Author(s)

Nicholas Cooley <npc19@pitt.edu>

Examples

```
data("linked_features", package = "SynExtend")

# Inspect the object class and dimensions
class(linked_features)
dim(linked_features)

# Print the object (upper triangle contains hit-linking data)
print(linked_features)

# Subsetting: extract the block linking sequences 1 and 2
block_1_2 <- linked_features[1, 2]
str(block_1_2)
```

---

linked_features *Example genecall data*

Description

An object of class LinkedPairs

Usage

```
data("linked_features", package = "SynExtend")
```

Format

An object of class LinkedPairs.

Details

An object of class LinkedPairs generated by the extdata.R script contained in SynExtend's inst/scripts folder. This object contains succinct data for runnable function examples.

Examples

```
data("linked_features", package = "SynExtend")
```

MakeBlastDb *Create a BLAST Database from R*

Description

Wrapper to create BLAST databases for subsequent queries using the commandline BLAST tool directly from R. Can operate on an XStringSet or a FASTA file.

This function requires the BLAST+ commandline tools, which can be downloaded here.

Usage

```
MakeBlastDb(seqs, dbtype=c('prot', 'nucl'),
          dbname=NULL, dbpath=NULL,
          extraArgs='', createDirectory=FALSE,
          verbose=TRUE)
```

Arguments

seqs	Sequence(s) to create a BLAST database from. This can be either an XStringSet or a path to a FASTA file.
dbtype	Character; Either 'prot' for amino acid input, 'nucl' for nucleotide input, or an unambiguous abbreviation.
dbname	Character; Name of the resulting database. If not provided, defaults to a random string prefixed by blastdb.
dbpath	Character; Path where database should be created. If not provided, defaults to TMPDIR.
extraArgs	Character; Additional arguments to be passed to the query executed on the command line. This should be a single string.
createDirectory	Logical; Determines if a directory should be created for the database if it doesn't already exist. If FALSE, the function will throw an error instead of creating a directory.
verbose	Logical; Determines if status messages should be displayed while running.

Details

MakeBlastDb is a barebones wrapper for makeblastdb from the BLAST+ commandline tools. It is set up for convenience purposes only and does not add any additional functionality. Requires a functioning installation of the BLAST+ commandline tools.

Value

Returns a length 2 named character vector specifying the name of the BLAST database and the path to it.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

See Also

BlastSeqs

Examples

```
#
```

MoranI *Moran's* I *Spatial Autocorrelation Index*

Description

Calculates Moran's *I* to measure spatial autocorrelation for a set of signals dispersed in space.

Usage

```
MoranI(values,
       weights,
       alternative=c('two.sided', 'less', 'greater'))
```

Arguments

values	Numeric; Vector containing signals for each point in space.
weights	Numeric object of class dist with Size attribute equivalent to the length of values, representing distances between each point in space.
alternative	Character; determines how p-value should be calculated for hypothesis testing against the null of no spatial correlation. Should be one of c("two.sided", "less", "greater"), or an unambiguous abbreviation.

Details

Moran's *I* is a measure of how much the spatial arrangement of a set of datapoints correlates with the value of each datapoint. The index takes a value in the range $[-1, 1]$, with values close to 1 indicating high correlation between location and value (points have increasingly similar values as they increase in proximity), values close to -1 indicating anticorrelation(points have increasingly different values as they increase in proximity), and values close to 0 indicating no correlation.

The value itself is calculated as:

$$I = \frac{N}{W} \frac{\sum_i^N \sum_j^N w_{ij}(x_i - \bar{x})(x_j - \bar{x})}{\sum_i^N (x_i - \bar{x})^2}$$

Here, N is the number of points, w_{ij} is the distance between points i and j, $W = \sum_{i,j} w_{ij}$ (the sum of all the weights), x_i is the value of point i, and $\bar{x}$ is the sample mean of the values.

Moran's *I* has a closed form calculation for variance and expected value, which are calculated within this function. The full form of the variance is fairly complex, but all the equations are available for reference here.

A p-value is estimated using the expected value and variance using a null hypothesis of no spatial autocorrelation, and the alternative hypothesis specified in the `alternative` argument. Note that if fewer than four datapoints are supplied, the variance of Moran's I is infinite. The function will return a standard deviation of `Inf` and a p-value of 1 in this case.

Value

A `list` object containing the following named values:

- `observed`: The value of Moran's *I* (numeric in the range $[-1, 1]$).
- `expected`: The expected value of Moran's *I* for the input data.
- `sd`: The standard deviation of Moran's *I* for the input data.
- `p.value`: The p-value for the input data, calculated with the alternative hypothesis as specified in `alternative`.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

References

Moran, P. A. P., *Notes on Continuous Stochastic Phenomena*. Biometrika, 1950. **37**(1): 17-23.

Gittleman, J. L. and M. Kot., *Adaptation: Statistics and a Null Model for Estimating Phylogenetic Effects*. Systematic Zoology, 1990. **39**:227-241.

Examples

```
# Make a distance matrix for a set of 50 points
# These are just random numbers in the range [0.1,2]
NUM_POINTS <- 50
distmat <- as.dist(matrix(runif(NUM_POINTS**2, 0.1, 2),
                          ncol=NUM_POINTS))

# Generate some random values for each of the points
vals <- runif(NUM_POINTS, 0, 3)

# Calculate Moran's I
MoranI(vals, distmat, alternative='two.sided')

# effect size should be pretty small
# and p-value close to 0.5
# since this is basically random data
```

NormVec *Unit normalize a vector*

Description

This function is designed to work internally to functions within SynExtend so it works on relatively simple atomic vectors and has little overhead checking.

Usage

```
NormVec(vec)
```

Arguments

vec	A numeric or integer vector.

Details

NormVec unit normalized a vector.

Value

A numeric vector the same length as the input.

Author(s)

Nicholas Cooley <npc19@pitt.edu>

See Also

NucleotideOverlap, SummarizePairs

Examples

```
x <- NormVec(rnorm(n = 50, mean = 2, sd = 2))
```

NucleotideOverlap *Tabulating Features Linked by Syntenic Hits*

Description

A function for concisely tabulating where genomic features are connected by syntenic hits.

Usage

```
NucleotideOverlap(SyntenyObject,
                  GeneCalls,
                  LimitIndex = FALSE,
                  AcceptContigNames = TRUE,
                  Verbose = FALSE)
```

Arguments

SyntenyObject	An object of class Synteny built from the FindSynteny function in the package DECIPHER.
GeneCalls	A named list of DataFrames built from SquaregffBy, objects of class GRanges imported from rtracklayer::import, or objects of class Genes created from the DECIPHER function FindGenes. DataFrames built by SquaregffBy can be used directly, while GRanges objects may also be used with limited functionality. Objects of class Genes generated by FindGenes function equivalently to those produced by SquaregffBy.
LimitIndex	Logical indicating whether to limit which indices in a synteny object to query. FALSE by default, when TRUE only the first sequence in all selected identifiers will be used. LimitIndex can be used to skip analysis of plasmids, or solely query a single chromosome.
AcceptContigNames	Match names of contigs between gene calls object and synteny object. Where relevant, the first white space and everything following are removed from contig names. If "TRUE", NucleotideOverlap assumes that the contigs at each position in the synteny object and "GeneCalls" object are in the same order. Is automatically set to TRUE when "GeneCalls" are of class "GRanges".
Verbose	Logical indicating whether or not to display a progress bar and print the time difference upon completion.

Details

Builds a matrix of lists that contain information about linked pairs of genomic features.

Value

A matrix of lists with location information about syntenic hits that link features. The upper triangle is populated by information about the hits themselves, while the lower triangle is populated by summary statistics of the linked features.

Author(s)

Nicholas Cooley <Nicholas.Cooley@ul.ie>

See Also

- FindSynteny

Examples

```
data("genecalls", package = "SynExtend")
data("syn", package = "SynExtend")

x <- NucleotideOverlap(SyntenyObject = syn,
                       GeneCalls = genecalls,
                       LimitIndex = FALSE,
                       Verbose = TRUE)
```

OneSite *Calculate a site on a right hyperbola.*

Description

This function is designed to work internally to functions within SynExtend so it works on relatively simple atomic vectors and has little overhead checking.

Usage

```
OneSite(X,
        Bmax,
        Kd)
```

Arguments

X	Numeric; an x coordinate value.
Bmax	Numeric; an asymptotic value.
Kd	Numeric; the half-max of the right hyperbola.

Details

OneSite calculates the Y-value for a given X-value on a right hyperbola.

Value

A numeric of length 1.

Author(s)

Nicholas Cooley <npc19@pitt.edu>

See Also

NucleotideOverlap, SummarizePairs

Examples

```
x <- OneSite(X = 3,
             Bmax = 10,
             Kd = 3)

# plot(x = 1:10, y = vapply(X = 1:10, FUN = function(x) {OneSite(X = x, Bmax = 5, Kd = 2)}, FUN.VALUE = vector(mod
```

PhyloDistance *Calculate Distance between Unrooted Phylogenies*

Description

Calculates distance between two unrooted phylogenies using a variety of metrics.

Usage

```
PhyloDistance(dend1, dend2,
              Method=c("CI", "RF", "KF", "JRF"),
              RawScore=FALSE, JRFExp=2)
```

Arguments

dend1	An object of class dendrogram, representing an unrooted bifurcating phylogenetic tree.
dend2	An object of class dendrogram, representing an unrooted bifurcating phylogenetic tree.
Method	Character; Method to use for calculating tree distances. The following values are supported: "CI", "RF", "KF", "JRF". See Details for more information.
RawScore	Logical; Determines if the function should return the distance between two trees (FALSE) or the component values used to calculate the distance (TRUE). See the pages specific to each algorithm for more information on what values are reported.
JRFExp	k-value used in calculation of JRF Distance. Unused if Method is not "JRF".

Details

This function implements a variety of tree distances, specified by the value of Method. The following values are supported, along with links to documentation pages for each function:

- "RF": Robinson-Foulds Distance
- "CI": Clustering Information Distance
- "JRF": Jaccard-Robinson-Foulds Distance, equivalent to the Nye Distance Metric when JRFExp=1
- "KF": Kuhner-Felsenstein Distance

Information on each of these algorithms, how scores are calculated, and references to literature can be found at the above links. Method "CI" is selected by default due to recent work showing this method as the most robust tree distance metric under general conditions.

Value

Returns a normalized distance, with 0 indicating identical trees and 1 indicating maximal difference. If the trees have no leaves in common, the function will return 1 if RawScore=FALSE, or c(0,NA,NA) if RawScore=TRUE.

If RawScore=TRUE, returns a vector of the components used to calculate the distance. This is typically a length 3 vector, but specific details can be found on the description for each algorithm linked above.

Note

Note that this function requires the input dendrograms to be labeled alike (ex. leaf labeled abc in dend1 represents the same species as leaf labeled abc in dend2). Labels can easily be modified using dendrapply.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

See Also

Robinson-Foulds Distance

Clustering Information Distance

Jaccard-Robinson-Foulds Distance

Kuhner-Felsenstein Distance

Examples

```
# making some toy dendrograms
set.seed(123)
dm1 <- as.dist(matrix(runif(64, 0.5, 5), ncol=8))
dm2 <- as.dist(matrix(runif(64, 0.5, 5), ncol=8))

tree1 <- as.dendrogram(hclust(dm1))
tree2 <- as.dendrogram(hclust(dm2))

# Robinson-Foulds Distance
PhyloDistance(tree1, tree2, Method="RF")

# Clustering Information Distance
PhyloDistance(tree1, tree2, Method="CI")

# Kuhner-Felsenstein Distance
PhyloDistance(tree1, tree2, Method="KF")

# Nye Distance Metric
PhyloDistance(tree1, tree2, Method="JRF", JRFExp=1)

# Jaccard-Robinson-Foulds Distance
PhyloDistance(tree1, tree2, Method="JRF", JRFExp=2)
```

PhyloDistance-CIDist *Clustering Information Distance*

Description

Calculate distance between two unrooted phylogenies using mutual clustering information of branch partitions.

Details

This function is called as part of PhyloDistance and calculates tree distance using the clustering information approach first described in Smith (2020). This function iteratively pairs internal tree branches of a phylogeny based on their similarity, then scores overall similarity as the sum of these measures. The similarity score is then converted to a distance by normalizing by the average entropy of the two trees. This metric has been demonstrated to outperform numerous other metrics in capabilities; see the original publication cited in References for more information.

Users may wish to use the actual similarity values rather than a distance metric; the option to specify RawScore=TRUE is provided for this case. Distance is calculated as $\frac{M-S}{M}$, where $M = \frac{1}{2}(H_1+H_2)$, H_i is the entropy of the i'th tree, and S is the similarity score between them. As shown in the original publication, this satisfies the necessary requirements to be considered a distance metric. Setting RawScore=TRUE will instead return a vector with (S, H_1, H_2, p), where p is an approximation for the two sided p-value of the result based on random simulations from Smith (2020).

Value

Returns a normalized distance, with 0 indicating identical trees and 1 indicating maximal difference. Note that branch lengths are not considered, so two trees with different branch lengths may return a distance of 0.

If RawScore=TRUE, returns a named length 4 vector with the first entry the similarity score, subsequent entries the entropy values for each tree, and the last entry the approximate p-value for the result based on simulations.

If the trees have no leaves in common, the function will return 1 if RawScore=FALSE, and c(0, NA, NA, NA) if TRUE.

Note

Note that this function requires the input dendrograms to be labeled alike (ex. leaf labeled abc in dend1 represents the same species as leaf labeled abc in dend2). Labels can easily be modified using dendrapply.

WARNING: This function is not the same as the implementation in ape. In this package, we use a rooted representation of the tree when calculating clades, so there is an additional partition compared to a purely unrooted bipartition view. Additionally, we use normalized similarity, whereas ape uses variation of information.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

References

Smith, Martin R. *Information theoretic generalized Robinson–Foulds metrics for comparing phylogenetic trees.* Bioinformatics, 2020. **36**(20):5007-5013.

Examples

```
# making some toy dendrograms
set.seed(123)
dm1 <- as.dist(matrix(runif(64, 0.5, 5), ncol=8))
dm2 <- as.dist(matrix(runif(64, 0.5, 5), ncol=8))
```

```
tree1 <- as.dendrogram(hclust(dm1))
tree2 <- as.dendrogram(hclust(dm2))

# get CI distance
PhyloDistance(tree1, tree2, Method="CI")

# get similarity score with individual entropies
PhyloDistance(tree1, tree2, Method="CI", RawScore=TRUE)
```

PhyloDistance-JRFDist *Jaccard-Robinson-Foulds Distance and Nye Similarity*

Description

Calculate JRF distance between two unrooted phylogenies. Nye Similarity is a special case of JRF distance, obtained when the JRF exponent k is set to 1.

Details

This function is called as part of PhyloDistance and calculates the Jaccard-Robinson-Foulds distance between two unrooted phylogenies. Each dendrogram is first pruned to only internal branches implying a partition in the shared leaf set; trivial partitions (where one leaf set contains 1 or 0 leaves) are ignored.

The total score is calculated by pairing branches and scoring their similarity. For a set of two branches A, B that partition the leaves into (A_1, A_2) and (B_1, B_2) (resp.), the distance between the branches is calculated as:

$$2 - 2\left(\frac{|X \cap Y|}{|X \cup Y|}\right)^k$$

where $X \in (A_1, A_2),\ Y \in (B_1, B_2)$ are chosen to maximize the score of the pairing, and k the value of ExpVal. The sum of these scores for all branches produces the overall distance between the two trees, which is then normalized by the number of branches in each tree.

There are a few special cases to this distance. If JRFExp=1, the distance is equivalent to the metric introduced in Nye et al. (2006). As JRFExp approaches infinity, the value becomes close to the (non-Generalized) Robinson Foulds Distance.

Value

Returns a normalized distance, with 0 indicating identical trees and 1 indicating maximal difference.

If RawScore=TRUE, returns a named length 3 vector with the first entry the summed distance score over the branch pairings, and the subsequent entries the number of partitions for each tree.

If the trees have no leaves in common, the function will return 1 if RawScore=FALSE, and c(0, NA, NA) if TRUE.

Note

Note that this function requires the input dendrograms to be labeled alike (ex. leaf labeled abc in dend1 represents the same species as leaf labeled abc in dend2). Labels can easily be modified using dendrapply.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

References

Nye, T. M. W., Liò, P., & Gilks, W. R. *A novel algorithm and web-based tool for comparing two alternative phylogenetic trees.* Bioinformatics, 2006. **22**(1): 117–119.

Böcker, S., Canzar, S., & Klau, G. W.. *The generalized Robinson-Foulds metric*. Algorithms in Bioinformatics, 2013. **8126**: 156–169.

Examples

```
# making some toy dendrograms
set.seed(123)
dm1 <- as.dist(matrix(runif(64, 0.5, 5), ncol=8))
dm2 <- as.dist(matrix(runif(64, 0.5, 5), ncol=8))

tree1 <- as.dendrogram(hclust(dm1))
tree2 <- as.dendrogram(hclust(dm2))

# Nye Metric
PhyloDistance(tree1, tree2, Method="JRF", JRFExp=1)

# Jaccard-RobinsonFoulds
PhyloDistance(tree1, tree2, Method="JRF", JRFExp=2)

# Good approximation to RF Dist (note RFDist is much faster for this)
PhyloDistance(tree1, tree2, Method="JRF", JRFExp=1000)
PhyloDistance(tree1, tree2, Method="RF")
```

PhyloDistance-KFDist *Kuhner-Felsenstein Distance*

Description

Calculate KF distance between two unrooted phylogenies.

Details

This function is called as part of PhyloDistance and calculates Kuhner-Felsenstein distance between two unrooted phylogenies. Each dendrogram is first pruned to only internal branches implying a partition in the shared leaf set; trivial partitions (where one leaf set contains 1 or 0 leaves) are ignored. The total score is calculated as the sum of squared differences between lengths of branches implying equivalent partitions. If a particular branch is unique to a given tree, it is treated as having length 0 in the other tree. The final score is normalized by the sum of squared lengths of all internal branches of both trees, resulting in a final distance that ranges from 0 to 1.

Value

Returns a normalized distance, with 0 indicating identical trees and 1 indicating maximal difference.

If the trees have no leaves in common, the function will return 1.

Note

Note that this function requires the input dendrograms to be labeled alike (ex. leaf labeled abc in dend1 represents the same species as leaf labeled abc in dend2). Labels can easily be modified using dendrapply.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

References

Robinson, D.F. and Foulds, L.R. *Comparison of phylogenetic trees.* Mathematical Biosciences, 1987. **53**(1–2): 131–147.

Kuhner, M. K. and Felsenstein, J. *Simulation comparison of phylogeny algorithms under equal and unequal evolutionary rates.* Molecular Biology and Evolution, 1994. **11**: 459–468.

Examples

```
# making some toy dendrograms
set.seed(123)
dm1 <- as.dist(matrix(runif(64, 0.5, 5), ncol=8))
dm2 <- as.dist(matrix(runif(64, 0.5, 5), ncol=8))

tree1 <- as.dendrogram(hclust(dm1))
tree2 <- as.dendrogram(hclust(dm2))

# get KF distance
PhyloDistance(tree1, tree2, Method="KF")
```

PhyloDistance-RFDist *Robinson-Foulds Distance*

Description

Calculate RF distance between two unrooted phylogenies.

Details

This function is called as part of PhyloDistance and calculates Robinson-Foulds distance between two unrooted phylogenies. Each dendrogram is first pruned to only internal branches implying a partition in the shared leaf set; trivial partitions (where one leaf set contains 1 or 0 leaves) are ignored. The total score is calculated as the number of unique partitions divided by the total number of partitions in both trees. Setting RawScore=TRUE will instead return a vector with (P_{shared}, P_1, P_2), corresponding to the shared partitions and partitions in the first and second trees (respectively).

This algorithm incorporates some optimizations from Pattengale et al. (2007) to improve computation time of the original fast RF algorithm detailed in Day (1985).

Value

Returns a normalized distance, with 0 indicating identical trees and 1 indicating maximal difference. Note that branch lengths are not considered, so two trees with different branch lengths may return a distance of 0.

If `RawScore=TRUE`, returns a named length 3 vector with the first entry the number of unique partitions, and the subsequent entries the number of partitions for each tree.

If the trees have no leaves in common, the function will return `1` if `RawScore=FALSE`, and `c(0, NA, NA)` if `TRUE`.

Note

Note that this function requires the input dendrograms to be labeled alike (ex. leaf labeled `abc` in `dend1` represents the same species as leaf labeled `abc` in `dend2`). Labels can easily be modified using `dendrapply`.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

References

Robinson, D.F. and Foulds, L.R. *Comparison of phylogenetic trees.* Mathematical Biosciences, 1987. **53**(1–2): 131–147.

Day, William H.E. *Optimal algorithms for comparing trees with labeled leaves.* Journal of classification, 1985. **2**(1): 7-28.

Pattengale, N.D., Gottlieb, E.J., and Moret, B.M. *Efficiently computing the Robinson-Foulds metric.* Journal of computational biology, 2007. **14**(6): 724-735.

Examples

```
# making some toy dendrograms
set.seed(123)
dm1 <- as.dist(matrix(runif(64, 0.5, 5), ncol=8))
dm2 <- as.dist(matrix(runif(64, 0.5, 5), ncol=8))

tree1 <- as.dendrogram(hclust(dm1))
tree2 <- as.dendrogram(hclust(dm2))

# get RF distance
PhyloDistance(tree1, tree2, Method="RF")

# get number of unique splits per tree
PhyloDistance(tree1, tree2, Method="RF", RawScore=TRUE)
```

`plot.EvoWeb` *Plot predictions in a EvoWeb object*

Description

EvoWeb objects can be returned from `predict.EvoWeaver`.

This function plots the predictions in the object using a force-directed embedding of connections in the adjacency matrix.

This function is being targetting for additional functionality in later releases.

Usage

```
## S3 method for class 'EvoWeb'
plot(x, NumSims=10,
      Gravity=0.05, Coulomb=0.1, Connection=5,
      MoveRate=0.25, Cutoff=0.2, ColorPalette=topo.colors,
      Verbose=TRUE, ...)
```

Arguments

`x`	A EvoWeb object. See `EvoWeb`
`NumSims`	Integer; Number of iterations to run the model for.
`Gravity`	Numeric; Strength of Gravity force. See 'Details'.
`Coulomb`	Numeric; Strength of Coulomb force. See 'Details'.
`Connection`	Numeric; Strength of Connective force. See 'Details'.
`MoveRate`	Numeric; Controls how far each point moves in each iteration.
`Cutoff`	Numeric; Cutoff value; if `abs(val)` < `Cutoff`, that Connection is shrunk to zero.
`ColorPalette`	Character; Color palette for graphing. Valid inputs are any palette available in `palette.pals()`. See `palette` for more info.
`Verbose`	Logical; Determines if status messages and progress bars should be displayed while running.
`...`	Additional parameters for consistency with generic.

Details

This function plots the `EvoWeb` object using a force-directed embedding. This embedding has three force components:

- Gravity Force: Attractive force pulling nodes towards `(0,0)`
- Coulomb Force: Repulsive force pushing close nodes away from each other
- Connective Force: Tries to push node connections to equal corresponding values in the adjacency matrix

The parameters in the function are sufficient to get an embedding, though users are welcome to try to tune them for a better visualization. This function is meant to aid with visualization of the adjacency matrix, not for concrete analyses of clusters.

The function included in this release is early stage. Next release cycle will update this function with an updated version of this algorithm to improve plotting, visualization, and runtime.

Value

No return value; creates a plot in the graphics window.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

See Also

predict.EvoWeaver

EvoWeb

Examples

```
exData <- get(data("ExampleStreptomycesData"))
ew <- EvoWeaver(exData$Genes)

# Subset isn't necessary but is faster for a working example
# Same w/ method='ExtantJaccard'
evoweb <- predict(ew, Method='ExtantJaccard', Subset=1:50)

plot(evoweb)
```

---

predict.EvoWeaver *Make predictions with EvoWeaver objects*

Description

This S3 method predicts pairwise functional associations between gene groups encoded in a EvoWeaver object. This returns an object of type EvoWeb, which is essentially an adjacency matrix with some extra S3 methods to make printing cleaner.

Usage

```
## S3 method for class 'EvoWeaver'
predict(object, Method='Ensemble',
         Subset=NULL,
         MySpeciesTree=SpeciesTree(object, Verbose=Verbose),
         PretrainedModel="KEGG",
         NoPrediction=FALSE,
         ReturnDataFrame=TRUE,
         Verbose=interactive(),
         CombinePVal=TRUE,
         useDNA=FALSE,...)
```

Arguments

object	A EvoWeaver object
Method	Character; Method(s) to use for prediction. This can be a character vector with multiple entries for predicting using multiple methods. See 'Details' for more information.

`Subset` Either a vector or a `2xN` matrix representing the subset of data to predict on.

If a vector, prediction proceeds for all possible pairs of elements specified in the vector (either by name, for character vector, or by index, for numeric vector). For example, `subset=1:3` will predict for pairs `(1,2), (1,3), (2,3)`.

If a matrix, subset is interpreted as a matrix of pairs, where each row of the matrix specifies a pair to evaluate. These can also be specifed by name (character) or by index (numeric).

`subset=rbind(c(1,2),c(1,3),c(2,3))` produces equivalent functionality to `subset=1:3`.

`MySpeciesTree` Object of class `dendrogram` representing the phylogenetic relationship of all genomes in the dataset. Required for `Method=c('RPContextTree', 'GLDistance', 'CorrGL', 'MoransI', 'Behdenna')`. `'Behdenna'` requires a rooted, bifurcating tree (other values of `Method` can handle arbitrary trees). Note that `EvoWeaver` can automatically infer a species tree if initialized with `dendrogram` objects.

`PretrainedModel`

A pretrained model for use with ensemble predictions. The default value is `"KEGG"`, corresponding to a built-in ensemble model trained on the KEGG MODULE database. Alternative values allowed are `"CORUM"`, for a built-in ensemble model trained on the CORUM database, or any user-trained model. See the examples for how to train an ensemble method to pass to `PretrainedModel`.

Has no effect if `Method != 'Ensemble'`.

`NoPrediction` Logical; determines if data should be returned prior to making prediction for `Method='Ensemble'`.

If `TRUE`, this will instead return a data.frame object with predictions from each algorithm for each pair. This dataframe is typically used to train an ensemble model.

If `FALSE`, EvoWeaver will return predictions for each pair (using user model if provided or a built-in otherwise).

`ReturnDataFrame`

Logical; Determines if the function should return a `data.frame` object or a list of `EvoWeb` objects. Setting this parameter to `FALSE` is not recommended.

`Verbose` Logical; Determines if status messages and progress bars should be displayed while running.

`CombinePVal` Logical; Determines if scores and p-values should be combined or returned as separate values.

`useDNA` Logical; Determines whether to interpret sequences as DNA or AA (only used for Sequence Level methods, see Details).

`...` Additional parameters for other predictors and consistency with generic.

Details

`predict.EvoWeaver` wraps several methods to create an easy interface for multiple prediction types. `Method='Ensemble'` is the default value, but each of the component analyses can also be accessed. Common arguments to `Method` include:

- `'Ensemble'`: Ensemble prediction combining individual coevolutionary predictors. See `Note` below.
- `'PhylogeneticProfiling'`: All Phylogenetic Profiling Algorithms used in the EvoWeaver manuscript.

- 'PhylogeneticStructure': All EvoWeaver Phylogenetic Structure Methods
- 'GeneOrganization': All EvoWeaver Gene Organization Methods
- 'SequenceLevel': All EvoWeaver Sequence Level Methods used in the EvoWeaver manuscript.

Additional information and references for each prediction algorithm can be found at the following pages:

- EvoWeaver Phylogenetic Profiling Methods
- EvoWeaver Phylogenetic Structure Methods
- EvoWeaver Gene Organization Methods
- EvoWeaver Sequence Level Methods

The standard return type is a data.frame object with one column per predictor and an additional two columns specifying the genes in each pair. If ReturnDataFrame=FALSE, this returns a EvoWeb object. See EvoWeb for more information. Use of this parameter is discouraged.

By default, EvoWeaver weights scores by their p-value to correct for spurious correlations. The returned scores are raw_score*(1-p_value). If CombinePVal=FALSE, EvoWeaver will instead return the raw score and the p-value separately. The resulting data.frame will have one column for the raw score (denoted METHOD.score) and one column for the p-value (denoted METHOD.pval). **Note: p-values are recorded as (1-p)**. Not all methods support returning p-values separately from the score; in this case, only a METHOD.score column will be returned.

Different methods require different types of input. The constructor EvoWeaver will notify the user which methods are runnable with the given data. Method Ensemble automatically selects the methods that can be run with the given input data.

See EvoWeaver for more information on input data types.

Complete listing of all supported methods (asterisk denotes a method used in Ensemble, if possible):

- * 'GLMI': MI of G/L profiles
- * 'GLDistance': Score-based method based on distance between inferred ancestral Gain/Loss events
- * 'PAJaccard': Centered Jaccard distance of P/A profiles with conserved clades collapsed
- * 'PAOverlap': Conservation of ancestral states based on P/A profiles
- * 'RPMirrorTree': MirrorTree using Random Projection for dimensionality reduction
- * 'RPContextTree': MirrorTree with Random Projection correcting for species tree and P/A conservation
- * 'GeneDistance': Co-localization analysis
- * 'MoransI': Co-localization analysis using Moran's I for phylogenetic correction and significance
- * 'OrientationMI': Mutual Information of Gene Relative Orientation
- * 'GeneVector': Correlation of distribution of sequence level residues following Zhao et al. (2022)
- * 'SequenceInfo': Mutual information of sites in multiple sequence alignment
- 'ExtantJaccard': Jaccard Index of Presence/Absence (P/A) profiles at extant leaves
- 'Hamming': Hamming similarity of P/A profiles
- 'PAPV': 1-p_value of P/A profiles
- 'ProfDCA': Direct Coupling Analysis of P/A profiles
- 'Behdenna': Analysis of Gain/Loss events following Behdenna et al. (2016)
- 'CorrGL': Correlation of ancestral Gain/Loss events

Value

If ReturnDataFrame=TRUE, returns a data.frame object where each row corresponds to a single prediction for a pair of gene groups. The first two columns contain the gene group identifiers for each pair, and the remaining columns contain each prediction.

If ReturnDataFrame=FALSE, the return type is a list of EvoWeb objects. See EvoWeb for more info.

Note

If NumCores is set to NULL, EvoWeaver will use one less core than is detected, or one core if detectCores() cannot detect the number of available cores. This is because of a potential issue where the R session can consume all available cores and then lose the ability to fork processes, with the only solution to restart the entire R session.

If ReturnDataFrame=FALSE and CombinePVal=FALSE, the resulting EvoWeb objects will contain values of type 'complex'. For each value, the real part denotes the raw score, and the imaginary part denotes 1-p, with p the p-value.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

See Also

EvoWeaver

EvoWeb

EvoWeaver Phylogenetic Profiling Predictors

EvoWeaver Phylogenetic Structure Predictors

EvoWeaver Gene Organization Predictors

EvoWeaver Sequence Level Predictors

Examples

```
##############
## Prediction with built-in model and data
###############

set.seed(555L)
exData <- get(data("ExampleStreptomycesData"))
ew <- EvoWeaver(exData$Genes[1:50], MySpeciesTree=exData$Tree)

# Subset isn't necessary but is faster for a working example
evoweb1 <- predict(ew, Subset=1:2)

# print out results as an adjacency matrix
if(interactive()) print(evoweb1)

###############
## Training own ensemble model
###############

datavals <- evoweb1[,-c(1,2,10)]
actual_values <- sample(c(0,1), nrow(datavals), replace=TRUE)
# This example just picks random numbers
```

```
# ***Do not do this for your own models***

# Make sure the actual values correspond to the right pairs!
datavals[,'y'] <- actual_values
myModel <- glm(y~., datavals[,-c(1,2)], family='binomial')

testEvoWeaverObject <- EvoWeaver(exData$Genes[51:60], MySpeciesTree=exData$Tree)
evoweb2 <- predict(testEvoWeaverObject,
                    PretrainedModel=myModel)

# Print result as a data.frame of pairwise scores
if(interactive()) print(evoweb2)
```

RandForest *Classification and Regression with Random Forests*

Description

RandForest implements a version of Breiman's random forest algorithm for classification and regression.

Usage

```
RandForest(formula, data, subset, verbose=interactive(),
           weights, na.action,
           method='rf.fit',
           rf.mode=c('auto', 'classification', 'regression'),
           contrasts=NULL, ...)

## S3 method for class 'RandForest'
predict(object, newdata=NULL,
                na.action=na.pass, ...)

## Called internally by `RandForest`
RandForest.fit(x, y=NULL,
    verbose=interactive(), ntree=10,
    mtry=floor(sqrt(ncol(x))),
    weights=NULL, replace=TRUE,
    sampsize=if(replace) nrow(x) else ceiling(0.632*nrow(x)),
    nodesize=1L, max_depth=NULL,
    method=NULL,
    terms=NULL,...)
```

Arguments

formula	an object of class "formula" (or one that can be coerced to that class): a symbolic description of the model to be fitted. See lm for more details.
data	An optional data frame, list, or environment (or object coercible by as.data.frame to a data frame) containing the variables in the model. If not found in data, the variables are taken from environment(formula), typically the environment from which RandForest is called.

subset	an optional vector specifying a subset of observations to be used in the fitting process.
weights	an optional vector of weights to be used in the fitting process. Should be NULL or a numeric vector.
na.action	a function which indicates what should happen when the data contain NAs. Currently experimental.
method	currently unused.
rf.mode	one of "auto", "classification", "regression" (or an unambiguous abbreviation), specifying the type of trees to build. If rf.mode="auto", the mode is inferred based on the type of the response variable.
contrasts	currently experimental; see lm.
...	further arguments passed to RandForest.fit.
object	an object of class 'RandForest' for prediction.
newdata	new data to predict on, typically provided as a data.frame object.
verbose	Logical; Determines if status messages should be displayed while running.
ntree	number of decision trees to grow.
mtry	number of variables to try at each split.
replace	logical; should data be sampled with replacement during training?
sampsize	number of datapoints to sample for training each component decision tree.
nodesize	number of datapoints to stop classification (see "Details")
max_depth	maximum depth of component decision trees.
x	used internally by RandForest.fit
y	used internally by RandForest.fit
terms	used internally by RandForest.fit

Details

RandForest implements a version of Breiman's original algorithm to train a random forest model for classification or regression. Random forests are comprised of a set of decision trees, each of which is trained on a subset of the available data. These trees are individually worse predictors than a single decision tree trained on the entire dataset. However, averaging predictions across the ensemble of trees forms a model that is often more accurate than single decision trees while being less susceptible to overfitting.

Random forests can either be trained for classification or regression. Classification forests are comprised of trees that assign inputs to a specific class. The output prediction is a vector comprised of the proportion of trees in the forest that assigned the datapoint to each available class. Regresssion forests are comprised of trees that assign each datapoint to a single continuous value, and the output prediction is comprised of the mean prediction across all component trees. When rf.mode="auto", the random forest will be trained in classification mode for response of type "factor", and in regression mode for response of type "numeric".

Several parameters exist to tune the behavior of random forests. The ntree argument controls how many decision trees are trained. At each decision point, the decision trees consider a random subset of available variables–the number of variables to sample is controlled by mtry. Each decision tree only sees a subset of available data to reduce its risk of overfitting. This subset is comprised of sampsize datapoints, which are sampled with or without replacement according to the replace argument.

Finally, the default behavior is to grow decision trees until they have fully classified all the data they see for training. However, this may lead to overfitting. Decision trees can be limited to smaller sizes by specifying the `max_depth` or `nodesize` arguments. `max_depth` refers to the depth of the decision tree. Setting this value to `n` means that every path from the root node to a leaf node will be at most length `n`. `nodesize` can be used to instead stop growing trees based on the size of the data to be partitioned at each decision tree node. If `nodesize=n`, then if a decision point receives less than `n` samples, it will stop trying to further split the data.

Classification forests are trained by maximizing the Gini Gain at each interior node. Split points are determined with exhaustive search for small data sizes, or simulated annealing for larger sizes. Regression forests are trained by maximizing the decrease in sum of squared error (SSE) if all points in each partition are assigned their mean output value. Nodes stop classification when either no partition improves the maximization metric (Gini Gain or decrease in SSE) or when the criteria specified by `nodesize` / `max_depth` are met.

Some of the arguments provided are for consistency with the base `lm` function. Use caution changing any values referred to as "Experimental" above. `NA` values may cause unintended behavior.

Value

An object of class `'RandForest'`, which itself contains a number of objects of class `'DecisionTree'` which can be used for prediction with `predict.RandForest`

Note

Generating a single decision tree model is possible by setting `ntree=1` and `sampsize=nrow(data)`. `'DecisionTree'` objects do not currently support prediction.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

References

Breiman, L. (2001), *Random Forests*, Machine Learning 45(1), 5-32.

See Also

DecisionTree class

Examples

```
set.seed(199L)
n_samp <- 100L
AA <- rnorm(n_samp, mean=1, sd=5)
BB <- rnorm(n_samp, mean=2, sd=3)
CC <- rgamma(n_samp, shape=1, rate=2)
err <- rnorm(n_samp, sd=0.5)
y <- AA + BB + 2*CC + err

d <- data.frame(AA,BB,CC,y)
train_i <- 1:90
test_i <- 91:100
train_data <- d[train_i,]
test_data <- d[test_i,]
```

```
rf_regr <- RandForest(y~., data=train_data, rf.mode="regression", max_depth=5L)
if(interactive()){
  # Visualize one of the decision trees
  plot(rf_regr[[1]])
}

## classification
y1 <- y < -5
y2 <- y < 0 & y >= -5
y3 <- y < 5 & y >= 0
y4 <- y >= 5
y_cl <- rep(0L, length(y))
y[y1] <- 1L
y[y2] <- 2L
y[y3] <- 3L
y[y4] <- 4L
d$y <- as.factor(y)
train_data <- d[train_i,]
test_data <- d[test_i,]

rf_classif <- RandForest(y~., data=train_data, rf.mode="classification", max_depth=5L)
if(interactive()){
  # Visualize one of the decision trees for classification
  plot(rf_classif[[1]])
}
```

SequenceSimilarity *Return a numeric value that represents the similarity between two aligned sequences as determined by a provided subsitution matrix.*

Description

Takes in a DNAStringSet or AAStringSet representing a pairwise alignment and a subsitution matrix such as those present in PFASUM, and return a numeric value representing sequence similarity as defined by the substitution matrix.

Usage

```
SequenceSimilarity(Seqs,
                   SubMat,
                   penalizeGapLetter = TRUE,
                   includeTerminalGaps = TRUE,
                   allowNegative = TRUE)
```

Arguments

Seqs	A DNAStringSet or AAStringSet of length 2.
SubMat	A named matrix representing a substitution matrix. If left "NULL" and "Seqs" is a AAStringSet, the 40th "PFASUM" matrix is used. If left "NULL" and "Seqs" is a DNAStringSet, a matrix with only the diagonal filled with "1"'s is used.
penalizeGapLetter	A logical indicating whether or not to penalize Gap-Letter matches. Defaults to "TRUE".

includeTerminalGaps
: A logical indicating whether or not to penalize terminal matches. Defaults to "TRUE".

allowNegative
: A logical indicating whether or not allow negative scores. Defaults to "TRUE". If "FALSE" scores that are returned as less than zero are converted to zero.

Details

Takes in a DNAStringSet or AAStringSet representing a pairwise alignment and a subsitution matrix such as those present in PFASUM, and return a numeric value representing sequence similarity as defined by the substitution matrix.

Value

Returns a single numeric.

Author(s)

Erik Wright <ESWRIGHT@pitt.edu> Nicholas Cooley <npc19@pitt.edu>

See Also

AlignSeqs, AlignProfiles, AlignTranslation, DistanceMatrix

Examples

```
db <- system.file("extdata", "Bacteria_175seqs.sqlite", package = "DECIPHER")
dna <- SearchDB(db, remove = "all")
alignedDNA <- AlignSeqs(dna[1:2])

DNAPlaceholder <- diag(15)
dimnames(DNAPlaceholder) <- list(DNA_ALPHABET[1:15],
                                 DNA_ALPHABET[1:15])

SequenceSimilarity(Seqs = alignedDNA,
                   SubMat = DNAPlaceholder,
                   includeTerminalGaps = TRUE,
                   penalizeGapLetter = TRUE,
                   allowNegative = TRUE)
```

---

simMat *Similarity Matrices*

Description

The simMat object is an internally utilized class that provides similar functionality to the dist object, but with matrix-like accessors.

Like dist, this object stores values as a vector, reducing memory by making use of assumed symmetry. simMat currently only supports numeric data types.

Usage

```
## Create a blank sym object
simMat(VALUE, nelem, NAMES=NULL, DIAG=FALSE)

## S3 method for class 'vector'
as.simMat(x, NAMES=NULL, DIAG=TRUE, ...)

## S3 method for class 'matrix'
as.simMat(x, ...)

## S3 method for class 'simMat'
print(x, ...)

## S3 method for class 'simMat'
as.matrix(x, ...)

## S3 method for class 'simMat'
as.data.frame(x, ...)

## S3 method for class 'simMat'
Diag(x, ...)

## S3 replacement method for class 'simMat'
Diag(x) <- value
```

Arguments

VALUE	Numeric (or NA_real_) indicating placeholder values. A vector of values can be provided for this function if desired.
nelem	Integer; number of elements represented in the matrix. This corresponds to the number of rows and columns of the object, so setting nelem=10 would produce a 10x10 matrix.
NAMES	Character (Optional); names for each row/column. If provided, this should be a character vector of length equal to nelem.
DIAG	Logical; Determines if the diagonal is included in the data. If FALSE, the constructor generates 1s for the diagonal.
x	For print and Diag, the "simMat" object to print. For as.vector or as.matrix, the vector or matrix (respectively). Note that as.matrix expects a symmetric matrix–providing a non-symmetric matrix will take only the upper triangle and produce a warning.
value	Numeric; value(s) to replace diagonal with.
...	Additional parameters provided for consistency with generic.

Details

The simMat object has a very similar format to dist objects, but with a few notable changes:

- simMat objects have streamlined print and show methods to make displaying large matrices better. print accepts an additional argument n corresponding to the maximum number of rows/columns to print before truncating.
- simMat objects support matrix-style get/set operations like s[1,] or s[1,3:5]

- simMat objects allow any values on the diagonal, rather than just zeros as in dist objects.
- simMat objects support conversion to matrices and data.frame objects
- simMat objects implement get/set Diag() methods. Note usage of capitalized Diag; this is to avoid conflicts and weirdness with using base diag.

See the examples for details on using these features.

The number of elements printed when calling print or show on a simMat object is determined by the "SynExtend.simMat" option.

Value

simMat and as.simMat return an object of class "simMat". Internally, the object stores the upper triangle of the matrix similar to how dist stores objects.

The object has the following attributes (besides "class" equal to "simMat"):

nrow	the number of rows in the matrix implied by the vector
NAMES	the names of the rows/columns

as.matrix(s) returns the equivalent matrix to a "simMat" object.

as.data.frame(s) returns a data.frame object corresponding to pairwise similarities.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

Examples

```
## Creating a blank simMat object initialized to zeros
s <- simMat(0, nelem=20)
s

## Print out 5 rows instead of 10
print(s, n=5)

## Create a simMat object with 5 entries from a vector
dimn <- 5
vec <- 1:(dimn*(dimn-1) / 2)
s1 <- as.simMat(vec, DIAG=FALSE)
s1

## Here we include the diagonal
vec <- 1:(dimn*(dimn+1) / 2)
s2 <- as.simMat(vec, DIAG=TRUE)
s2

## Subsetting
s2[1,]
s2[1,3:4]
# all entries except first row
s2[-1,]
# all combos not including 1
s2[-1,-1]

## Replace values (automatically recycled)
```

```
s2[1,] <- 10
s2

## Get/set diagonal
Diag(s1)
Diag(s1) <- 5
s1
```

SquaregffBy *Convert a GRanges object to a DataFrame*

Description

A DataFrame representing a portion of the data contained in a gff file / GRanges object.

Usage

```
SquaregffBy(gff_object,
            collect_by = c("gene",
                           "pseudogene",
                           "ncRNA_gene",
                           "tRNA_gene"),
            verbose = FALSE)
```

Arguments

gff_object	A GRanges object produced by rtracklayer's import.
collect_by	A character vector containing feature types to collect from the gff object.
verbose	A logical indicating whether to print out messages and a progress bar.

Details

Given a DataFrame with a Range column containing an IRangesList, return a data.frame the length of the unlisted IRangesList. Each row represents the coordinates of a subfeature as described by the contents of the IRangesList. Key and SubKey columns can be used to trace child features back to their parents.

Value

A DataFrame containing excerpts from the attributes fields of a gff file, organized by the parent feature.

Author(s)

Nicholas Cooley <Nicholas.Cooley@ul.ie>

See Also

- SquaregffBy
- FindSynteny
- NucleotideOverlap

Examples

```
library(rtracklayer)
grange_obj <- import(con = system.file("extdata",
                                       "GCF_023585725.1_ASM2358572v1_genomic.gff.gz",
                                       package = "SynExtend"),
                     format = "gff")
ImportedGFF <- SquaregffBy(gff_object = grange_obj,
                           verbose = TRUE)
```

`subset.dendrogram` *Subsetting dendrogram objects*

Description

Subsets dendrogram objects based on leaf labels. Subsetting can either be by leaves to keep, or leaves to remove.

NOTE: This man page is specifically for `subset.dendogram`, see `?base::subset` for the generic `subset` function defined for vectors, matrices, and data frames.

Usage

```
## S3 method for class 'dendrogram'
subset(x, subset, invert=FALSE, ...)
```

Arguments

`x`	An object of class `'dendogram'`
`subset`	Character; A vector of labels to keep (see `invert`).
`invert`	Logical; If `TRUE`, subsets to the leaves *not* in `subset`.
`...`	Additional arguments for consistency with generic.

Value

An object of class `'dendrogram'` corresponding to the subset of the tree.

Note

If none of the labels specified in the `subset` argument appear in the tree (or if all do when `invert=TRUE`), a warning is thrown and an empty object of class `'dendrogram'` is returned.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

See Also

`subset`

Examples

```
d <- as.dendrogram(hclust(dist(USArrests), "ave"))

# Show original dendrogram
plot(d)

# Subset to first 10 labels
d1 <- subset(d, labels(d)[1:10])
plot(d1)

# Subset d1 to all except the first 2 labels
d2 <- subset(d1, labels(d1)[1:2], invert=TRUE)
plot(d2)
```

SummarizePairs *Summarizing Linked Feature Pairs from a LinkedPairs Object*

Description

A function for generating a comprehensive summary of linked genomic feature pairs from an object of class LinkedPairs. For each pair, the function computes alignment statistics (percent identity, alignment score), k-mer distance, syntenic block context, and an approximate background-corrected score, returning results as an object of class PairSummaries.

Usage

```
SummarizePairs(SynExtendObject,
               DataBase01,
               DefaultTranslationTable = "11",
               KmerSize = 5,
               Verbose = FALSE,
               ShowPlot = FALSE,
               Processors = 1,
               Storage = 2,
               Anchors = c("enforce", "infer", "ignore"),
               ...)
```

Arguments

SynExtendObject
: An object of class LinkedPairs, typically produced by NucleotideOverlap. Each populated cell in the upper triangle of this matrix object describes the syntenic hits that link a pair of genomes.

DataBase01
: Either a connection object to a DECIPHER-compatible SQLite database, or a character string giving the path to such a database on disk. The database must contain nucleotide sequences (indexed by identifier) that correspond to the genomes referenced in SynExtendObject. If a path string is provided, the RSQLite package must be installed. Amino acid sequences will be read from the database if an AAs table is present; otherwise they will be computed on the fly and written back to the database.

`DefaultTranslationTable`
: A character string of length 1 specifying the NCBI genetic code identifier to use when translating coding sequences whose translation table is not recorded in the `GeneCalls` attribute of `SynExtendObject`. Defaults to `"11"` (the bacterial, archaeal, and plant plastid code). Must be a valid identifier accepted by `Biostrings::getGeneticCode`.

`KmerSize`
: A positive integer specifying the k-mer width used to compute nucleotide frequency profiles for each feature. These profiles are used to derive a k-mer distance (`KDist`) between each pair of features. Must be less than 10. Defaults to `5`.

`Verbose`
: Logical indicating whether to display a progress bar and print the elapsed time upon completion. Defaults to `FALSE`.

`ShowPlot`
: Logical. Reserved for future use. Currently has no effect. Defaults to `FALSE`.

`Processors`
: A positive whole-number integer, or `NULL`, specifying the number of processors to use for parallel operations passed to `AlignPairs`. When `NULL`, the number of available cores is detected automatically via `DECIPHER`. Defaults to `1`.

`Storage`
: A positive numeric value specifying the maximum amount of memory (in gigabytes) that the internal data pool may occupy before cached sequence data for earlier genomes is evicted. Increase this value when analysing many large genomes to avoid redundant database queries; decrease it on memory-constrained systems. Defaults to `2`.

`Anchors`
: A character string controlling how alignment anchors are constructed when calling `AlignPairs`. Must be one of `"enforce"` (force terminal anchors at the ends of each sequence, the default), `"infer"` (infer anchors from syntenic hit positions; not yet implemented), or `"ignore"` (pass no anchors). Partial matching is supported.

`...`
: Additional named arguments reserved for future use. Currently not passed to any downstream function.

Details

For each occupied cell in the upper triangle of `SynExtendObject`, the function:

1. Retrieves or constructs nucleotide and amino acid feature sequences from `DataBase01` using the `GeneCalls` stored as an attribute of `SynExtendObject`.
2. Computes k-mer frequency profiles (`KmerSize`-mers) for all nucleotide features and derives a normalised Euclidean k-mer distance for each pair.
3. Aligns each pair with `AlignPairs`, optionally anchored according to the `Anchors` argument.
4. Computes local and approximate global percent identity (PID) and alignment score from the `AlignPairs` output.
5. Computes a per-pair approximate background alignment score (`Delta_Background`) by subtracting an expected score derived from the residue composition of the two genomes from the observed approximate global alignment score.
6. Assigns syntenic block identifiers (`Block_UID`) via `BlockByRank`, with a globally unique offset applied across all pairwise comparisons so that block IDs do not collide between genome pairs.
7. Computes a hit-weighted positional consensus score (`Consensus`) from the syntenic hit positions stored in the lower triangle of `SynExtendObject`.

If Amino acid sequences for a given identifier are not already present they are written back to the database under the table name `AAs` and reused on subsequent iterations, reducing redundant computation for large multi-genome comparisons.

Memory consumption is managed by evicting the least-recently-needed genome data from the internal pool when its size exceeds `Storage` gigabytes. In practice this is unlikely to matter for most users unless you are pushing the bounds of storage and memory on consumer hardware.

Value

A `data.frame` of class `c("data.frame", "PairSummaries")` with one row per linked feature pair. Columns are:

p1 Character. Name of the query feature.

p2 Character. Name of the subject feature.

Consensus Numeric. A positional consensus score (between 0 and 1) summarising the agreement of syntenic hit anchors across the pair; 1 indicates complete disagreement and 0 indicates perfect agreement.

p1featurelength Integer. Nucleotide length of the query feature.

p2featurelength Integer. Nucleotide length of the subject feature.

blocksize Integer. Number of feature pairs belonging to the same syntenic block as this pair.

KDist Numeric. Normalised Euclidean distance between the k-mer frequency profiles of the two features.

TotalMatch Integer. Total number of exact nucleotide overlap bases recorded for this pair in the `LinkedPairs` object.

MaxMatch Integer. Maximum single k-mer hit size recorded for this pair.

UniqueMatches Integer. Number of unique k-mer matches recorded for this pair.

Local_PID Numeric. Fraction of matched positions within the aligned region (local percent identity).

Local_Score Numeric. Alignment score normalised by the alignment length.

Approx_Global_PID Numeric. Fraction of matched positions normalised by the length of the longer feature (approximate global percent identity).

Approx_Global_Score Numeric. Alignment score normalised by the length of the longer feature.

Alignment Character. Alphabet used for alignment: `"AA"` for amino acid or `"NT"` for nucleotide.

Block_UID Integer. A globally unique block identifier grouping syntenic feature pairs that co-occur in the same syntenic block. Pairs not assigned to a multi-pair block receive a unique singleton ID.

Delta_Background Numeric. The difference between the approximate global alignment score and the expected score under a background model derived from genome-wide residue composition; higher values indicate greater similarity above background.

The returned object retains the following attributes:

GeneCalls The named list of gene calls inherited from `SynExtendObject`.

KmerSize The value of `KmerSize` used during the run.

DefaultTranslationTable The value of `DefaultTranslationTable` used during the run.

Author(s)

Nicholas Cooley <Nicholas.Cooley@ul.ie>

See Also

- NucleotideOverlap
- FindSynteny

Examples

```
library(DBI)
data("linked_features")
tmp01 <- system.file("extdata",
                     "example_db.sqlite",
                     package = "SynExtend")
tmp02 <- tempfile()
file.copy(from = tmp01,
          to = tmp02)

drv <- dbDriver("SQLite")
conn01 <- dbConnect(drv = drv,
                    tmp02)

x <- SummarizePairs(SynExtendObject = linked_features,
                    DataBase01 = conn01)
```

SuperTree *Create a Species Tree from Gene Trees*

Description

Given a set of unrooted gene trees, creates a species tree. While this function also works for rooted gene trees, the resulting root may not be accurately placed.

Usage

```
SuperTree(myDendList, NAMEFUN=NULL, Verbose=TRUE, ...)
```

Arguments

myDendList	List of dendrogram objects, where each entry is an unrooted gene tree.
NAMEFUN	Optional function to apply to each leaf to convert gene tree leaf labels into species names. This function should take as input a character vector and return a character vector of the same size. By default equals NULL, indicating that gene tree leaves are already labeled with species identifiers. See details for more information.
Verbose	Logical; Determines if status messages and progress bars should be displayed while running.
...	Further arguments passed to Treeline

Details

This implementation follows the Weighted ASTRID algorithm for estimating a species tree from a set of unrooted gene trees. Input gene trees are not required to have identical species sets, as the algorithm can handle missing entries in gene trees. The algorithm essentially works by averaging the Cophenetic distance matrices of all gene trees, then constructing a neighbor-joining tree from the resulting distance matrix. See the original paper linked in the references section for more information.

If two species never appear together in a gene tree, their distance cannot be estimated in the algorithm and will thus be missing. `SuperTree` handles this by imputing the value using the distances available with data-interpolating empirical orthogonal functions (DINEOF). This approach has relatively high accuracy even up to high levels of missingness. Eigenvector calculation speed is improved using a Lanczos algorithm for matrix compression.

`SuperTree` allows an optional argument called `NAMEFUN` to apply a renaming step to leaf labels. Gene trees as constructed by other functions in `SynExtend` (ex. `DisjointSet`) often include other information aside from species name when labeling genes, but `SuperTree` requires that leaf nodes of the gene tree are labeled with just an identifier corresponding to which species/genome each leaf is from. Duplicate values are allowed. See the examples section for more details on what this looks like and how to handle it.

Value

A `dendrogram` object corresponding to the species tree constructed from input gene trees.

Author(s)

Aidan Lakshman <ahl27@pitt.edu>

References

Liu B., Warnow T. *Weighted ASTRID: fast and accurate species trees from weighted internode distances*. Algorithms Mol Biol, 2023 Jul 19; **18**(1):6.

Taylor, M.H., Losch, M., Wenzel, M. and Schröter, J. *On the sensitivity of field reconstruction and prediction using empirical orthogonal functions derived from gappy data*. Journal of Climate, 2013. **26**(22): 9194-9205.

See Also

`Treeline`, `SuperTreeEx`

Examples

```
# Loads a list of dendrograms
# each is a gene tree from Streptomyces genomes
data("SuperTreeEx", package="SynExtend")

# Notice that the labels of the tree are in #_#_# format
# See the man page for SuperTreeEx for more info
labs <- labels(exData[[1]])
if(interactive()) print(labs)

# The first number corresponds to the species,
# so we need to trim the rest in each leaf label
namefun <- function(x) gsub("([0-9A-Za-z]*)_.*", "\\1", x)
```

```
namefun(labs) # trims to just first number

# This function replaces gene identifiers with species identifiers
# we pass it to NAMEFUN
# Note NAMEFUN should take in a character vector and return a character vector
tree <- SuperTree(exData, NAMEFUN=namefun)
```

SuperTreeEx *Example Dendrograms*

Description

A set of four dendrograms for use in SuperTree examples.

Usage

```
data("SuperTreeEx")
```

Format

A list with four elements, where each is a object of type dendrogram corresponding to a gene tree constructed from a set of 301 *Streptomyces* genomes. Each leaf node is labeled in the form A_B_C, where A is a number identifying the genome, B is a number identifying the contig, and C is a number identifying the gene. Altogether, each label uniquely identifies a gene.

Examples

```
data(SuperTreeEx, package="SynExtend")
```

---

syn *Example genecall data*

Description

An object of class Synteny

Usage

```
data("syn", package = "SynExtend")
```

Format

An object of class Synteny.

Details

An object of class Synteny generated by the extdata.R script contained in SynExtend's inst/scripts folder. This object contains succinct data for runnable function examples.

Examples

```
data("syn", package = "SynExtend")
```

ToFeatureSpace *Tabulating Features Linked by Syntenic Hits*

Description

A function for concisely tabulating where genomic features are connected by syntenic hits.

Usage

```
ToFeatureSpace(hit_blocks,
               qranges,
               qstrands,
               sranges,
               sstrands)
```

Arguments

hit_blocks	A list of data.frames created by calling split on as.data.frame of the lower triangle of the NucleotideOverlap object. Splitting on the interaction of the QueryGene and SubjectGene columns.
qranges	An IRangesList.
qstrands	A vector of integers, 0 for a feature in the negative strand, 1 for the positive strand.
sranges	An IRangesList.
sstrands	A vector of integers, 0 for a feature in the negative strand, 1 for the positive strand.

Details

Builds a list of 4-row x n-column matrices from the information provided by NucleotideOverlap.

Value

A list of matrices to be passed to AlignPairs as anchors.

Author(s)

Nicholas Cooley <Nicholas.Cooley@ul.ie>

See Also

- FindSynteny
- NucleotideOverlap
- SummarizePairs
- AlignPairs

Examples

```
data("genecalls", package = "SynExtend")
data("syn", package = "SynExtend")

Links <- NucleotideOverlap(SyntenyObject = syn,
                           GeneCalls = genecalls,
                           LimitIndex = FALSE,
                           Verbose = TRUE)
```

Index