Package 'universalmotif'

November 17, 2026

Title A Toolkit for Sequence Motif Analysis

Version 1.31.65

URL https://bioconductor.org/packages/universalmotif/

BugReports https://github.com/bjmt/universalmotif/issues

Description A comprehensive toolkit for working with sequence motifs in R. Imports and exports most common motif formats (JASPAR, MEME, HOMER, TRANSFAC, CIS-BP, UNIPROBE) and interoperates with the other Bioconductor motif packages. Analysis functions cover de novo motif discovery, motif-vs-motif comparison and clustering, P-value calculation, sequence scanning, enrichment against shuffled or composition-matched backgrounds, positional bias testing, and pairwise motif co-occurrence. Also includes utilities for sequence shuffling, motif trimming, higher-order representations, ground-truth simulation by motif implantation, and logo-plotting functionality.

Depends R (>= 3.5.0)

License GPL-3

Encoding UTF-8

Imports methods, stats, utils, parallel, MASS, ggplot2 (>= 4.0.0), yaml, IRanges, Rcpp, Biostrings, BiocGenerics, S4Vectors, rlang, grid, grDevices, MatrixGenerics

Suggests spelling, knitr, bookdown, TFBSTools, rmarkdown, MotifDb, testthat, BiocParallel, seqLogo, motifStack, dplyr, ape, ggtree, processx, ggseqlogo, cowplot, GenomicRanges, GenomeInfoDb, ggbio, rtracklayer, BSgenome, BSgenome.Athaliana.TAIR.TAIR9

Enhances PWMEnrich, rGADEM

LinkingTo Rcpp, RcppThread

VignetteBuilder knitr

biocViews MotifAnnotation, MotifDiscovery, DataImport, GeneRegulation

RoxygenNote 7.3.2

Roxygen list(markdown = TRUE, old_usage = TRUE)

Language en-GB

Collate 'RcppExports.R' 'add_multifreq.R' 'compare_motifs.R'
'compare_motifs_lite.R' 'compare_report.R'
'universalmotif-class.R' 'convert_motifs.R' 'convert_type.R'
'create_motif.R' 'create_sequences.R' 'data.R' 'dedup_hits.R'
'enrich_motifs.R' 'enrich_motifs_lite.R' 'filter_motifs.R'
'geom_motif.R' 'get_bkg.R' 'implant_motifs.R' 'make_DBscores.R'
'mask_complexity.R' 'match_bkg.R' 'merge_motifs.R'
'merge_motifs_lite.R' 'merge_similar.R' 'merge_similar_lite.R'
'motif_coocc.R' 'motif_finder.R' 'motif_peaks.R'
'motif_proximity.R' 'motif_pvalue.R' 'motif_rc.R'
'motif_tree.R' 'motif_tree_lite.R' 'plot_comparison.R'
'plot_coocc.R' 'plot_enrichment.R' 'plot_hits.R'
'plot_score_distribution.R' 'read_cisbp.R' 'read_homer.R'
'read_jaspar.R' 'read_matrix.R' 'read_meme.R' 'read_motifs.R'
'read_transfac.R' 'read_uniprobe.R' 'run_meme.R'
'sample_sites.R' 'scan_sequences.R' 'scan_sequences_lite.R'
'sequence_complexity.R' 'shuffle_motifs.R'
'shuffle_sequences.R' 'switch_alph.R' 'trim_cwm.R'
'trim_motifs.R' 'universalmotif-methods.R' 'universalmotif.R'
'universalmotif_df.R' 'utils-internal.R' 'utils-motif.R'
'utils-sequence.R' 'view_logo.R' 'view_motifs.R'
'view_motifs_lite.R' 'write_homer.R' 'write_jaspar.R'
'write_matrix.R' 'write_meme.R' 'write_motifs.R'
'write_transfac.R' 'zzz.R'

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

git_branch devel

git_last_commit c23644b

git_last_commit_date 2026-11-11

Repository Bioconductor 3.24

Date/Publication 2026-11-16

Author Benjamin Jean-Marie Tremblay [aut, cre] (ORCID:
<https://orcid.org/0000-0002-7441-2951>),
Sabrina Nystrom [ctb] (ORCID: <https://orcid.org/0000-0003-1000-1579>)

Maintainer Benjamin Jean-Marie Tremblay <benjamin.tremblay@uwaterloo.ca>

Contents

add_multifreq 4
ArabidopsisMotif 6
ArabidopsisPromoters 6
compare_motifs 7
compare_motifs_lite 10
convert_motifs 13
convert_type 16
create_motif 20
create_sequences 24
dedup_hits 26
enrich_motifs 27
enrich_motifs_lite 30

examplemotif . . . 32
examplemotif2 . . . 32
filter_motifs . . . 33
fontDFroboto . . . 34
geom_motif . . . 35
get_bkg . . . 37
implant_motifs . . . 39
JASPAR2018_CORE_DBSCORES . . . 41
make_DBscores . . . 41
mask_complexity . . . 43
match_bkg . . . 45
merge_motifs . . . 48
merge_motifs_lite . . . 50
merge_similar . . . 51
merge_similar_lite . . . 53
motif_coocc . . . 55
motif_finder . . . 57
motif_peaks . . . 60
motif_proximity . . . 62
motif_pvalue . . . 66
motif_rc . . . 69
motif_tree . . . 70
motif_tree_lite . . . 73
plot_comparison . . . 75
plot_coocc . . . 77
plot_enrichment . . . 78
plot_hits . . . 80
plot_match_bkg . . . 82
plot_motif_peaks . . . 83
plot_motif_proximity . . . 84
plot_score_distribution . . . 85
read_cisbp . . . 86
read_homer . . . 87
read_jaspar . . . 88
read_matrix . . . 89
read_meme . . . 90
read_motifs . . . 92
read_transfac . . . 93
read_uniprobe . . . 94
reexports . . . 95
run_meme . . . 95
sample_sites . . . 98
scan_sequences . . . 98
scan_sequences_lite . . . 102
sequence_complexity . . . 104
shuffle_motifs . . . 107
shuffle_sequences . . . 108
switch_alph . . . 110
tidy-motifs . . . 110
trim_cwm . . . 112
trim_motifs . . . 113
universalmotif-class . . . 114

universalmotif-pkg . . . 118
utilities . . . 118
utils-motif . . . 119
utils-sequence . . . 125
view_logo . . . 129
view_motifs . . . 130
view_motifs_lite . . . 133
write_homer . . . 136
write_jaspar . . . 137
write_matrix . . . 138
write_meme . . . 139
write_motifs . . . 141
write_transfac . . . 142

Index **143**

`add_multifreq` *Add multi-letter information to a motif.*

Description

If the original sequences are available for a particular motif, then they can be used to generate higher-order PPM matrices. See the "Motif import, export, and manipulation" vignette for more information.

Usage

```
add_multifreq(motif, sequences, add.k = 2:3, RC = FALSE,
  threshold = 0.001, threshold.type = "pvalue", motifs.perseq = 1,
  add.bkg = FALSE)
```

Arguments

`motif`	See `convert_motifs()` for acceptable formats. If the motif is not a universalmotif motif, then it will be converted.
`sequences`	`XStringSet` The alphabet must match that of the motif. If these sequences are all the same length as the motif, then they are all used to generate the multi-freq matrices. Otherwise `scan_sequences()` is first run to find the best sequence stretches within these.
`add.k`	`numeric(1)` The k-let lengths to add.
`RC`	`logical(1)` If `TRUE`, check reverse complement of the input sequences. Only available for DNA/RNA.
`threshold`	`numeric(1)` See details.
`threshold.type`	`character(1)` One of `c('pvalue', 'qvalue', 'logodds', 'logodds.abs')`. See details.
`motifs.perseq`	`numeric(1)` If `scan_sequences()` is run, then this indicates how many hits from each sequence is to be used.
`add.bkg`	`logical(1)` Indicate whether to add corresponding higher order background information to the motif. Can sometimes be detrimental when the input consists of few short sequences, which can increase the likelihood of adding zero or near-zero probabilities.

Details

See `scan_sequences()` for more info on scanning parameters.

At each position in the motif, then the probability of each k-let covering from the initial position to `ncol - 1` is calculated. Only positions within the motif are considered: this means that the final k-let probability matrix will have `ncol - 1` fewer columns. Calculating k-let probabilities for the missing columns would be trivial however, as you would only need the background frequencies. Since these would not be useful for `scan_sequences()` though, they are not calculated.

Currently `add_multifreq()` does not try to stay faithful to the default motif matrix when generating multifreq matrices. This means that if the sequences used for training are completely different from the actual motif, the multifreq matrices will be as well. However this is only really a problem if you supply `add_multifreq()` with a set of sequences of the same length as the motif. In this case `add_multifreq()` is forced to create the multifreq matrices from these sequences. Otherwise `add_multifreq()` will scan the input sequences for the motif and use the best matches to construct the multifreq matrices.

This 'multifreq' representation is only really useful within the **universalmotif** environment. Despite this, if you wish it can be preserved in text using `write_motifs()`.

A note on motif size:

The number of rows for each k-let matrix is `n^k`, with `n` being the number of letters in the alphabet being used. This means that the size of the k-let matrix can become quite large as k increases. For example, if one were to wish to represent a DNA motif of length 10 as a 10-let, this would require a matrix with 1,048,576 rows (though at this point if what you want is to search for exact sequence matches, the motif format itself is not very useful).

Value

A universalmotif object with filled `multifreq` slot. The `bkg` slot is also expanded with corresponding higher order probabilities if `add.bkg = TRUE`.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

`scan_sequences()`, `convert_motifs()`, `write_motifs()`

Examples

```
sequences <- create_sequences(seqlen = 10)
motif <- create_motif()
motif.trained <- add_multifreq(motif, sequences, add.k = 2:4)
## peek at the 2-let matrix:
motif.trained["multifreq"]$`2`
```

ArabidopsisMotif *Arabidopsis motif in* universalmotif *format.*

Description

Arabidopsis motif trained from ArabidopsisPromoters using MEME version 4. This motif was generated at the command line using the following command: meme promoters.fa -revcomp -nmotifs 3 -mod anr -

Usage

```
ArabidopsisMotif
```

Format

universalmotif

Examples

```
data(ArabidopsisMotif)
ArabidopsisMotif
```

ArabidopsisPromoters *Arabidopsis promoters as a* DNAStringSet.

Description

50 Arabidopsis promoters, each 1000 bases long. See the "Sequence manipulation and scanning" vignette for an example workflow describing extracting promoter sequences.

Usage

```
ArabidopsisPromoters
```

Format

DNAStringSet

Examples

```
data(ArabidopsisPromoters)
ArabidopsisPromoters
```

compare_motifs *Compare motifs.*

Description

Compare motifs using one of the several available metrics. See the "Motif comparisons and P-values" vignette for detailed information. Note: for regular DNA/RNA motif comparison using PCC, see `compare_motifs_lite()` for a faster and simpler alternative.

Usage

```
compare_motifs(motifs, compare.to, db.scores, use.freq = 1,
  use.type = "PPM", method = "PCC", tryRC = TRUE, min.overlap = 6,
  min.mean.ic = 0.25, min.position.ic = 0, relative_entropy = FALSE,
  normalise.scores = FALSE, max.p = 0.01, max.e = 10, nthreads = 1,
  score.strat = "a.mean", output.report, output.report.max.print = 10)
```

Arguments

`motifs`	See `convert_motifs()` for acceptable motif formats.
`compare.to`	`numeric` If missing, compares all motifs to all other motifs. Otherwise compares all motifs to the specified motif(s).
`db.scores`	`data.frame` or `DataFrame`. See `details`.
`use.freq`	`numeric(1)`. For comparing the `multifreq` slot.
`use.type`	`character(1)` One of `'PPM'` and `'ICM'`. The latter allows for taking into account the background frequencies if `relative_entropy = TRUE`. Note that `'ICM'` is not allowed when `method = c("ALLR", "ALLR_LL")`.
`method`	`character(1)` One of PCC, EUCL, SW, KL, ALLR, BHAT, HELL, SEUCL, MAN, ALLR_LL, WEUCL, WPCC. See details.
`tryRC`	`logical(1)` Try the reverse complement of the motifs as well, report the best score.
`min.overlap`	`numeric(1)` Minimum overlap required when aligning the motifs. Setting this to a number higher then the width of the motifs will not allow any overhangs. Can also be a number between 0 and 1, representing the minimum fraction that the motifs must overlap.
`min.mean.ic`	`numeric(1)` Minimum mean information content between the two motifs for an alignment to be scored. This helps prevent scoring alignments between low information content regions of two motifs. Note that this can result in some comparisons failing if no alignment passes the mean IC threshold. Use `average_ic()` to filter out low IC motifs to get around this if you want to avoid getting `NA`s in your output.
`min.position.ic`	`numeric(1)` Minimum information content required between individual alignment positions for it to be counted in the final alignment score. It is recommended to use this together with `normalise.scores = TRUE`, as this will help punish scores resulting from only a fraction of an alignment.
`relative_entropy`	`logical(1)` Change the ICM calculation affecting `min.position.ic` and `min.mean.ic`. See `convert_type()`.

`normalise.scores`
: `logical(1)` Favour alignments which leave fewer unaligned positions, as well as alignments between motifs of similar length. Similarity scores are multiplied by the ratio of aligned positions to the total number of positions in the larger motif, and the inverse for distance scores.

`max.p`
: `numeric(1)` Maximum P-value allowed in reporting matches. Only used if `compare.to` is set.

`max.e`
: `numeric(1)` Maximum E-value allowed in reporting matches. Only used if `compare.to` is set. The E-value is the P-value multiplied by the number of input motifs times two.

`nthreads`
: `numeric(1)` Run `compare_motifs()` in parallel with `nthreads` threads. `nthreads = 0` uses all available threads.

`score.strat`
: `character(1)` How to handle column scores calculated from motif alignments. "sum": add up all scores. "a.mean": take the arithmetic mean. "g.mean": take the geometric mean. "median": take the median. "wa.mean", "wg.mean": weighted arithmetic/geometric mean. "fzt": Fisher Z-transform. Weights are the total information content shared between aligned columns.

`output.report`
: `character(1)` Path (ending in `.html`) for a self-contained HTML report of the comparison. The report has a summary header (the call and key settings), a sortable results table (query, target, alignment offset, strand, score, and P/E-values), and a query-vs-target sequence-logo alignment for each top match. Requires `compare.to` to be set (for all-vs-all output see `motif_tree()`) and the `knitr` and `rmarkdown` packages (plus pandoc).

`output.report.max.print`
: `numeric(1)` Maximum number of top matches to show in the report (table and figures).

Details

Available metrics:

The following metrics are available:

- Euclidean distance (`EUCL`) (Choi et al. 2004)
- Weighted Euclidean distance (`WEUCL`)
- Kullback-Leibler divergence (`KL`) (Kullback and Leibler 1951; Roepcke et al. 2005)
- Hellinger distance (`HELL`) (Hellinger 1909)
- Squared Euclidean distance (`SEUCL`)
- Manhattan distance (`MAN`)
- Pearson correlation coefficient (`PCC`)
- Weighted Pearson correlation coefficient (`WPCC`)
- Sandelin-Wasserman similarity (`SW`), or sum of squared distances (Sandelin and Wasserman 2004)
- Average log-likelihood ratio (`ALLR`) (Wang and Stormo 2003)
- Lower limit ALLR (`ALLR_LL`) (Mahony et al. 2007)
- Bhattacharyya coefficient (`BHAT`) (Bhattacharyya 1943)

Comparisons are calculated between two motifs at a time. All possible alignments are scored, and the best score is reported. In an alignment scores are calculated individually between columns. How those scores are combined to generate the final alignment scores depends on `score.strat`.

See the "Motif comparisons and P-values" vignette for a description of the various metrics. Note that `PCC`, `WPCC`, `SW`, `ALLR`, `ALLR_LL` and `BHAT` are similarities; higher values mean more similar motifs. For the remaining metrics, values closer to zero represent more similar motifs.

Small pseudocounts are automatically added when one of the following methods is used: `KL`, `ALLR`, `ALLR_LL`, `IS`. This is avoid zeros in the calculations.

Calculating P-values:

To note regarding p-values: P-values are pre-computed using the `make_DBscores()` function. If not given, then uses a set of internal precomputed P-values from the JASPAR2018 CORE motifs. These precalculated scores are dependent on the length of the motifs being compared. This takes into account that comparing small motifs with larger motifs leads to higher scores, since the probability of finding a higher scoring alignment is higher.

The default P-values have been precalculated for regular DNA motifs. They are of little use for motifs with a different number of alphabet letters (or even the `multifreq` slot).

Value

`matrix` if `compare.to` is missing; `DataFrame` otherwise. For the latter, function args are stored in the `metadata` slot.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Bhattacharyya A (1943). "On a measure of divergence between two statistical populations defined by their probability distributions." *Bulletin of the Calcutta Mathematical Society*, **35**, 99-109.

Choi I, Kwon J, Kim S (2004). "Local feature frequency profile: a method to measure structural similarity in proteins." *PNAS*, **101**, 3797-3802.

Hellinger E (1909). "Neue Begrundung der Theorie quadratischer Formen von unendlichvielen Veranderlichen." *Journal fur die reine und angewandte Mathematik*, **136**, 210-271.

Khan A, Fornes O, Stigliani A, Gheorghe M, Castro-Mondragon JA, van der Lee R, Bessy A, Cheneby J, Kulkarni SR, Tan G, Baranasic D, Arenillas DJ, Sandelin A, Vandepoele K, Lenhard B, Ballester B, Wasserman WW, Parcy F, Mathelier A (2018). "JASPAR 2018: update of the open-access database of transcription factor binding profiles and its web framework." *Nucleic Acids Research*, **46**, D260-D266.

Kullback S, Leibler RA (1951). "On information and sufficiency." *The Annals of Mathematical Statistics*, **22**, 79-86.

Itakura F, Saito S (1968). "Analysis synthesis telephony based on the maximum likelihood method." In *6th International Congress on Acoustics*, C-17.

Mahony S, Auron PE, Benos PV (2007). "DNA Familial Binding Profiles Made Easy: Comparison of Various Motif Alignment and Clustering Strategies." *PLoS Computational Biology*, **3**.

Pietrokovski S (1996). "Searching databases of conserved sequence regions by aligning protein multiple-alignments." *Nucleic Acids Research*, **24**, 3836-3845.

Roepcke S, Grossmann S, Rahmann S, Vingron M (2005). "T-Reg Comparator: an analysis tool for the comparison of position weight matrices." *Nucleic Acids Research*, **33**, W438-W441.

Sandelin A, Wasserman WW (2004). "Constrained binding site diversity within families of transcription factors enhances pattern discovery bioinformatics." *Journal of Molecular Biology*, **338**, 207-215.

Wang T, Stormo GD (2003). "Combining phylogenetic data with co-regulated genes to identify motifs." *Bioinformatics*, **19**, 2369-2380.

See Also

convert_motifs(), motif_tree(), view_motifs(), make_DBscores(), compare_motifs_lite(), view_motifs_lite()

Examples

```
motif1 <- create_motif(name = "1")
motif2 <- create_motif(name = "2")
motif1vs2 <- compare_motifs(c(motif1, motif2), method = "PCC")
## To get a dist object:
as.dist(1 - motif1vs2)

motif3 <- create_motif(name = "3")
motif4 <- create_motif(name = "4")
motifs <- c(motif1, motif2, motif3, motif4)
## Compare motif "2" to all the other motifs:
if (R.Version()$arch != "i386") {
compare_motifs(motifs, compare.to = 2, max.p = 1, max.e = Inf)
}

## If you are working with a large list of motifs and the mean.min.ic
## option is not set to zero, you may get a number of failed comparisons
## due to low IC. To filter the list of motifs to avoid these, use
## the average_ic() function to remove motifs with low average IC:
## Not run:
library(MotifDb)
motifs <- convert_motifs(MotifDb)[1:100]
compare_motifs(motifs)
#> Warning in compare_motifs(motifs) :
#>   Some comparisons failed due to low IC
motifs <- motifs[average_ic(motifs) > 0.5]
compare_motifs(motifs)

## End(Not run)
```

---

compare_motifs_lite *Faster minimalist motif comparison.*

Description

compare_motifs_lite() is a deliberately pared-down counterpart to compare_motifs(). It uses the per-column Pearson correlation as its similarity metric, computes significance from a TOMTOM-style null PMF (either empirical, from the target database, or parametric, via a Dirichlet-Multinomial over a K=5 simplex grid), applies both Bonferroni and Benjamini-Hochberg adjustment, and returns either a long-format data.frame of the significant pairs or a square score / p-value / q-value matrix.

Usage

```
compare_motifs_lite(motifs, compare.to = NULL, qvalue = 0.1,
  min.overlap = 5L, RC = TRUE, null = c("empirical", "parametric"),
  bkg = NULL, matrix.out = c("score", "pvalue", "qvalue"), nthreads = 1,
  output.report, output.report.max.print = 10)
```

Arguments

motifs	See convert_motifs() for accepted motif formats. DNA or RNA only.
compare.to	integer, character, or NULL. When NULL (default), all-pairs mode: every motif is compared against every other and a square matrix is returned (see matrix.out). When integer/character, the named/indexed motifs are treated as queries and the *entire* input is treated as the target database; a long-format data.frame of significant matches is returned.
qvalue	numeric(1). q-value cutoff for the long-format output. Default 0.1. Ignored in matrix mode.
min.overlap	integer(1). Minimum overlap length (in columns) for a candidate alignment. Default 5.
RC	logical(1). If TRUE (default), also score query motifs in reverse-complement orientation and report the best of the two.
null	character(1). Null-PMF mode: "empirical" (default) or "parametric". See Details.
bkg	numeric(4) or NULL. Background base frequencies (A, C, G, T). If NULL, uniform c(.25, .25, .25, .25) is used. Critical for "parametric" (sets the Dirichlet prior); affects only pseudocount smoothing for "empirical".
matrix.out	character(1). In matrix mode, which metric to return: "score" (mean PCC; default), "pvalue", or "qvalue". Ignored in long-format mode.
nthreads	numeric(1). Number of threads. nthreads = 0 uses all available threads.
output.report	character(1). Path (ending in .html) for a self-contained HTML report of the matches: a summary header, a sortable results table (query, target, offset, strand, overlap, score, P/q-value), and a query-vs-target sequence-logo alignment per top match. Requires compare.to to be set and the knitr and rmarkdown packages (plus pandoc). Only meaningful in long-format mode.
output.report.max.print	numeric(1). Maximum number of top matches to show in the report (table and figures). Default 10.

Details

Use compare_motifs() when you need any of the broader features: alternative similarity metrics (EUCL, ALLR, KL, ...), score-aggregation strategies, IC-based filters, multifreq scoring, ICM/PPM dispatch, relative-entropy mode, or the JASPAR-derived db.scores lookup-table p-value path.

P-values are computed in two modes:

null = "empirical" **(default)** Per-query column, score against every target-database column, histogram into 201 PCC bins, convolve to overlap length L. Cost per query is O(N x T) where N is the number of target motifs and T their mean width. Best for databases with more than a few hundred total columns.

`null = "parametric"` Enumerate the 56 compositions of A/C/G/T on a K=5 simplex grid, weight each by a Dirichlet-Multinomial PMF under alpha = 4 * bkg, score against the query column. Per-query cost is constant in database size. Best for small databases or when histogram sparsity would distort the empirical estimate.

The reported `score` is the mean PCC over the overlap (always in $[-1, 1]$), not the raw sum.

Value

- **Matrix mode** (`compare.to = NULL`): a square numeric matrix indexed by motif name, holding the value selected by `matrix.out`.
- **Long-format mode** (`compare.to` provided): a `data.frame` with columns `subject`, `subject.i`, `target`, `target.i`, `offset`, `strand`, `overlap`, `score`, `subject.consensus`, `target.consensus`, `pvalue`, `qvalue`, filtered to `qvalue <= qvalue` and sorted by q-value ascending. Coordinates are 1-based; `offset` is the query column where target column 1 aligns.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Gupta S, Stamatoyannopoulos JA, Bailey TL, Noble WS (2007). "Quantifying similarity between motifs." *Genome Biology*, **8**(2), R24.

See Also

`compare_motifs()`, `scan_sequences_lite()`, `view_motifs_lite()`

Other lite motif functions: `enrich_motifs_lite()`, `merge_motifs_lite()`, `merge_similar_lite()`, `motif_tree_lite()`, `scan_sequences_lite()`, `view_motifs_lite()`

Examples

```
library(universalmotif)
set.seed(1)
motifs <- lapply(1:6, function(i)
  create_motif(paste(sample(c("A","C","G","T"), 8, replace = TRUE),
                     collapse = ""), name = paste0("M", i)))
## Matrix of mean PCC scores
m <- compare_motifs_lite(motifs)
round(m, 2)
## Long-format hits at q <= 0.5 with motif 1 as the query
compare_motifs_lite(motifs, compare.to = 1, qvalue = 0.5)
```

convert_motifs *Convert motif class.*

Description

Allows for easy transfer of motif information between different classes as defined by other Bioconductor packages. This function is also used by nearly all other functions in this package, so any motifs of a compatible class can be used without needing to be converted beforehand.

Usage

```
convert_motifs(motifs, class = "universalmotif-universalmotif")

## S4 method for signature 'AsIs'
convert_motifs(motifs,
  class = "universalmotif-universalmotif")

## S4 method for signature 'list'
convert_motifs(motifs,
  class = "universalmotif-universalmotif")

## S4 method for signature 'universalmotif'
convert_motifs(motifs,
  class = "universalmotif-universalmotif")

## S4 method for signature 'MotifList'
convert_motifs(motifs,
  class = "universalmotif-universalmotif")

## S4 method for signature 'TFFMFirst'
convert_motifs(motifs,
  class = "universalmotif-universalmotif")

## S4 method for signature 'PFMatrix'
convert_motifs(motifs,
  class = "universalmotif-universalmotif")

## S4 method for signature 'PWMatrix'
convert_motifs(motifs,
  class = "universalmotif-universalmotif")

## S4 method for signature 'ICMatrix'
convert_motifs(motifs,
  class = "universalmotif-universalmotif")

## S4 method for signature 'XMatrixList'
convert_motifs(motifs,
  class = "universalmotif-universalmotif")

## S4 method for signature 'pwm'
convert_motifs(motifs,
```

```
  class = "universalmotif-universalmotif")

## S4 method for signature 'pcm'
convert_motifs(motifs,
  class = "universalmotif-universalmotif")

## S4 method for signature 'pfm'
convert_motifs(motifs,
  class = "universalmotif-universalmotif")

## S4 method for signature 'PWM'
convert_motifs(motifs,
  class = "universalmotif-universalmotif")

## S4 method for signature 'Motif'
convert_motifs(motifs,
  class = "universalmotif-universalmotif")

## S4 method for signature 'matrix'
convert_motifs(motifs,
  class = "universalmotif-universalmotif")
```

Arguments

`motifs`	Single motif object or list. See details.
`class`	`character(1)` Desired motif class. Input as 'package-class'. If left empty, defaults to 'universalmotif-universalmotif'. (See details.)

Details

External classes may not represent gap definitions or higher-order matrices. These losses produce a warning on export. For complete preservation of the original object, use `saveRDS()`.

Input:

The following packge-class combinations can be used as input:

- MotifDb-MotifList
- TFBSTools-PFMatrix
- TFBSTools-PWMatrix
- TFBSTools-ICMatrix
- TFBSTools-PFMatrixList
- TFBSTools-PWMatrixList
- TFBSTools-ICMatrixList
- TFBSTools-TFFMFirst
- seqLogo-pwm
- motifStack-pcm
- motifStack-pfm
- PWMEnrich-PWM
- motifRG-Motif
- universalmotif-universalmotif
- matrix

Output:

The following package-class combinations can be output:

- MotifDb-MotifList
- TFBSTools-PFMatrix
- TFBSTools-PWMatrix
- TFBSTools-ICMatrix
- TFBSTools-TFFMFirst
- seqLogo-pwm
- motifStack-pcm
- motifStack-pfm
- PWMEnrich-PWM
- Biostrings-PWM (`type = 'log2prob'`)
- rGADEM-motif
- universalmotif-universalmotif (the default, no need to specify this)

Note: MotifDb-MotifList output was a later addition to `convert_motifs()`. As a result, to stay consistent with previous behaviour most functions will always convert MotifDb-MotifList objects to a list of `universalmotif` motifs, even if other formats would be simply returned as is (e.g. for other formats, `filter_motifs()` will return the input format; for MotifDb-MotifList, a list of `universalmotif` objects will be returned).

Value

Single motif object or list.

Methods (by class)

- `convert_motifs(AsIs)`: Generate an error as a reminder to run `to_list()` instead of using the column from `to_df()` directly.
- `convert_motifs(list)`: Convert a list of motifs.
- `convert_motifs(universalmotif)`: Convert a universalmotif object.
- `convert_motifs(MotifList)`: Convert MotifList motifs. (**MotifDb**)
- `convert_motifs(TFFMFirst)`: Convert TFFMFirst motifs. (**TFBSTools**)
- `convert_motifs(PFMatrix)`: Convert PFMatrix motifs. (**TFBSTools**)
- `convert_motifs(PWMatrix)`: Convert PWMatrix motifs. (**TFBSTools**)
- `convert_motifs(ICMatrix)`: Convert ICMatrix motifs. (**TFBSTools**)
- `convert_motifs(XMatrixList)`: Convert XMatrixList motifs. (**TFBSTools**)
- `convert_motifs(pwm)`: Convert pwm motifs. (**seqLogo**)
- `convert_motifs(pcm)`: Convert pcm motifs. (**motifStack**)
- `convert_motifs(pfm)`: Convert pfm motifs. (**motifStack**)
- `convert_motifs(PWM)`: Convert PWM motifs. (**PWMEnrich**)
- `convert_motifs(Motif)`: Convert Motif motifs. (**motifRG**)
- `convert_motifs(matrix)`: Create motif from matrices.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Bembom O (2018). *seqLogo: Sequence logos for DNA sequence alignments*. R package version 1.46.0.

Droit A, Gottardo R, Robertson G, Li L (2014). *rGADEM: de novo motif discovery*. R package version 2.28.0.

Mercier E, Gottardo R (2014). *MotIV: Motif Identification and Validation*. R package version 1.36.0.

Ou J, Wolfe SA, Brodsky MH, Zhu LJ (2018). "motifStack for the analysis of transcription factor binding site evolution." *Nature Methods*, **15**, 8-9. doi: 10.1038/nmeth.4555.

Shannon P, Richards M (2018). *MotifDb: An Annotated Collection of Protein-DNA Binding Sequence Motifs*. R package version 1.22.0.

Stojnic R, Diez D (2015). *PWMEnrich: PWM enrichment analysis*. R package version 4.16.0.

Tan G, Lenhard B (2016). "TFBSTools: an R/Bioconductor package for transcription factor binding site analysis." *Bioinformatics*, **32**, 1555-1556. doi: 10.1093/bioinformatics/btw024.

Yao Z (2012). *motifRG: A package for discriminative motif discovery, designed for high throughput sequencing dataset*. R package version 1.24.0.

Examples

```
# Convert from universalmotif:
jaspar <- read_jaspar(system.file("extdata", "jaspar.txt",
                                  package = "universalmotif"))
if (requireNamespace("motifStack", quietly = TRUE)) {
  jaspar.motifstack.pfm <- convert_motifs(jaspar, "motifStack-pfm")
}

# Convert from another class to universalmotif:
if (requireNamespace("TFBSTools", quietly = TRUE)) {
library(TFBSTools)
data(MA0003.2)
motif <- convert_motifs(MA0003.2)

# Convert from another class to another class
if (requireNamespace("PWMEnrich", quietly = TRUE)) {
  motif <- convert_motifs(MA0003.2, "PWMEnrich-PWM")
}

# The 'convert_motifs' function is embedded in the rest of the universalmotif
# functions: non-universalmotif class motifs can be used
MA0003.2.trimmed <- trim_motifs(MA0003.2)
# Note: if the motif object going in has information that the
# 'universalmotif' class can't hold, it will be lost
}
```

`convert_type` *Convert universalmotif type.*

Description

Switch between position count matrix (PCM), position probability matrix (PPM), position weight matrix (PWM), information count matrix (ICM), and contribution weight matrix (CWM) types. See the "Introduction to sequence motifs" vignette for details. Please also note that type conversion occurs implicitly throughout the `universalmotif` package, so there is generally no need to perform this manual conversion. Also please be aware that the message concerning pseudocount-adjusting motifs can be disabled via `options(pseudocount.warning=FALSE)`.

Usage

```
convert_type(motifs, type, pseudocount, nsize_correction = FALSE,
  relative_entropy = FALSE)
```

Arguments

`motifs`	See `convert_motifs()` for acceptable formats.
`type`	`character(1)` One of `c('PCM', 'PPM', 'PWM', 'ICM')`. `'CWM'` is only valid as the source type of a conversion; `convert_type(other, "CWM")` errors because contribution scores cannot be recovered from a probabilistic matrix.
`pseudocount`	`numeric(1)` Correction to be applied to prevent `-Inf` from appearing in PWM matrices. If missing, the pseudocount stored in the universalmotif 'pseudocount' slot will be used.
`nsize_correction`	`logical(1)` If true, the ICM at each position will be corrected to account for small sample sizes. Only used if `relative_entropy = FALSE`.
`relative_entropy`	`logical(1)` If true, the ICM will be calculated as relative entropy. See details.

Details

PCM:

Position count matrix (PCM), also known as position frequency matrix (PFM). For n sequences from which the motif was built, each position is represented by the numbers of each letter at that position. In theory all positions should have sums equal to n, but not all databases are this consistent. If converting from another type to PCM, column sums will be equal to the 'nsites' slot. If empty, 100 is used.

PPM:

Position probability matrix (PPM), also known as position frequency matrix (PFM). At each position, the probability of individual letters is calculated by dividing the count for that letter by the total sum of counts at that position (`letter_count / position_total`). As a result, each position will sum to 1. Letters with counts of 0 will thus have a probability of 0, which can be undesirable when searching for motifs in a set of sequences. To avoid this a pseudocount can be added (`(letter_count + pseudocount) / (position_total + pseudocount)`).

PWM:

Position weight matrix (PWM; Stormo et al. (1982)), also known as position-specific weight matrix (PSWM), position-specific scoring matrix (PSSM), or log-odds matrix. At each position, each letter is represented by it's log-likelihood (`log2(letter_probability / background_probility)`), which is normalized using the background letter frequencies. A PWM matrix is constructed from a PPM. If any position has 0-probability letters to which pseudocounts were not added, then the final log-likelihood of these letters will be `-Inf`.

ICM:

Information content matrix (ICM; Schneider and Stephens 1990). An ICM is a PPM where each letter probability is multiplied by the total information content at that position. The information content of each position is determined as: `totalIC - Hi`, where the total information totalIC

`totalIC <- log2(alphabet_length)`, and the Shannon entropy (Shannon 1948) for a specific position (Hi)
`Hi <- -sum(sapply(alphabet_frequencies, function(x) x * log(2))`.

As a result, the total sum or height of each position is representative of it's sequence conservation, measured in the unit 'bits', which is a unit of energy (Schneider 1991; see https://fr-s-schneider.ncifcrf.gov/logorecommendations.html for more information). However not all programs will calculate information content the same. Some will 'correct' the total information content at each position using a correction factor as described by Schneider et al. (1986). This correction can applied by setting `nsize_correction = TRUE`, however it will only be applied if the 'nsites' slot is not empty. This is done using `TFBSTools:::schneider_correction` (Tan and Lenhard 2016). As such, converting from an ICM to which some form of correction has been applied will result in a PCM/PPM/PWM with slight inaccuracies.

Another method of calculating information content is calculating the relative entropy, also known as Kullback-Leibler divergence (Kullback and Leibler 1951). This accounts for background frequencies, which can be useful for genomes with a heavy imbalance in letter frequencies. For each position, the individual letter frequencies are calculated as `letter_freq * log2(letter_freq / bkg_freq)`. When calculating information content using Shannon entropy, the maximum content for each position will always be `log2(alphabet_length)`. This does not hold for information content calculated as relative entropy. Please note that conversion from ICM assumes the information content was *not* calculated as relative entropy.

CWM:

Contribution weight matrix (CWM), as produced by attribution methods such as TF-MoDISco (Shrikumar et al. 2018) and DeepLIFT (Shrikumar, Greenside, and Kundaje 2017). Each cell holds a signed real-valued contribution score: how much the given letter at the given position drives a model's prediction. Unlike the four probabilistic types above, CWMs are not normalised (columns do not sum to 1 or to `nsites`), values may be negative (the model penalises that letter at that position), and there is no canonical maximum height. CWM is therefore only ever a source type for `convert_type()`: `convert_type(cwm, "PPM")` column-normalises absolute values via the TF-MoDISco-lite convention `ppm[i,j] = |cwm[i,j]| / sum_i |cwm[i,j]|` (columns whose absolute column sum is zero collapse to a uniform distribution). `convert_type(cwm, "PWM" | "ICM" | "PCM")` routes through `CWM -> PPM` first and then applies the standard `PPM -> *` step, so `CWM -> PWM` produces a real log-odds matrix (`log2(ppm / bkg)`), not a re-labelled CWM. The reverse direction (`PPM`, `PWM`, `ICM`, or `PCM -> CWM`) errors with a clear message: contribution scores cannot be inferred from a probability distribution, so a CWM must originate from an attribution method (read via `read_meme()` with `CWM = TRUE`, or built with `create_motif(matrix, type = "CWM")`).

Value

See `convert_motifs()` for possible output motif objects.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Kullback S, Leibler RA (1951). "On information and sufficiency." *The Annals of Mathematical Statistics*, **22**, 79-86.

Nishida K, Frith MC, Nakai K (2009). "Pseudocounts for transcription factor binding sites." *Nucleic Acids Research*, **37**, 939-944.

Schneider TD, Stormo GD, Gold L, Ehrenfeucht A (1986). "Information content of binding sites on nucleotide sequences." *Journal of Molecular Biology*, **188**, 415-431.

Schneider TD, Stephens RM (1990). "Sequence Logos: A New Way to Display Consensus Sequences." *Nucleic Acids Research*, **18**, 6097-6100.

Schneider TD (1991). "Theory of Molecular Machines. II. Energy Dissipation from Molecular Machines." *Journal of Theoretical Biology*, **148**, 125-137.

Shannon CE (1948). "A Mathematical Theory of Communication." *Bell System Technical Journal*, **27**, 379-423.

Shrikumar A, Greenside P, Kundaje A (2017). "Learning important features through propagating activation differences." *Proceedings of the 34th International Conference on Machine Learning*, **70**, 3145-3153.

Shrikumar A, Tian K, Avsec Ž, Shcherbina A, Banerjee A, Sharmin M, Nair S, Kundaje A (2018). "TF-MoDISco v0.4.4.2-alpha: Technical Note." arXiv:1811.00416.

Stormo GD, Schneider TD, Gold L, Ehrenfeucht A (1982). "Use of the Perceptron algorithm to distinguish translational initiation sites in E. coli." *Nucleic Acids Research*, **10**, 2997-3011.

Tan G, Lenhard B (2016). "TFBSTools: an R/Bioconductor package for transcription factor binding site analysis." *Bioinformatics*, **32**, 1555-1556. doi: 10.1093/bioinformatics/btw024.

See Also

convert_motifs()

Examples

```
jaspar.pcm <- read_jaspar(system.file("extdata", "jaspar.txt",
                                      package = "universalmotif"))

## The motifs pseudocounts are 1: these will be used in the PCM->PPM
## calculation
jaspar.pwm <- convert_type(jaspar.pcm, type = "PPM")

## Setting pseudocount to 0 will prevent any correction from being
## applied to PPM/PWM matrices, overriding the motifs own pseudocounts
jaspar.pwm <- convert_type(jaspar.pcm, type = "PWM", pseudocount = 0)

## CWM -> PPM via |cwm| / sum (the TF-MoDISco convention):
cwm.mat <- matrix(c( 0.10, -0.20,  0.80, -0.10,
                    -0.30,  0.90, -0.20,  0.10,
                     0.70, -0.10, -0.10, -0.40,
                    -0.20, -0.10, -0.10,  0.90),
                  nrow = 4, byrow = FALSE,
                  dimnames = list(c("A","C","G","T"), NULL))
cwm <- create_motif(cwm.mat, type = "CWM", name = "modisco_example")
convert_type(cwm, "PPM")
```

create_motif *Create a motif.*

Description

Create a motif from a set of sequences, a matrix, or generate a random motif. See the "Motif import, export and manipulation" vignette for details.

Usage

```
create_motif(input, alphabet, type = "PPM", name = "motif",
  pseudocount = 0, bkg, nsites, altname, family, organism, bkgsites, strand,
  pval, qval, eval, extrainfo, add.multifreq)

## S4 method for signature 'missing'
create_motif(input, alphabet, type = "PPM",
  name = "motif", pseudocount = 0, bkg, nsites, altname, family, organism,
  bkgsites, strand, pval, qval, eval, extrainfo, add.multifreq)

## S4 method for signature 'numeric'
create_motif(input, alphabet, type = "PPM",
  name = "motif", pseudocount = 0, bkg, nsites, altname, family, organism,
  bkgsites, strand, pval, qval, eval, extrainfo, add.multifreq)

## S4 method for signature 'character'
create_motif(input, alphabet, type = "PPM",
  name = "motif", pseudocount = 0, bkg, nsites, altname, family, organism,
  bkgsites, strand, pval, qval, eval, extrainfo, add.multifreq)

## S4 method for signature 'matrix'
create_motif(input, alphabet, type = "PPM",
  name = "motif", pseudocount = 0, bkg, nsites, altname, family, organism,
  bkgsites, strand, pval, qval, eval, extrainfo, add.multifreq)

## S4 method for signature 'DNAStringSet'
create_motif(input, alphabet, type = "PPM",
  name = "motif", pseudocount = 0, bkg, nsites, altname, family, organism,
  bkgsites, strand, pval, qval, eval, extrainfo, add.multifreq)

## S4 method for signature 'RNAStringSet'
create_motif(input, alphabet, type = "PPM",
  name = "motif", pseudocount = 0, bkg, nsites, altname, family, organism,
  bkgsites, strand, pval, qval, eval, extrainfo, add.multifreq)

## S4 method for signature 'AAStringSet'
create_motif(input, alphabet, type = "PPM",
  name = "motif", pseudocount = 0, bkg, nsites, altname, family, organism,
  bkgsites, strand, pval, qval, eval, extrainfo, add.multifreq)

## S4 method for signature 'BStringSet'
create_motif(input, alphabet, type = "PPM",
```

```
name = "motif", pseudocount = 0, bkg, nsites, altname, family, organism,
bkgsites, strand, pval, qval, eval, extrainfo, add.multifreq)
```

Arguments

input	character, numeric, matrix, XStringSet, or missing.
alphabet	character(1) One of c('DNA', 'RNA', 'AA'), or a combined string representing the letters. If no alphabet is provided then it will try and guess the alphabet from the input.
type	character(1) One of c('PCM', 'PPM', 'PWM', 'ICM', 'CWM'). 'CWM' is intended for matrix input only. See convert_type() for the CWM details and the available conversions.
name	character(1) Motif name.
pseudocount	numeric(1) Correction to be applied to prevent -Inf from appearing in PWM matrices. Defaults to 0.
bkg	numeric A vector of probabilities, each between 0 and 1. If higher order backgrounds are provided, then the elements of the vector must be named. If unnamed, then the order of probabilities must be in the same order as the alphabetically sorted sequence alphabet.
nsites	numeric(1) Number of sites the motif was constructed from. If blank, then create_motif() will guess the appropriate number if possible. To prevent this, provide nsites = numeric(). NA_real_ also denotes unknown. Otherwise supply a finite positive integer; zero is not an unknown-value sentinel. CWM column sums do not represent site counts.
altname	character(1) Alternate motif name.
family	character(1) Transcription factor family.
organism	character(1) Species of origin.
bkgsites	numeric(1) Total number of sites used to find the motif.
strand	character(1) Whether the motif is specific to a certain strand. Acceptable strands are '+', '-', and '+-' (to represent both strands). Note that '-+' and '*' can also be provided to represent both strands, but the final strand in the universalmotif object will be set to '+-'.
pval	numeric(1) P-value associated with motif.
qval	numeric(1) Adjusted P-value associated with motif.
eval	numeric(1) E-value associated with motif.
extrainfo	character Any other extra information, represented as a named character vector.
add.multifreq	numeric If the motif is created from a set of sequences, then the add_multifreq() function can be run at the same time (with RC = FALSE).

Details

The aim of this function is provide an easy interface to creating universalmotif motifs, as an alternative to the default class constructor (i.e. new('universalmotif', name=...)). See examples for potential use cases.

Note: when generating random motifs, the nsites slot is also given a random value.

Note: type = "CWM" is only supported when input is a matrix. CWMs originate from contribution-attribution methods (TF-MoDISco, DeepLIFT) and cannot be derived from sequences, consensus

strings, or random generation. The input matrix is stored as-is (signed real values, no column-sum constraint); no auto-normalisation is applied. See convert_type() for the full CWM semantics.

See the examples section for more info on motif creation.

Value

universalmotif object.

Methods (by class)

- create_motif(missing): Create a random motif of length 10.
- create_motif(numeric): Create a random motif with a specified length.
- create_motif(character): Create motif from a consensus string.
- create_motif(matrix): Create motif from a matrix.
- create_motif(DNAStringSet): Create motif from a DNAStringSet.
- create_motif(RNAStringSet): Create motif from a RNAStringSet.
- create_motif(AAStringSet): Create motif from a AAStringSet.
- create_motif(BStringSet): Create motif from a BStringSet.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

convert_type(), add_multifreq(), create_sequences(), shuffle_motifs().

Examples

```
##### create motifs from a single string

# Motif is by default generated as a PPM: change final type as desired
DNA.motif <- create_motif("TATAWAW")
DNA.motif <- create_motif("TATAWAW", type = "PCM")

# Nsites will be set to the number of input sequences unless specified or
# a single string is used as input
DNA.motif <- create_motif("TTTTTTT", nsites = 10)

# Ambiguity letters can be used:
DNA.motif <- create_motif("TATAWAW")
DNA.motif <- create_motif("NNVVWWAAWWDDN")

# Be careful about setting nsites when using ambiguity letters!
DNA.motif <- create_motif("NNVVWWAAWWDDN", nsites = 1)

RNA.motif <- create_motif("UUUCCG")

# 'create_motif' will try to detect the alphabet type; this can be
# unreliable for AA and custom alphabets as DNA and RNA alphabets are
# detected first
AA.motif <- create_motif("AVLK", alphabet = "AA")
```

```
custom.motif <- create_motif("QWER", alphabet = "QWER")
# Specify custom alphabet
custom.motif <- create_motif("QWER", alphabet = "QWERASDF")

###### Create motifs from multiple strings of equal length

DNA.motif <- create_motif(c("TTTT", "AAAA", "AACC", "TTGG"), type = "PPM")
DNA.motif <- create_motif(c("TTTT", "AAAA", "AACC", "TTGG"), nsites = 20)
RNA.motif <- create_motif(c("UUUU", "AAAA", "AACC", "UUGG"), type = "PWM")
AA.motif <- create_motif(c("ARNDCQ", "EGHILK", "ARNDCQ"), alphabet = "AA")
custom.motif <- create_motif(c("POIU", "LKJH", "POIU", "CVBN"),
                             alphabet = "POIULKJHCVBN")

# Ambiguity letters are only allowed for single consensus strings: the
# following fails
## Not run:
create_motif(c("WWTT", "CCGG"))
create_motif(c("XXXX", "XXXX"), alphabet = "AA")

## End(Not run)

##### Create motifs from XStringSet objects

library(Biostrings)

DNA.set <- DNAStringSet(c("TTTT", "AAAA", "AACC", "TTGG"))
DNA.motif <- create_motif(DNA.set)
RNA.set <- RNAStringSet(c("UUUU", "AACC", "UUCC"))
RNA.motif <- create_motif(RNA.set)
AA.set <- AAStringSet(c("VVVLLL", "AAAIII"))
AA.motif <- create_motif(AA.set)

# Custom motifs can be created from BStringSet objects
B.set <- BStringSet(c("QWER", "ASDF", "ZXCV", "TYUI"))
custom.motif <- create_motif(B.set)

##### Create motifs with filled 'multifreq' slot

DNA.motif.k2 <- create_motif(DNA.set, add.multifreq = 2)

##### Create motifs from matrices

mat <- matrix(c(1, 1, 1, 1,
                2, 0, 2, 0,
                0, 2, 0, 2,
                0, 0, 0, 0),
                nrow = 4, byrow = TRUE)
DNA.motif <- create_motif(mat, alphabet = "DNA")
RNA.motif <- create_motif(mat, alphabet = "RNA", nsites = 20)
custom.motif <- create_motif(mat, alphabet = "QWER")

# Specify custom alphabet
custom.motif <- create_motif(mat, alphabet = "QWER")

# Alphabet can be detected from rownames
rownames(mat) <- DNA_BASES
DNA.motif <- create_motif(mat)
```

```
rownames(mat) <- c("Q", "W", "E", "R")
custom.motif <- create_motif(mat)

# Matrices can also be used as input
mat.ppm <- matrix(c(0.1, 0.1, 0.1, 0.1,
                    0.5, 0.5, 0.5, 0.5,
                    0.1, 0.1, 0.1, 0.1,
                    0.3, 0.3, 0.3, 0.3),
                    nrow = 4, byrow = TRUE)

DNA.motif <- create_motif(mat.ppm, alphabet = "DNA", type = "PPM")

# Contribution weight matrices (signed, no column-sum constraint).
# Typically these come from TF-MoDISco / DeepLIFT pipelines and are
# imported via `read_meme(..., CWM = TRUE)`, but can be built from
# a matrix directly:
cwm.mat <- matrix(c( 0.10, -0.20,  0.80, -0.10,
                    -0.30,  0.90, -0.20,  0.10,
                     0.70, -0.10, -0.10, -0.40,
                    -0.20, -0.10, -0.10,  0.90),
                  nrow = 4, byrow = FALSE,
                  dimnames = list(c("A","C","G","T"), NULL))
CWM.motif <- create_motif(cwm.mat, type = "CWM", name = "modisco_example")

##### Create random motifs

# These are generated as PPMs with 10 positions

DNA.motif <- create_motif()
RNA.motif <- create_motif(alphabet = "RNA")
AA.motif <- create_motif(alphabet = "AA")
custom.motif <- create_motif(alphabet = "QWER")

# The number of positions can be specified

DNA.motif <- create_motif(5)

# If the background frequencies are not provided, they are generated
# using `rpois`; positions are created using `rdirichlet(1, bkg)`.
# (calling `create_motif()` creates motifs with an average
# positional IC of 1)

DNA.motif <- create_motif(bkg = c(0.3, 0.2, 0.2, 0.3))
DNA.motif <- create_motif(10, bkg = c(0.1, 0.4, 0.4, 0.1))
```

`create_sequences` *Create random sequences.*

Description

Generate random sequences from any set of characters, represented as `XStringSet` objects.

Usage

```
create_sequences(alphabet = "DNA", seqnum = 100, seqlen = 100, freqs,
  nthreads = 1, rng.seed = sample.int(10000, 1))
```

Arguments

alphabet	character(1) One of c('DNA', 'RNA', 'AA'), or a string of characters to be used as the alphabet.
seqnum	numeric(1) Number of sequences to generate.
seqlen	numeric(1) Length of random sequences.
freqs	numeric A named vector of probabilities. The length of the vector must be the power of the number of letters in the sequence alphabet. Probabilities can only be provided for a single size k.
nthreads	numeric(1) Run create_sequences() in parallel with nthreads threads. nthreads = 0 uses all available threads. Note that no speed up will occur for jobs with seqnum = 1.
rng.seed	numeric(1) Set random number generator seed. Since sequence creation can occur simultaneously in multiple threads using C++, it cannot communicate with the regular R random number generator state and thus requires an independent seed. Each individual sequence creation instance is given the following seed: rng.seed * index. The default is to pick a random number as chosen by sample(), which effectively is making create_sequences() dependent on the R RNG state.

Value

XStringSet The returned sequences are *unnamed*.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

create_motif(), shuffle_sequences()

Examples

```
## Create DNA sequences with slightly increased AT content:
sequences <- create_sequences(freqs = c(A=0.3, C=0.2, G=0.2, T=0.3))
## Create custom sequences:
sequences.QWER <- create_sequences("QWER")
## You can include non-alphabet characters are well, even spaces:
sequences.custom <- create_sequences("!@#$ ")
```

dedup_hits *Deduplicate overlapping motif hits.*

Description

Given a set of motif hits, typically the output of `scan_sequences()` or `scan_sequences_lite()`, or a `GRanges` / `data.frame` from any other source, drop overlapping hits within each `(sequence, motif, strand)` group using a greedy algorithm: cluster overlapping hits, then within each cluster keep the hit with the best priority and, after that, any other hit that does not overlap an already-kept hit.

Usage

```
dedup_hits(hits, by = "pvalue", reverse = FALSE, ignore.strand = FALSE,
  ignore.motif = FALSE, seq = "sequence", motif = "motif",
  strand = "strand")
```

Arguments

`hits`	`data.frame` or `GRanges`. For a `data.frame`, must have columns `start`, `end`, and the columns named by `by`, `seq`, `motif`, `strand`. For a `GRanges`, the `seqnames`/`start`/`end`/`strand` slots are used; the `motif` and priority columns are read from `mcols()`.
`by`	`character(1)`. Name of the priority column. The semantic is auto-detected: a column named `pvalue` (or anything starting with `pval`) is treated as lower-is-better; anything else (e.g. `score`) is treated as higher-is-better. Use `reverse = TRUE` to flip.
`reverse`	`logical(1)`. Flip the priority direction.
`ignore.strand`	`logical(1)`. If `TRUE`, `+` and `-` hits compete with each other at overlapping coordinates.
`ignore.motif`	`logical(1)`. If `TRUE`, hits from different motifs compete with each other at overlapping coordinates.
`seq`	`character(1)`. Name of the sequence-id column in a `data.frame` input. Default `"sequence"`. Ignored for `GRanges`.
`motif`	`character(1)`. Name of the motif-id column in a `data.frame` input. Default `"motif"`. Ignored for `GRanges`.
`strand`	`character(1)`. Name of the strand column in a `data.frame` input. Default `"strand"`. Ignored for `GRanges`.

Details

Note that this differs from `scan_sequences()`'s built-in `no.overlaps` argument, which collapses every connected-overlap cluster to a single survivor (hierarchical clustering with `cutree`). The greedy approach used here keeps legitimately distinct hits even when an intermediate noisy hit bridges them, which is usually the desired behaviour.

Coordinate convention: rows must have `start <= end`. Overlaps are inclusive: two hits overlap if `a$start <= b$end & b$start <= a$end`.

Value

The input `hits`, subsetted to the kept rows in the original order.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

scan_sequences_lite(), scan_sequences()

Examples

```
library(universalmotif)
motifs <- list(create_motif("TATAAA", name = "M1"),
               create_motif("CACGTG", name = "M2"))
seqs <- create_sequences(seqnum = 5, seqlen = 400, rng.seed = 1)
hits <- scan_sequences_lite(motifs, seqs, pvalue = 5e-2,
                            return.granges = FALSE)
deduped <- dedup_hits(hits)
nrow(hits)
nrow(deduped)
```

---

enrich_motifs *Enrich for input motifs in a set of sequences.*

Description

Given a set of target and background sequences, test if the input motifs are significantly enriched in the targets sequences relative to the background sequences. See the "Sequence manipulation and scanning" vignette. Note: for regular DNA/RNA motif enrichment, see enrich_motifs_lite() for a faster and simpler alternative.

Usage

```
enrich_motifs(motifs, sequences, bkg.sequences, max.p = 0.001,
  max.q = 0.001, max.e = 0.001, qval.method = "fdr", threshold = 1e-04,
  threshold.type = "pvalue", verbose = 0, RC = TRUE, use.freq = 1,
  shuffle.k = 2, shuffle.method = "euler", return.scan.results = FALSE,
  nthreads = 1, rng.seed = sample.int(10000, 1), motif_pvalue.k = 8,
  use.gaps = TRUE, allow.nonfinite = FALSE, warn.NA = TRUE,
  no.overlaps = TRUE, no.overlaps.by.strand = FALSE,
  no.overlaps.strat = "score", respect.strand = FALSE,
  motif_pvalue.method = c("dynamic", "exhaustive"),
  scan_sequences.qvals.method = c("BH", "fdr", "bonferroni"),
  mode = c("total.hits", "seq.hits"), pseudocount = 1)
```

Arguments

motifs	See convert_motifs() for acceptable motif formats.
sequences	XStringSet Sequences to scan. Alphabet should match motif.
bkg.sequences	XStringSet Optional. If missing, shuffle_sequences() is used to create background sequences from the input sequences.
max.p	numeric(1) P-value threshold.

max.q numeric(1) Adjusted P-value threshold. This is only useful if multiple motifs are being enriched for.

max.e numeric(1). The E-value is calculated by multiplying the P-value with the number of input motifs times two (McLeay and Bailey 2010).

qval.method character(1) See stats::p.adjust().

threshold numeric(1) See details.

threshold.type character(1) One of c('pvalue', 'qvalue', 'logodds', 'logodds.abs'). See details.

verbose numeric(1) 0 for no output, 4 for max verbosity.

RC logical(1) If TRUE, check reverse complement of the input sequences. Only available for DNA/RNA.

use.freq numeric(1) The default, 1, uses the motif matrix (from the motif['motif'] slot) to search for sequences. If a higher number is used, then the matching k-let matrix from the motif['multifreq'] slot is used. See add_multifreq().

shuffle.k numeric(1) The k-let size to use when shuffling input sequences. Only used if no background sequences are input. See shuffle_sequences().

shuffle.method character(1) One of c('euler', 'markov', 'linear'). See shuffle_sequences().

return.scan.results
logical(1) Return output from scan_sequences(). For large jobs, leaving this as FALSE can save a small amount time by preventing construction of the complete results data.frame from scan_sequences().

nthreads numeric(1) Run scan_sequences() in parallel with nthreads threads. nthreads = 0 uses all available threads. Note that no speed up will occur for jobs with only a single motif and sequence.

rng.seed numeric(1) Set random number generator seed. Since shuffling can occur simultaneously in multiple threads using C++, it cannot communicate with the regular R random number generator state and thus requires an independent seed. Each individual sequence in an XStringSet object will be given the following seed: rng.seed * index. See shuffle_sequences().

motif_pvalue.k numeric(1) Control motif_pvalue() approximation. See motif_pvalue().

use.gaps logical(1) Set this to FALSE to ignore motif gaps, if present.

allow.nonfinite
logical(1) If FALSE, then apply a pseudocount if non-finite values are found in the PWM. Note that if the motif has a pseudocount greater than zero and the motif is not currently of type PWM, then this parameter has no effect as the pseudocount will be applied automatically when the motif is converted to a PWM internally. This value is set to FALSE by default in order to stay consistent with pre-version 1.8.0 behaviour. A message will be printed if a pseudocount is applied. To disable this, set options(pseudocount.warning=FALSE).

warn.NA logical(1) Whether to warn about the presence of non-standard letters in the input sequence, such as those in masked sequences.

no.overlaps logical(1) Remove overlapping hits from the same motifs. Overlapping hits from different motifs are preserved. Please note that the current implementation of this feature can add significantly to the run time for large inputs.

no.overlaps.by.strand
logical(1) Whether to discard overlapping hits from the opposite strand (TRUE), or to only discard overlapping hits on the same strand (FALSE).

`no.overlaps.strat`
: `character(1)` One of `c("score", "order")`. The former option keeps the highest scoring overlapping hit (and the first of these within ties), and the latter simply keeps the first overlapping hit.

`respect.strand`
: `logical(1)` If motifs are DNA/RNA, then setting this option to `TRUE` will make `scan_sequences()` only scan the strands of the input sequences as indicated in the motif `strand` slot.

`motif_pvalue.method`
: `character(1)` One of `c("dynamic", "exhaustive")`. Algorithm used for calculating P-values. The `"exhaustive"` method involves finding all possible motif matches at or above the specified score using a branch-and-bound algorithm, which can be computationally intensive (Hartman et al., 2013). Additionally, the computation must be repeated for each hit. The `"dynamic"` method calculates the distribution of possible motif scores using a much faster dynamic programming algorithm, and can be recycled for multiple scores (Grant et al., 2011). The only disadvantage is the inability to use `allow.nonfinite = TRUE`. See `motif_pvalue()` for details.

`scan_sequences.qvals.method`
: `character(1)` One of `c("fdr", "BH", "bonferroni")`. The method for calculating adjusted P-values for individual motif hits. These are described in depth in the Sequence Searches vignette.

`mode`
: `character(1)` One of `c("total.hits", "seq.hits")`. The former enriches for the total count of motif hits across all sequences, whereas the latter only counts motif hits once per sequence (useful for cases where there are many small sequences).

`pseudocount`
: `integer(1)` Add a pseudocount to the motif hit counts when performing the Fisher test.

Details

To find enriched motifs, `scan_sequences()` is run on both target and background sequences. `stats::fisher.test()` is run to test for enrichment.

See `scan_sequences()` for more info on scanning parameters.

Value

`DataFrame` Enrichment results in a `DataFrame`. Function args and (optionally) scan results are stored in the `metadata` slot.

Author(s)

Benjamin Jean-Marie Tremblay <benjamin.tremblay@uwaterloo.ca>

References

McLeay R, Bailey TL (2010). "Motif Enrichment Analysis: A unified framework and method evaluation." *BMC Bioinformatics*, **11**.

See Also

`scan_sequences()`, `shuffle_sequences()`, `add_multifreq()`, `motif_pvalue()`, `enrich_motifs_lite()`, `scan_sequences_lite()`, `match_bkg()`

Examples

```
data(ArabidopsisPromoters)
data(ArabidopsisMotif)
if (R.Version()$arch != "i386") {
enrich_motifs(ArabidopsisMotif, ArabidopsisPromoters, threshold = 0.01)
}
```

`enrich_motifs_lite` *Fast minimalist motif enrichment.*

Description

`enrich_motifs_lite()` is a deliberately pared-down counterpart to `enrich_motifs()`. It exposes a single p-value cutoff for the hits, a single q-value cutoff for the results, and two test modes (`"seqs"` and `"sites"`, both Fisher's exact), and it leans on `scan_sequences_lite()` under the hood to scan the target and background sequences. The p-value adjustment is hard-coded to Benjamini-Hochberg.

Usage

```
enrich_motifs_lite(motifs, sequences, bkg.sequences = NULL, pvalue = 1e-04,
  qvalue = 0.1, test = c("seqs", "sites"), RC = TRUE, shuffle.k = 2L,
  rng.seed = sample.int(1e+06, 1), pseudocount = 1L, nthreads = 1)
```

Arguments

`motifs`	See `convert_motifs()` for accepted motif formats. DNA or RNA only; amino-acid and custom alphabet motifs are rejected.
`sequences`	`XStringSet`. Target sequences.
`bkg.sequences`	`XStringSet` or `NULL`. Background sequences. If `NULL` (default), target sequences are shuffled k-let-conserving via `shuffle_sequences()` with `k = shuffle.k`, matching `enrich_motifs()` default behaviour.
`pvalue`	`numeric(1)`. P-value cutoff passed to `scan_sequences_lite()` for reporting individual hits. Default `1e-4`.
`qvalue`	`numeric(1)`. Result-level q-value cutoff. Motifs whose BH-adjusted enrichment p-value exceeds this are filtered out. Default `0.1`.
`test`	`character(1)`. One of `"seqs"` (default) or `"sites"`. `"seqs"` runs Fisher's exact on per-sequence hit presence (does this sequence contain >=1 hit?), equivalent to `enrich_motifs()`'s `mode = "seq.hits"`. `"sites"` runs Fisher's exact on per-position rates (how many hits per scannable position?), equivalent to `mode = "total.hits"`.
`RC`	`logical(1)`. If `TRUE` (default), scan both strands.
`shuffle.k`	`integer(1)`. K-let size for the background shuffle (only used when `bkg.sequences = NULL`). Default `2L`.
`rng.seed`	`integer(1)`. RNG seed for background shuffling. Default `sample.int(1e6, 1)`; set explicitly for reproducible runs.
`pseudocount`	`integer(1)`. Pseudocount added uniformly to each cell of the Fisher 2x2 table; useful when one side has zero counts. Default `1L`.
`nthreads`	`numeric(1)`. Number of threads passed to `scan_sequences_lite()` and `shuffle_sequences()`. `nthreads = 0` uses all available threads.

Details

Use `enrich_motifs()` when you need any of: q-value adjustment methods other than BH; multi-freq / gapped motifs; multiple-testing E-values; the `respect.strand`, `allow.nonfinite`, or non-pvalue `threshold.type` machinery; or amino-acid motifs.

Internally, `scan_sequences_lite()` is run once on the target set and once on the background set; the resulting hit tables are aggregated per motif into a 2x2 contingency table whose entries depend on `test`:

`"seqs"` `a` = target sequences with >=1 hit; `b` = background sequences with >=1 hit; `c` = target sequences with no hits; `d` = background sequences with no hits.

`"sites"` `a` = total hits in target; `b` = total hits in background; `c` = scannable target positions minus `a`; `d` = scannable background positions minus `b`. Scannable positions are computed per motif as `sum(pmax(width(seqs) - motif_width + 1, 0))`, doubled if `RC = TRUE`.

Each cell has `pseudocount` added before `stats::fisher.test()` is run with `alternative = "greater"`. Q-values are computed via `p.adjust(method = "BH")` across all input motifs (not just the ones that pass the filter), then rows with `qvalue > qvalue` are dropped.

Value

A `data.frame` with one row per significant motif, sorted by `pvalue` ascending. Columns:

- `motif`: motif name
- `motif.i`: 1-based index into the input `motifs`
- `consensus`: IUPAC consensus of the motif
- `target.seq.n`: number of target sequences scanned
- `target.seq.hits`: number of target sequences containing >=1 hit
- `target.site.hits`: total hit count across all target sequences
- `bkg.seq.n`: number of background sequences scanned
- `bkg.seq.hits`: number of background sequences containing >=1 hit
- `bkg.site.hits`: total hit count across all background sequences
- `enrichment`: fold-enrichment (rate ratio; floor of 0.5/n in the denominator to avoid division by zero)
- `log2.enrichment`: `log2(enrichment)`
- `pvalue`: Fisher's exact one-sided ("greater") p-value
- `qvalue`: Benjamini-Hochberg adjusted p-value across motifs

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

McLeay R, Bailey TL (2010). "Motif Enrichment Analysis: A unified framework and method evaluation." *BMC Bioinformatics*, **11**.

See Also

enrich_motifs(), scan_sequences_lite(), shuffle_sequences(), match_bkg(), motif_pvalue()

Other lite motif functions: compare_motifs_lite(), merge_motifs_lite(), merge_similar_lite(), motif_tree_lite(), scan_sequences_lite(), view_motifs_lite()

Examples

```
library(universalmotif)
data(ArabidopsisPromoters)
data(ArabidopsisMotif)
if (R.Version()$arch != "i386") {
  enrich_motifs_lite(ArabidopsisMotif, ArabidopsisPromoters,
                pvalue = 1e-3, qvalue = 0.5, rng.seed = 1)
}
```

---

examplemotif *Example motif in* universalmotif *format.*

Description

A simple DNA motif. To recreate this motif: create_motif("TATAWAW", nsites = numeric())

Usage

```
examplemotif
```

Format

universalmotif

Examples

```
data(examplemotif)
examplemotif
```

---

examplemotif2 *Another example motif in* universalmotif *format.*

Description

A simple DNA motif with a non-empty multifreq slot. To recreate to this motif: add_multifreq(examplemotif, DNAStringSet(rep(c("CAAAACC", "CTTTTCC"), 3)))

Usage

```
examplemotif2
```

Format

universalmotif

Examples

```
data(examplemotif2)
examplemotif2
```

`filter_motifs` *Filter a list of motifs.*

Description

Filter motifs based on the contents of available universalmotif slots. If the input motifs are not of universalmotif, then they will be converted for the duration of the `filter_motifs()` operation.

Usage

```
filter_motifs(motifs, name, altname, family, organism, width, alphabet, type,
  icscore, nsites, strand, pval, qval, eval, extrainfo)
```

Arguments

`motifs`	`list` See `convert_motifs()` for acceptable formats.
`name`	`character` Keep motifs by names.
`altname`	`character` Keep motifs by altnames.
`family`	`character` Keep motifs by family.
`organism`	`character` Keep motifs by organism.
`width`	`numeric(1)` Keep motifs with minimum width.
`alphabet`	`character` Keep motifs by alphabet.
`type`	`character` Keep motifs by type.
`icscore`	`numeric(1)` Keep motifs with minimum total IC.
`nsites`	`numeric(1)` Keep motifs with minimum number of target sites.
`strand`	`character` Keeps motifs by strand.
`pval`	`numeric(1)` Keep motifs by max P-value.
`qval`	`numeric(1)` Keep motifs by max Q-value.
`eval`	`numeric(1)` Keep motifs by max E-val.
`extrainfo`	`character` Named character vector of items that must be present in motif `extrainfo` slots.

Value

`list` Motifs. An attempt will be made to preserve the original class, see `convert_motifs()` for limitations.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

Examples

```
## By minimum IC:
jaspar <- read_jaspar(system.file("extdata", "jaspar.txt",
                                  package = "universalmotif"))
jaspar.ic3 <- filter_motifs(jaspar, icscore = 3)

## Starting from version 1.10.0 of the universalmotif package, one
## could instead make use of the universalmotif_df structure:
jaspar.ic3 <- jaspar |> to_df() |> subset(icscore > 3) |> to_list()

## By organism:
## Not run:
library(MotifDb)
motifs <- convert_motifs(MotifDb)
motifs <- filter_motifs(motifs, organism = c("Athaliana", "Mmusculus"),
                        extrainfo = c("dataSource" = "cisbp_1.02"))
## Or:
motifs <- convert_motifs(MotifDb) |> to_df() |>
  subset(organism %in% c("Athaliana", "Mmusculus") &
    dataSource == "cisbp_1.02") |> to_list()

## End(Not run)
```

---

fontDFroboto *Polygon coordinates for plotting letters.*

Description

`DataFrame` of polygon coordinates used by `view_motifs()` for plotting letters. It was generated using the `createPolygons` function from the `gglogo` package for the font Roboto Medium.

Usage

```
fontDFroboto
```

Format

S4Vectors::DataFrame

Examples

```
data(fontDFroboto)
head(fontDFroboto)
```

geom_motif *Embed motif logos in ggplot2 constructions.*

Description

These functions expose the logo-drawing machinery of `view_motifs()` and `view_logo()` as composable **ggplot2** layers, so a sequence logo can be dropped into a plot you are already building (annotating a coverage track, adding an inset to a scatter plot, placing a logo at every tip of a tree, and so on) instead of being returned as a standalone plot.

Usage

```
geom_motif(mapping = NULL, data = NULL, stat = StatMotif,
  position = "identity", ..., motif, use.type = "ICM",
  colour.scheme = NULL, fontDF = NULL, width = 1, height = 0.9,
  na.rm = FALSE, show.legend = FALSE, inherit.aes = TRUE)

geom_logo(mapping = NULL, data = NULL, stat = StatLogo,
  position = "identity", ..., logo, fill = "black", colour.scheme = NULL,
  fontDF = NULL, width = 1, height = 0.9, na.rm = FALSE,
  show.legend = FALSE, inherit.aes = TRUE)

annotate_motif(motif, xmin, xmax, ymin, ymax, use.type = "ICM",
  colour.scheme = NULL, fontDF = NULL, ...)

annotate_logo(logo, xmin, xmax, ymin, ymax, fill = "black",
  colour.scheme = NULL, fontDF = NULL, ...)
```

Arguments

`mapping`	Aesthetic mapping created with `ggplot2::aes()`. The layer understands two positioning modes (see Details): a bounding box (`xmin`, `xmax`, `ymin`, `ymax`) or an anchor (`x`, `y`). The optional `motif` aesthetic selects, per group, which entry of a named-list `motif` (or `logo`) argument to draw, e.g. `aes(x = 0, y = y, motif = label)`.
`data`	A data frame, if the layer should override the plot data.
`stat, position, na.rm, show.legend, inherit.aes`	Standard **ggplot2** layer arguments. Colours are resolved internally rather than through the plot's `fill` scale, so the layer never collides with a `fill` mapping in the host plot and no legend is added; `show.legend` therefore defaults to `FALSE`.
`...`	Passed to `ggplot2::layer()`, and on to the underlying logo engine: `min.height`, `x.spacer`, `y.spacer`, `sort.positions`, `sort.positions.decreasing`, `flip.neg` (same meaning as in `view_motifs()` / `view_logo()`).
`motif`	A single motif (anything `convert_motifs()` accepts) or a **named list** of motifs. With a named list, map the `motif` aesthetic to a column whose values are names of that list to draw a different logo per group.
`use.type`	`character(1)` One of `c('PCM', 'PPM', 'PWM', 'ICM')`. The matrix type used for display (motif variants only). See `view_motifs()`.

`colour.scheme` `character` A named vector giving a colour for every individual character. For the motif variants the default is the standard DNA/RNA/AA palette (or a generated palette for custom alphabets). For the bare-matrix logo variants the alphabet is unknown, so when `colour.scheme` is `NULL` every letter is drawn in the single `fill` colour instead.

`fontDF` `data.frame` or `DataFrame` Polygon paths for the letters. By default the bundled fontDFroboto glyphs are used.

`width` `numeric(1)` In anchored mode, the width of one motif position in data coordinates (so the logo spans `x` to `x + ncol * width`).

`height` `numeric(1)` In anchored mode, the total logo height in data coordinates, centred on `y`.

`logo` A single row-named numeric matrix or a **named list** of such matrices, used the same way as `motif` but drawn with the values taken as-is (the `view_logo()` path).

`fill` `character(1)` A single colour for every letter (logo variants), used when `colour.scheme` is `NULL`.

`xmin, xmax, ymin, ymax`
`numeric(1)` Bounding box in data coordinates for `annotate_motif()` / `annotate_logo()`. The logo is linearly rescaled to fill the box.

Details

There are two pairs. `geom_motif()` / `annotate_motif()` take a `universalmotif` object (or anything `convert_motifs()` accepts) and honour `use.type` exactly like `view_motifs()`. `geom_logo()` / `annotate_logo()` take a bare row-named numeric matrix with arbitrary values, like `view_logo()`. In each pair the `annotate_*()` function places one logo at fixed coordinates (no data or aesthetic inheritance), while the `geom_*()` function is a true layer whose position comes from aesthetics, so it picks up faceting and can be driven from a data frame.

Letters are drawn as filled, borderless polygons whose colour is resolved internally (so the layer never competes for the plot's `fill` scale and adds no legend). Consequently the `colour`, `alpha`, and `linetype` aesthetics have no effect; appearance is controlled only through `colour.scheme` (or `fill` for the logo variants).

Two positioning modes are supported and chosen per row from the supplied aesthetics:

- **Bounding box**: supply `xmin`, `xmax`, `ymin`, `ymax` (the model used by `ggplot2::annotation_custom()`). The whole logo is rescaled into the box.
- **Anchored**: supply `x` (the left edge) and `y` (the vertical centre); the box is then `xmin = x`, `xmax = x + ncol * width`, `ymin = y - height/2`, `ymax = y + height/2`. This is the natural fit for placing logos at tree tips.

To draw several motifs aligned to a common column frame (for example a stack of aligned logos, or aligned tip logos on a tree), align them first with `view_motifs_lite()` (or `view_motifs()`) and pass the result to `geom_logo()`:

```
aligned <- view_motifs_lite(motifs, return.raw = TRUE, use.type = "ICM",
                            sort.by = "none")
ggplot(tipdf, aes(x = 0, y = y, motif = label)) +
  geom_logo(logo = aligned, colour.scheme = DNA_COLOURS)
```

Value

A **ggplot2** layer to add to a plot with +.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

view_motifs(), view_logo(), view_motifs_lite(), motif_tree_lite()

Examples

```
library(ggplot2)
data(examplemotif)

## Annotate a logo onto an existing plot:
df <- data.frame(x = 1:20, y = cumsum(rnorm(20)))
ggplot(df, aes(x, y)) +
  geom_line() +
  annotate_motif(examplemotif, xmin = 5, xmax = 11, ymin = 1, ymax = 4)

## A different motif per row, anchored at each y:
m1 <- create_motif("TATAAA", name = "a")
m2 <- create_motif("GGGCGG", name = "b")
tipdf <- data.frame(x = 0, y = c(1, 2), label = c("a", "b"))
ggplot(tipdf, aes(x = x, y = y, motif = label)) +
  geom_motif(motif = list(a = m1, b = m2), height = 0.8)
```

---

get_bkg *Calculate sequence background.*

Description

For a set of input sequences, calculate the overall sequence background for any k-let size. For very large sequences DNA and RNA sequences (in the billions of bases), please be aware of the much faster and more efficient Biostrings::oligonucleotideFrequency(). get_bkg() can still be used in these cases, though it may take several seconds or minutes to calculate the results (depending on requested k-let sizes).

Usage

```
get_bkg(sequences, k = 1:3, as.prob = NULL, pseudocount = 0,
  alphabet = NULL, to.meme = NULL, RC = FALSE, list.out = NULL,
  nthreads = 1, merge.res = TRUE, window = FALSE, window.size = 0.1,
  window.overlap = 0)
```

Arguments

sequences	XStringSet Input sequences. Note that if multiple sequences are present, the results will be combined into one (unless merge.res = FALSE).
k	integer Size of k-let. Background can be calculated for any k-let size.
as.prob	Deprecated.
pseudocount	integer(1) Add a count to each possible k-let. Prevents any k-let from having 0 or 1 probabilities.

alphabet	character(1) Provide a custom alphabet to calculate a background for. If NULL, then standard letters will be assumed for DNA, RNA and AA sequences, and all unique letters found will be used for BStringSet type sequences. Note that letters which are not a part of the standard DNA/RNA/AA alphabets or in the provided alphabet will not be counted in the totals during probability calculations.
to.meme	If not NULL, then get_bkg() will return the sequence background in MEME Markov Background Model format. Input for this argument will be used for cat(..., file = to.meme) within get_bkg(). See http://meme-suite.org/doc/bfile-format.html for a description of the format.
RC	logical(1) Calculate the background of the reverse complement of the input sequences as well. Only valid for DNA/RNA.
list.out	Deprecated.
nthreads	numeric(1) Run get_bkg() in parallel with nthreads threads. nthreads = 0 uses all available threads. Note that no speed up will occur for jobs with only a single sequence.
merge.res	logical(1) Whether to merge results from all sequences or return background data for individual sequences.
window	logical(1) Determine background in windows.
window.size	numeric Window size. If a number between 0 and 1 is provided, the value is calculated as the number multiplied by the sequence length.
window.overlap	numeric Overlap between windows. If a number between 0 and 1 is provided, the value is calculated as the number multiplied by the sequence length.

Value

If to.meme = NULL, a DataFrame with columns klet, count, and probability. If merge.res = FALSE, there will be an additional sequence column. If window = TRUE, there will be an additional start and stop columns.

If to.meme is not NULL, then NULL is returned, invisibly.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Bailey TL, Elkan C (1994). "Fitting a mixture model by expectation maximization to discover motifs in biopolymers." *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*, **2**, 28-36.

See Also

create_sequences(), scan_sequences(), shuffle_sequences()

Examples

```
## Compare to Biostrings version
library(Biostrings)
seqs.DNA <- create_sequences()
bkg.DNA <- get_bkg(seqs.DNA, k = 3)
```

```
bkg.DNA2 <- oligonucleotideFrequency(seqs.DNA, 3, 1, as.prob = FALSE)
bkg.DNA2 <- colSums(bkg.DNA2)
all(bkg.DNA$count == bkg.DNA2)

## Create a MEME background file
get_bkg(seqs.DNA, k = 1:3, to.meme = stdout(), pseudocount = 1)

## Non-DNA/RNA/AA alphabets
seqs.QWERTY <- create_sequences("QWERTY")
bkg.QWERTY <- get_bkg(seqs.QWERTY, k = 1:2)
```

`implant_motifs` *Implant sampled motif instances into sequences at known positions.*

Description

`implant_motifs()` overwrites bases in an existing `XStringSet` with samples drawn column-by-column from one or more motif PPMs at chosen positions, returning the modified sequences together (optionally) with a ground-truth data.frame of where each implant was placed. This produces benchmark-grade positive sequences for testing `scan_sequences_lite()`, `motif_finder()`, `enrich_motifs_lite()`, `motif_peaks()`, or any other discovery / scanning pipeline against a known answer.

Usage

```
implant_motifs(motifs, sequences, n.per.seq = 1L, rate = NULL,
  positions = NULL, strand = c("both", "+"), centre.bias = 1L,
  min.spacing = 0L, max.retries = 100L, return.indices = FALSE)
```

Arguments

`motifs`	See `convert_motifs()` for accepted motif formats. DNA or RNA only, amino-acid and custom alphabet motifs are rejected. A single motif or a list of motifs.
`sequences`	`XStringSet`. Host sequences to plant into. Must share the motif alphabet.
`n.per.seq`	`integer(1)`. Mode A. Number of instances to plant in each sequence. Default `1L`.
`rate`	`numeric(1)` or `NULL`. Mode B. Per-bp Poisson rate. If non-`NULL`, overrides `n.per.seq`. Default `NULL`.
`positions`	`list` or `NULL`. Mode C. List of length `length(sequences)`; element `i` is an integer vector of 1-based start positions. Overrides `n.per.seq` and `rate`. Default `NULL`.
`strand`	`"both"` or `"+"`. Whether to also draw from the reverse complement. Default `"both"`.
`centre.bias`	`integer(1)`. Irwin-Hall N. `1L` (default) is uniform; larger values cluster implants near the centre of each sequence. Ignored in `positions` mode.
`min.spacing`	`integer(1)`. Minimum gap (in bp) between the end of one implant and the start of the next within the same sequence. `0L` (default) allows abutting but not overlapping implants. Ignored in `positions` mode.
`max.retries`	`integer(1)`. Per-implant cap on placement attempts before giving up (with a warning at the end if any targets fell short). Default `100L`.

`return.indices` `logical(1)`. If `FALSE` (default), return only the modified `XStringSet`. If `TRUE`, return a `list` with the modified sequences plus a `data.frame` of per-implant ground-truth metadata.

Details

Three insertion modes (mutually exclusive; later non-`NULL` argument wins):

1. Fixed count (`n.per.seq`): plant exactly `n.per.seq` instances in every sequence. Matches HOMER-style ground-truth simulations.
2. Poisson rate (`rate`): per-sequence count drawn from `rpois(1, rate * width(seq))`. Density scales naturally with sequence length.
3. Explicit positions (`positions`): a `list` of length `length(sequences)`; element `i` is a 1-based integer vector of start positions for that sequence. Bypasses `centre.bias`, `min.spacing`, and `max.retries`.

For each implant the function (a) picks a motif uniformly at random from `motifs`, (b) chooses a strand (uniform `+/-` if `strand = "both"`), (c) picks a start position (uniform or centre- biased via Irwin-Hall, retried up to `max.retries` times if it collides with an already-placed implant within `min.spacing`), (d) samples one instance from the motif's PPM column-by-column, reverse-complements if needed, and (e) overwrites that span in the target sequence.

Value

If `return.indices = FALSE`: an `XStringSet` of the same length, width, and class as `sequences`, with bases overwritten at the implant positions. Names are preserved. If `return.indices = TRUE`: a `list` with two elements – `sequences` (the modified `XStringSet`, as above) and `indices` (a `data.frame` with columns `sequence.i` (1-based), `motif.i` (1-based index into `motifs`), `start` (1-based), `width`, `strand` (`"+"` / `"-"`), and `planted` (the actual implanted string, already reverse-complemented when `strand == "-"`)).

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

`sample_sites()`, `create_sequences()`, `scan_sequences_lite()`, `motif_finder()`, `motif_peaks()`, `enrich_motifs_lite()`

Examples

```
data(ArabidopsisMotif)
seqs  <- create_sequences("DNA", seqnum = 100, seqlen = 500)
set.seed(1)
res   <- implant_motifs(ArabidopsisMotif, seqs, n.per.seq = 1,
                        return.indices = TRUE)
out   <- res$sequences
truth <- res$indices
## Recover-rate check
hits <- scan_sequences(ArabidopsisMotif, out, threshold = 0.85,
                       threshold.type = "logodds")
## Positional density of the recovered hits (qvalue = 1 keeps the
## motif even when, as here, the implants are placed uniformly)
peaks <- motif_peaks(as.data.frame(hits), seq.length = 500, qvalue = 1)
```

```
plot_motif_peaks(peaks)
```

JASPAR2018_CORE_DBSCORES *JASPAR2018 CORE database scores*

Description

For use with compare_motifs(). The precomputed scores allow for fast P-value estimation. These scores were generated using make_DBscores() with the JASPAR2018 CORE motif set. The scores are organized in a DataFrame. In this DataFrame is the location and scale of scores resulting from a statistical distribution using the the comparisons of JASPAR2018 CORE motifs with randomized motifs of the specified subject and target motif length. Created using make_DBscores() from universalmotif v1.4.0. The parameters used can be seen via S4Vectors::metadata(JASPAR2018_CORE_DBSCORES).

Usage

```
JASPAR2018_CORE_DBSCORES
```

Format

DataFrame with function args in the metadata slot.

Examples

```
data(JASPAR2018_CORE_DBSCORES)
head(JASPAR2018_CORE_DBSCORES)
```

---

make_DBscores *Create P-value databases.*

Description

Generate data used by compare_motifs() for P-value calculations. By default, compare_motifs() uses an internal database based on the JASPAR2018 core motifs (Khan et al. 2018). Parameters for distributions are are estimated for every combination of motif widths.

Usage

```
make_DBscores(db.motifs, method = c("PCC", "EUCL", "SW", "KL", "WEUCL",
  "ALLR", "BHAT", "HELL", "WPCC", "SEUCL", "MAN", "ALLR_LL"),
  shuffle.db = TRUE, shuffle.k = 3, shuffle.method = "linear",
  rand.tries = 1000, widths = 5:30, min.position.ic = 0,
  normalise.scores = c(FALSE, TRUE), min.overlap = 6, min.mean.ic = 0.25,
  progress = TRUE, nthreads = 1, tryRC = TRUE, score.strat = c("sum",
  "a.mean", "g.mean", "median", "wa.mean", "wg.mean", "fzt"))
```

Arguments

db.motifs	list Database motifs.
method	character(1) One of PCC, EUCL, SW, KL, ALLR, BHAT, HELL, SEUCL, MAN, ALLR_LL, WEUCL, WPCC. See details.
shuffle.db	logical(1) Deprecated. Does nothing. generate random motifs with create_motif().
shuffle.k	numeric(1) See shuffle_motifs().
shuffle.method	character(1) See shuffle_motifs().
rand.tries	numeric(1) Approximate number of comparisons to perform for every combination of widths.
widths	numeric Motif widths to use in P-value database calculation.
min.position.ic	numeric(1) Minimum information content required between individual alignment positions for it to be counted in the final alignment score. It is recommended to use this together with normalise.scores = TRUE, as this will help punish scores resulting from only a fraction of an alignment.
normalise.scores	logical(1) Favour alignments which leave fewer unaligned positions, as well as alignments between motifs of similar length. Similarity scores are multiplied by the ratio of aligned positions to the total number of positions in the larger motif, and the inverse for distance scores.
min.overlap	numeric(1) Minimum overlap required when aligning the motifs. Setting this to a number higher then the width of the motifs will not allow any overhangs. Can also be a number between 0 and 1, representing the minimum fraction that the motifs must overlap.
min.mean.ic	numeric(1) Minimum mean information content between the two motifs for an alignment to be scored. This helps prevent scoring alignments between low information content regions of two motifs. Note that this can result in some comparisons failing if no alignment passes the mean IC threshold. Use average_ic() to filter out low IC motifs to get around this if you want to avoid getting NAs in your output.
progress	logical(1) Show progress.
nthreads	numeric(1) Run compare_motifs() in parallel with nthreads threads. nthreads = 0 uses all available threads.
tryRC	logical(1) Try the reverse complement of the motifs as well, report the best score.
score.strat	character(1) How to handle column scores calculated from motif alignments. "sum": add up all scores. "a.mean": take the arithmetic mean. "g.mean": take the geometric mean. "median": take the median. "wa.mean", "wg.mean": weighted arithmetic/geometric mean. "fzt": Fisher Z-transform. Weights are the total information content shared between aligned columns.

Details

See compare_motifs() for more info on comparison parameters.

To replicate the internal **universalmotif** DB scores, run make_DBscores() with the default settings. Note that this will be a slow process.

Arguments widths, method, normalise.scores and score.strat are vectorized; all combinations will be attempted.

Value

A DataFrame with score distributions for the input database. If more than one make_DBscores() run occurs (i.e. args method, normalise.scores or score.strat are longer than 1), then the function args are included in the metadata slot.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Khan A, Fornes O, Stigliani A, Gheorghe M, Castro-Mondragon JA, van der Lee R, Bessy A, Cheneby J, Kulkarni SR, Tan G, Baranasic D, Arenillas DJ, Sandelin A, Vandepoele K, Lenhard B, Ballester B, Wasserman WW, Parcy F, Mathelier A (2018). "JASPAR 2018: update of the open-access database of transcription factor binding profiles and its web framework." *Nucleic Acids Research*, **46**, D260-D266.

See Also

compare_motifs()

Examples

```
## Not run:
library(MotifDb)
motifs <- convert_motifs(MotifDb[1:100])
scores <- make_DBscores(motifs, method = "PCC")
compare_motifs(motifs, 1:100, db.scores = scores)

## End(Not run)
```

---

mask_complexity *Mask low-complexity regions of DNA and RNA sequences.*

Description

Homopolymers, microsatellites and other short repeats give motifs rich in the same letters many overlapping hits, which can inflate scanning results and enrichment. mask_complexity() finds these regions with the symmetric DUST algorithm and replaces them with letter, so that scan_sequences(), enrich_motifs() and the like skip them.

Usage

```
mask_complexity(seqs, window.size = 64, threshold = 20,
  lowercase = FALSE, letter = "-", return.ranges = FALSE, nthreads = 1)
```

Arguments

seqs	XStringSet DNA or RNA sequences. Read a soft-masked FASTA file with Biostrings::readBStringSet() to keep its lowercase letters, since DNAStringSet objects have none.
window.size	integer(1) The DUST window in bases.
threshold	integer(1) The score above which a region is masked. A higher threshold masks less.
lowercase	logical(1) Also mask lowercase letters, as soft-masked genomes mark their repeats.
letter	character(1) The masking letter. It must be valid for the class of seqs, such as "-" or "N" for DNAStringSet.
return.ranges	logical(1) Return the regions as a GRanges object, which mask_ranges() takes, instead of the masked sequences. Unnamed sequences are named by their place in seqs. Requires the **GenomicRanges** package.
nthreads	numeric(1) Number of threads. Zero uses all available threads.

Details

Regions are found with symmetric DUST (Morgulis et al. 2006) as implemented in sdust (Li 2016), and match those of NCBI's dustmasker with the same window.size and threshold, 64 and 20 by default. Letters other than A, C, G, T and U, in either case, split a sequence into parts that are scored on their own, so that a gap of Ns or an earlier mask does not join the sequence either side of it; dustmasker instead scores across them, and caps the window at 64.

When masking sequences before an enrichment test, mask the targets and the background in the same way, and if the background is made by shuffling the targets, mask after shuffling, or the masking letters will be shuffled into the background.

A BStringSet given with lowercase = TRUE comes back as a BStringSet, which can be turned into a DNAStringSet for scanning with Biostrings::DNAStringSet().

Value

The masked sequences, of the same class as seqs, or with return.ranges = TRUE a GRanges object of the regions.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Li H (2016). "Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences." *Bioinformatics*, **32**, 2103-2110.

Morgulis A, Gertz EM, Schaffer AA, Agarwala R (2006). "A fast and symmetric DUST implementation to mask low-complexity DNA sequences." *Journal of Computational Biology*, **13**, 1028-1040.

See Also

mask_seqs(), mask_ranges(), sequence_complexity(), scan_sequences()

Examples

```
seqs <- Biostrings::DNAStringSet(c(
  s1 = "GGTTGTACTGTACAGGAACTTATAGAATAATTCTGGATGTATATATATATATATATATATATATATATTCCAAACG",
  s2 = "ACGTACGTAAAAAAAAAAAAAAAAAAAAAAAAAACGTTGCAGTCGATCGATGCATGCAT"))
mask_complexity(seqs)

## Fewer, stronger regions with a higher threshold, as ranges:
if (requireNamespace("GenomicRanges", quietly = TRUE)) {
  mask_complexity(seqs, threshold = 30, return.ranges = TRUE)
}

## Soft-masked letters too:
soft <- Biostrings::BStringSet(c(s3 = "acgtacgtCAGTCGATCGATGCATGCATTTGACCAGT"))
mask_complexity(soft, lowercase = TRUE)
```

---

match_bkg *Sample composition-matched background sequences from a universe.*

Description

For each sequence in `sequences`, `match_bkg()` draws one or more background sequences from `universe` that share its GC fraction and length. This is the HOMER-style binned matching used by motif enrichment tools to control for sequence-composition biases (GC, length) that would otherwise inflate enrichment estimates relative to shuffle-only nulls. Matching can be extended to additional external per-sequence covariates (e.g. ChIP signal strength, conservation, mappability) via `covariates`, which simply add more binned axes to the grid.

Usage

```
match_bkg(sequences, universe = NULL, genome = NULL, exclude = NULL,
  n.candidates = NULL, chromosomes = NULL, n.per.target = 1L,
  n.bins.gc = 20L, n.bins.length = 10L, covariates = NULL,
  universe.covariates = NULL, n.bins.covariates = 10L,
  bin.type.covariates = "quantile", unique = TRUE,
  return.indices = FALSE)
```

Arguments

`sequences`	`XStringSet`. Target (positive) sequences. DNA or RNA only – GC matching is undefined for other alphabets.
`universe`	`XStringSet` or `NULL`. Pool of candidate background sequences. Must share `sequences`'s alphabet and be at least `n.per.target * length(sequences)` long when `unique = TRUE`. Mutually exclusive with `genome`: supply exactly one of the two.
`genome`	A `BSgenome` object, or `NULL`. When supplied, the universe is built internally by sampling `n.candidates` random genomic windows whose widths match the target width distribution, optionally excluding any window that overlaps `exclude`. This is the typical shortcut for ChIP-seq style backgrounds, where the universe would otherwise be a manual `sample()` + `subsetByOverlaps()` + `getSeq()` boilerplate. Requires the `BSgenome`, `GenomicRanges`, and `IRanges` packages.

exclude — GRanges or NULL. Only honoured when genome is non-NULL. Random windows that overlap any range in exclude are dropped before sampling. Typical use: pass the target peak set so the background doesn't sample from peak regions.

n.candidates — integer(1) or NULL. Only honoured when genome is non-NULL. Number of random windows to sample before any exclude filtering. Default NULL picks max(10000, length(sequences) * 10).

chromosomes — character or NULL. Only honoured when genome is non-NULL. Names of seqlevels to sample from. Default NULL uses GenomeInfoDb::standardChromosomes(genome) when available (typically the autosomes plus the sex chromosomes), falling back to all seqlevels of the genome.

n.per.target — integer(1). Number of background sequences to draw per target. Default 1L.

n.bins.gc — integer(1). Number of equal-width GC bins. Default 20L.

n.bins.length — integer(1). Number of quantile-based length bins. Default 10L.

covariates — NULL, or a data.frame / matrix / numeric vector of external per-sequence covariates to additionally match on, with one row per sequence in sequences and one numeric column per covariate. A bare numeric vector is treated as a single covariate named "covariate1". Each covariate becomes an extra binned matching axis. Column names must not be "i", "gc" or "length" (these collide with the return.indices columns). Covariate values must be finite (no NA/NaN/Inf). Mutually exclusive with genome: internally-sampled genomic windows have no covariate values, so covariate matching requires an explicit universe paired with universe.covariates. Default NULL.

universe.covariates — NULL, or the covariate table for the universe sequences. Required (and only used) when covariates is supplied. Must have one row per universe sequence and the same column names, in the same order, as covariates. Default NULL.

n.bins.covariates — integer(1) applied to every covariate, or an integer vector named by covariate column (naming every covariate). Number of bins per covariate axis; each must be >= 2. Default 10L.

bin.type.covariates — character. How to bin each covariate axis: "quantile" (default, breaks from the universe-covariate quantiles, like length) or "equalwidth" (equal-width breaks over the combined target/universe range, like GC). A single value applies to every covariate, or supply a character vector named by covariate column. Quantile binning is rank-based, hence scale- and shift-invariant; no covariate normalisation is needed.

unique — logical(1). If TRUE (default), each universe sequence is drawn at most once. Falls back to with-replacement sampling (with a warning) if the universe is too small to honour uniqueness.

return.indices — logical(1). If TRUE, return a data.frame of per-pair match indices and composition values instead of the matched XStringSet. Default FALSE.

Details

Algorithm (HOMER-style):

1. Compute per-sequence GC fraction and length for sequences and universe (plus any supplied covariates).

2. Bin the universe into a grid: GC bins equal-width over [0, 1], length bins quantile-based on the universe widths, and one extra axis per covariate (quantile- or equal-width binned, see `bin.type.covariates`).
3. For each target, locate its joint bin and sample `n.per.target` universe sequences from that bin (without replacement when `unique = TRUE`). If the bin is empty or undersized, expand outward in Manhattan-distance rings.

Use the result as a `bkg.sequences` argument to `enrich_motifs_lite()` or `motif_finder()` for a composition-controlled null. `shuffle_sequences()` is a complementary alternative that randomises each input in place (preserves k-let composition); `match_bkg()` instead samples real sequences from a larger pool.

Value

If `return.indices = FALSE` (default): an `XStringSet` (same subclass as `sequences`) of length `length(sequences) * n.per.target`. The first `n.per.target` entries correspond to `sequences[[1]]`, the next to `sequences[[2]]`, etc. If `return.indices = TRUE`: a `data.frame` with columns `target.i`, `target.gc`, `target.length`, `universe.i`, `universe.gc`, `universe.length`, followed (when `covariates` is supplied) by a `target.<name>` / `universe.<name>` pair for each covariate.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Heinz S, Benner C, Spann N, et al. (2010). "Simple combinations of lineage-determining transcription factors prime cis-regulatory elements required for macrophage and B cell identities." *Molecular Cell*, **38**(4):576-589. doi:10.1016/j.molcel.2010.05.004.

See Also

`shuffle_sequences()`, `enrich_motifs_lite()`, `motif_finder()`, `plot_match_bkg()`

Examples

```
data(ArabidopsisPromoters)
target   <- ArabidopsisPromoters[1:10]
universe <- ArabidopsisPromoters[11:50]
bkg <- match_bkg(target, universe, n.per.target = 2)
## Use as a null background for enrichment / discovery
# enrich_motifs_lite(motifs, target, bkg.sequences = bkg)
# motif_finder(target, bkg.sequences = bkg)
plot_match_bkg(target, bkg)

## Additionally match on an external covariate (e.g. a ChIP signal
## score aligned to each sequence). Covariate matching needs an
## explicit universe + universe.covariates.
sig.t <- runif(length(target))
sig.u <- runif(length(universe))
idx <- match_bkg(target, universe,
                 covariates          = data.frame(signal = sig.t),
                 universe.covariates = data.frame(signal = sig.u),
                 return.indices      = TRUE)
plot_match_bkg(indices = idx)   # GC, length and signal panels
```

```
## Not run:
## With a BSgenome, sample the universe from random genomic windows
## whose widths match the targets:
library(BSgenome.Athaliana.TAIR.TAIR9)
bkg <- match_bkg(target, genome = Athaliana, n.per.target = 2)

## End(Not run)
```

`merge_motifs` *Merge motifs.*

Description

Aligns the motifs using `compare_motifs()`, then averages the motif PPMs. Currently the `multifreq` slot, if filled in any of the motifs, will be dropped. Only 0-order background probabilities will be kept. Motifs are merged one at a time, starting with the first entry in the list. For merging regular DNA/RNA motifs, see the simpler and faster `merge_motifs_lite()`.

Usage

```
merge_motifs(motifs, method = "ALLR", use.type = "PPM", min.overlap = 6,
  min.mean.ic = 0.25, tryRC = TRUE, relative_entropy = FALSE,
  normalise.scores = FALSE, min.position.ic = 0, score.strat = "sum",
  new.name = NULL)
```

Arguments

`motifs`	See `convert_motifs()` for acceptable motif formats.
`method`	`character(1)` One of PCC, EUCL, SW, KL, ALLR, BHAT, HELL, SEUCL, MAN, ALLR_LL, WEUCL, WPCC. See details.
`use.type`	`character(1)` One of `'PPM'` and `'ICM'`. The latter allows for taking into account the background frequencies if `relative_entropy = TRUE`. Note that `'ICM'` is not allowed when `method = c("ALLR", "ALLR_LL")`.
`min.overlap`	`numeric(1)` Minimum overlap required when aligning the motifs. Setting this to a number higher then the width of the motifs will not allow any overhangs. Can also be a number between 0 and 1, representing the minimum fraction that the motifs must overlap.
`min.mean.ic`	`numeric(1)` Minimum mean information content between the two motifs for an alignment to be scored. This helps prevent scoring alignments between low information content regions of two motifs. Note that this can result in some comparisons failing if no alignment passes the mean IC threshold. Use `average_ic()` to filter out low IC motifs to get around this if you want to avoid getting NAs in your output.
`tryRC`	`logical(1)` Try the reverse complement of the motifs as well, report the best score.
`relative_entropy`	`logical(1)` Change the ICM calculation affecting `min.position.ic` and `min.mean.ic`. See `convert_type()`.

`normalise.scores`
: `logical(1)` Favour alignments which leave fewer unaligned positions, as well as alignments between motifs of similar length. Similarity scores are multiplied by the ratio of aligned positions to the total number of positions in the larger motif, and the inverse for distance scores.

`min.position.ic`
: `numeric(1)` Minimum information content required between individual alignment positions for it to be counted in the final alignment score. It is recommended to use this together with `normalise.scores = TRUE`, as this will help punish scores resulting from only a fraction of an alignment.

`score.strat`
: `character(1)` How to handle column scores calculated from motif alignments. "sum": add up all scores. "a.mean": take the arithmetic mean. "g.mean": take the geometric mean. "median": take the median. "wa.mean", "wg.mean": weighted arithmetic/geometric mean. "fzt": Fisher Z-transform. Weights are the total information content shared between aligned columns.

`new.name`
: `character(1)`, `NULL` Instead of collapsing existing names (if `NULL`), assign a new one manually for the merged motif.

Details

See `compare_motifs()` for more info on comparison parameters.

If using a comparison metric where 0s are not allowed (`KL`, `ALLR`, `ALLR_LL`, `IS`), then pseudocounts will be added internally. These pseudocounts are only used for comparison and alignment, and are not used in the final merging step.

Note: `score.strat = "a.mean"` is NOT recommended, as `merge_motifs()` will not discriminate between two alignments with equal mean scores, even if one alignment is longer than the other.

Value

A single motif object. See `convert_motifs()` for available formats.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

`compare_motifs()`, `merge_motifs_lite()`, `merge_similar()`, `trim_motifs()`

Examples

```
## Not run:
library(MotifDb)
merged.motif <- merge_motifs(MotifDb[1:5])

## End(Not run)

m1 <- create_motif("TTAAACCCC", name = "1")
m2 <- create_motif("AACC", name = "2")
m3 <- create_motif("AACCCCGG", name = "3")
view_motifs(merge_motifs(c(m1, m2, m3)))
```

`merge_motifs_lite` *Merge a list of motifs into a single consensus motif.*

Description

`merge_motifs_lite()` is a faster minimalist counterpart to `merge_motifs()` that aligns motifs using `compare_motifs_lite()`'s alignment finder and averages their position-probability columns in a single shared coordinate frame.

Usage

```
merge_motifs_lite(motifs, min.overlap = 5L, RC = TRUE, new.name = NULL,
  weighted = FALSE, nthreads = 1)
```

Arguments

`motifs`	`list` of motifs. See `convert_motifs()` for accepted formats. DNA or RNA only.
`min.overlap`	`integer(1)`. Minimum overlap (in columns) for an alignment to be considered. Default `5L`. Forwarded to `compare_motifs_lite()`.
`RC`	`logical(1)`. If `TRUE` (default), also test reverse-complement alignments. Motifs whose best alignment hits the `-` strand of the anchor are reverse-complemented before averaging, so the merged motif is always returned in the anchor's orientation.
`new.name`	`character(1)` or `NULL`. Name for the merged motif. If `NULL` (default), set to `paste0("merged_", paste(names, collapse = "+"))`.
`weighted`	`logical(1)`. If `TRUE`, weight each input motif's column contribution by its `@nsites` slot. Default `FALSE` (simple unweighted mean). Motifs whose `@nsites` is missing or zero fall back to weight 1 in either mode.
`nthreads`	`numeric(1)`. Number of threads passed to `compare_motifs_lite()`. `nthreads = 0` uses all available threads.

Details

For each non-anchor motif, `compare_motifs_lite()` gives the best alignment offset, strand, and overlap against the anchor. Each aligned motif (reverse-complemented if its best alignment is `-` strand) is then placed in a unified column coordinate frame whose origin coincides with the anchor's first column. The output's column at frame position `p` is the (optionally `@nsites`-weighted) mean of the PPM columns from all input motifs that cover `p`; positions covered by only some motifs are still emitted (no automatic IC-trimming). Run `trim_motifs()` on the result if you want to shrink low-information edges.

The merged motif's `@bkg` is the unweighted mean of the input motifs' `@bkg` vectors (or uniform if missing). `@nsites` is the sum of input `@nsites`.

Value

A single `universalmotif` S4 object of type `"PPM"`.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

merge_motifs(), merge_similar_lite(), compare_motifs_lite(), trim_motifs()

Other lite motif functions: compare_motifs_lite(), enrich_motifs_lite(), merge_similar_lite(), motif_tree_lite(), scan_sequences_lite(), view_motifs_lite()

Examples

```
library(universalmotif)
m1 <- create_motif("TTGACATA", name = "a")
m2 <- create_motif("CTTGACAT", name = "b")
m3 <- create_motif("TGACATAT", name = "c")
merged <- merge_motifs_lite(list(m1, m2, m3))
merged
```

---

merge_similar *Identify and merge similar motifs within a collection of motifs (or simply cluster motifs).*

Description

Given a list of motifs, merge_similar() will identify similar motifs with compare_motifs(), and merge similar ones with merge_motifs(). For merging regular DNA/RNA motifs, see the simpler and faster merge_similar_lite().

Usage

```
merge_similar(motifs, threshold = 0.95, threshold.type = "score.abs",
  method = "PCC", use.type = "PPM", min.overlap = 6, min.mean.ic = 0,
  tryRC = TRUE, relative_entropy = FALSE, normalise.scores = FALSE,
  min.position.ic = 0, score.strat.compare = "a.mean",
  score.strat.merge = "sum", nthreads = 1, return.clusters = FALSE)
```

Arguments

motifs	See convert_motifs() for acceptable motif formats.
threshold	numeric(1) The minimum (for similarity metrics) or maximum (for distance metrics) threshold score for merging.
threshold.type	character(1) Type of score used for thresholding. Currently unused.
method	character(1) One of PCC, EUCL, SW, KL, BHAT, HELL, SEUCL, MAN, WEUCL, WPCC. See compare_motifs(). (The ALLR and ALLR_LL methods cannot be used for distance matrix construction.)
use.type	character(1) One of 'PPM' and 'ICM'. The latter allows for taking into account the background frequencies if relative_entropy = TRUE. Note that 'ICM' is not allowed when method = c("ALLR", "ALLR_LL").

`min.overlap` numeric(1) Minimum overlap required when aligning the motifs. Setting this to a number higher then the width of the motifs will not allow any overhangs. Can also be a number between 0 and 1, representing the minimum fraction that the motifs must overlap.

`min.mean.ic` numeric(1) Minimum mean information content between the two motifs for an alignment to be scored. This helps prevent scoring alignments between low information content regions of two motifs. Note that this can result in some comparisons failing if no alignment passes the mean IC threshold. Use `average_ic()` to filter out low IC motifs to get around this if you want to avoid getting NAs in your output.

`tryRC` logical(1) Try the reverse complement of the motifs as well, report the best score.

`relative_entropy` logical(1) Change the ICM calculation affecting `min.position.ic` and `min.mean.ic`. See `convert_type()`.

`normalise.scores` logical(1) Favour alignments which leave fewer unaligned positions, as well as alignments between motifs of similar length. Similarity scores are multiplied by the ratio of aligned positions to the total number of positions in the larger motif, and the inverse for distance scores.

`min.position.ic` numeric(1) Minimum information content required between individual alignment positions for it to be counted in the final alignment score. It is recommended to use this together with `normalise.scores = TRUE`, as this will help punish scores resulting from only a fraction of an alignment.

`score.strat.compare` character(1) The `score.strat` parameter used by `compare_motifs()`. For clustering purposes, the `"sum"` option cannot be used.

`score.strat.merge` character(1) The `score.strat` parameter used by `merge_motifs()`. As discussed in `merge_motifs()`, the `"sum"` option is recommended over `"a.mean"` to maximize the overlap between motifs.

`nthreads` numeric(1) Run `compare_motifs()` in parallel with `nthreads` threads. `nthreads = 0` uses all available threads.

`return.clusters` logical(1) Return the clusters instead of merging.

Details

See `compare_motifs()` for more info on comparison parameters, and `merge_motifs()` for more info on motif merging.

Value

See `convert_motifs()` for available output formats.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

`compare_motifs()`, `merge_motifs()`, `merge_similar_lite()`

Examples

```
## A clear cluster of three related motifs plus one unrelated motif:
m1 <- create_motif("TTGACATA", name = "a")
m2 <- create_motif("CTTGACAT", name = "b")
m3 <- create_motif("GGGGCCCC", name = "c")
motifs <- list(m1, m2, m3)
length(motifs)
motifs <- merge_similar(motifs)
length(motifs)

## Not run:
## Collapsing redundancy within a larger database:
library(MotifDb)
motifs <- filter_motifs(MotifDb, family = "bHLH")[1:50]
length(motifs)
motifs <- merge_similar(motifs)
length(motifs)

## End(Not run)
```

---

merge_similar_lite *Cluster similar motifs by significance and merge each cluster.*

Description

`merge_similar_lite()` is a faster minimalist counterpart to `merge_similar()` that builds a pairwise similarity-significance graph using `compare_motifs_lite()`'s empirical or parametric null q-values, finds connected components on that graph, and collapses each component (cluster) into a single merged motif via `merge_motifs_lite()`.

Usage

```
merge_similar_lite(motifs, qvalue = 0.01, null = c("empirical",
  "parametric"), min.overlap = 5L, RC = TRUE, bkg = NULL,
  weighted = FALSE, return.clusters = FALSE, nthreads = 1)
```

Arguments

`motifs`	`list` of motifs. DNA or RNA only.
`qvalue`	`numeric(1)`. BH-adjusted q-value cutoff for linking two motifs in the significance graph. Default `0.01`.
`null`	`character(1)`. `"empirical"` (default) or `"parametric"`, forwarded to `compare_motifs_lite()`.
`min.overlap`	`integer(1)`. Forwarded to `compare_motifs_lite()` / `merge_motifs_lite()`. Default `5L`.
`RC`	`logical(1)`. If `TRUE` (default), test reverse-complement alignments.
`bkg`	`numeric(4)` or `NULL`. Background frequencies, forwarded to `compare_motifs_lite()`. `NULL` uses uniform.
`weighted`	`logical(1)`. Forwarded to `merge_motifs_lite()`. Default `FALSE`.

return.clusters
: logical(1). If TRUE, return a universalmotif_df of cluster assignments instead of merged motifs. Default FALSE.

nthreads
: numeric(1). Threads forwarded to compare_motifs_lite() and merge_motifs_lite(). nthreads = 0 uses all available threads.

Details

Unlike merge_similar() (which builds a distance matrix from raw similarity scores and applies hierarchical clustering at an absolute score cutoff) this function clusters on statistical significance: two motifs are linked if their pairwise q-value meets the chosen qvalue cutoff. This matches the STAMP / TOMTOM-clustering semantics ("group all motifs that are significantly similar"), is deterministic, has no linkage-method choice to expose, and expresses the cutoff in q-value units rather than abstract distance.

The significance graph is built from a symmetrised q-value matrix: Qsym[i,j] = pmin(Q[i,j], Q[j,i]). compare_motifs_lite()'s per-query BH adjustment makes its native q-value matrix non-symmetric; the min- symmetrisation defines a pair as linked if either direction meets the cutoff (more permissive of the two, which is appropriate when the goal is to group similar motifs).

Clustering is by connected components on the resulting graph, implemented with a small union-find. Each component (size >= 2) becomes a single merged motif via merge_motifs_lite(); singletons (size == 1) pass through unchanged.

Value

If return.clusters = FALSE (default): a list of universalmotif S4 objects with similar motifs collapsed (the list is shorter than the input whenever any merging occurred). Singletons pass through unchanged.

If return.clusters = TRUE: a universalmotif_df (as produced by to_df()) describing the input motifs – one row per input motif, with the motif column holding the motif objects and name their names – augmented with two extra columns: motif.i (1-based input index) and cluster (1-based cluster id; motifs in the same cluster share an id).

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

merge_similar(), merge_motifs_lite(), compare_motifs_lite()

Other lite motif functions: compare_motifs_lite(), enrich_motifs_lite(), merge_motifs_lite(), motif_tree_lite(), scan_sequences_lite(), view_motifs_lite()

Examples

```
## A clear cluster of 3 related motifs + 1 unrelated motif
m1 <- create_motif("TTGACATA", name = "a")
m2 <- create_motif("CTTGACAT", name = "b")
m3 <- create_motif("TGACATAT", name = "c")
m4 <- create_motif("GGGCCCCC", name = "unrelated")
merged <- merge_similar_lite(list(m1, m2, m3, m4), qvalue = 0.05)
length(merged)  # 2: one merged + the unrelated singleton
```

motif_coocc *Find significantly co-occurring motif pairs in a set of sequences.*

Description

motif_coocc() tests every motif pair for over-co-occurrence across a set of sequences using a one-sided Fisher's exact test on the 2x2 contingency table of per-sequence presence/absence. The result is the long-format pair table, BH-corrected across all tested pairs. Optionally, when max.distance is non-NULL, the function also reports two descriptive spatial columns, both.clustered (number of co-occurring sequences with a within-max.distance (A, B) hit pair) and median.distance (the median nearest-pair spacing), so heterodimer-like arrangements can be flagged. The Fisher p-value itself is always computed on the unfiltered 2x2 (the spatial filter is informational, not a test).

Usage

```
motif_coocc(motifs, sequences = NULL, hits = NULL, n.sequences = NULL,
  pvalue = 1e-04, RC = TRUE, no.overlaps = FALSE,
  no.overlaps.by = c("pvalue", "score"), max.distance = NULL,
  min.coocc = 1L, pseudocount = 0L, self.pairs = FALSE, nthreads = 1L)
```

Arguments

motifs	See convert_motifs() for accepted motif formats. A single motif or a list of motifs. Required in both paths to provide motif names and to dimension the pair table.
sequences	XStringSet or NULL. If supplied, the function scans internally via scan_sequences_lite(); DNA/RNA only.
hits	data.frame, GRanges, or NULL. Precomputed hit table with motif.i and sequence.i columns. If supplied, sequences is ignored and n.sequences must be provided. Any alphabet accepted on this path.
n.sequences	integer(1) or NULL. Total number of host sequences (needed to fill the "neither" cell of the 2x2 when using the precomputed-hits path).
pvalue	numeric(1). Per-hit p-value threshold for the internal scan. Default 1e-4. Ignored on the hit-table path.
RC	logical(1). Scan the reverse complement too? Default TRUE. Ignored when a hits table is provided.
no.overlaps	logical(1). Passed to scan_sequences_lite() on the internal-scan path: if TRUE, overlapping hits of the *same* motif are collapsed to one (greedy; see dedup_hits()) before co-occurrence is computed. This does not change the presence-based Fisher test (a sequence either contains a motif or not, however many overlapping copies it has), so it only affects the descriptive spatial columns reported under max.distance, where it stops tandem/overlapping matches of one motif being counted as separate instances. Overlaps *between* the two motifs of a pair are always kept, so heterodimer footprints are preserved. Ignored (with a warning) when a precomputed hits table is supplied. Default FALSE.
no.overlaps.by	character(1). Tie-break priority when no.overlaps = TRUE: "pvalue" (default) keeps the lowest-p hit, "score" the highest-scoring. Passed to scan_sequences_lite().

`max.distance` `integer(1)` or `NULL`. If non-`NULL`, two extra descriptive columns are added to the output: `both.clustered` (subset of co-occurring sequences with a within-`max.distance` (A, B) hit pair) and `median.distance` (median of those nearest-pair distances). The Fisher p-value is unchanged; it always tests "do A and B occur in the same sequences more often than chance?" on the unfiltered 2x2. Requires a `start` column on the hit table.

`min.coocc` `integer(1)`. Pairs with fewer than `min.coocc` co-occurring sequences are tagged with NA p- and q-values (not tested). Default `1L` (only test pairs that co-occur at least once).

`pseudocount` `numeric(1)`. A non-negative whole number added to every cell of the 2x2 contingency before the Fisher test (matches the convention from `enrich_motifs_lite()`). Default `0L`.

`self.pairs` `logical(1)`. If `TRUE`, include `(i, i)` rows, usually only meaningful for motifs that bind as homodimers. These rows are not tested, since a motif's 2x2 table with itself has empty off-diagonal cells: `both` counts the sequences with hits at two or more positions and `a_only` those with hits at only one, and hits at the same position on both strands count once. Default `FALSE`.

`nthreads` `integer(1)`. Threads for the internal scan (when `sequences` is supplied). The pairwise co-occurrence test itself is vectorised R (one `phyper()` call over all pairs) and does not use threads.

Details

Two input paths are supported:

1. Internal scan (default): pass `motifs` + `sequences` and `motif_coocc()` will call `scan_sequences_lite()` with the given `pvalue` and `RC` arguments to build the hit table. DNA/RNA only (the restriction comes from `scan_sequences_lite()`).
2. Precomputed hits: pass `motifs` + `hits` + `n.sequences`. `hits` is a `data.frame` or `GRanges` with `motif.i` and `sequence.i` columns (the native return shape of `scan_sequences()` / `scan_sequences_lite()`), optionally `start` when `max.distance` is set. This path accepts any motif alphabet; once a hit table exists, co-occurrence is pure set arithmetic on integer indices and no sequence bytes are read.

Value

`data.frame` with columns:

- `motif_a`, `motif_b`: motif names (1-based indexing into the supplied `motifs` list; deduplicated names if duplicates exist).
- `a_only`, `b_only`, `both`, `neither`: 2x2 contingency cells.
- `odds_ratio`: sample odds ratio of the (pseudocount-adjusted) 2x2, `(both * neither) / (a_only * b_only)`; `Inf` when an off-diagonal cell is zero. `NA` when `both` < `min.coocc`.
- `pvalue`: one-sided Fisher's exact p-value (alternative = "greater"). NA when `both` < `min.coocc`.
- `qvalue`: BH-corrected q-value across all tested pairs.
- `both.clustered` (spatial mode only): subset of `both` where a within-`max.distance` (A, B) hit pair exists.
- `median.distance` (spatial mode only): median of those nearest-pair distances; `NA` if no clustered pair. Rows are sorted by q-value ascending; NA-pvalue rows go last.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Heinz S, Benner C, Spann N, et al. (2010). "Simple combinations of lineage-determining transcription factors prime cis-regulatory elements required for macrophage and B cell identities." *Molecular Cell*, **38**(4):576-589. doi:10.1016/j.molcel.2010.05.004. (HOMER's pairwise-Fisher co-occurrence convention.)

Whitington T, Frith MC, Johnson J, Bailey TL (2011). "Inferring transcription factor complexes from ChIP-seq data." *Nucleic Acids Research*, **39**(15):e98. doi:10.1093/nar/gkr341. (SpaMo, the spatial-distance motif-pair analysis.)

See Also

`scan_sequences_lite()`, `enrich_motifs_lite()`, `motif_peaks()`

Examples

```
## Not run:
library(universalmotif)
data(ArabidopsisMotif)
seqs <- create_sequences("DNA", seqnum = 200, seqlen = 500)
## Toy: implant two copies of the same motif jointly in all seqs.
set.seed(1)
imp <- implant_motifs(list(ArabidopsisMotif, ArabidopsisMotif),
                      seqs, n.per.seq = 1, return.indices = TRUE)
co <- motif_coocc(list(ArabidopsisMotif, ArabidopsisMotif),
                  imp$sequences, pvalue = 1e-3)
co

## End(Not run)
```

`motif_finder` *Discover* de novo *motifs in a set of sequences.*

Description

`motif_finder()` is a minimalist *de novo* motif discovery function. The pipeline works through a configurable range of motif widths; at each width it enumerates the over-represented k-mer seeds (using a Fisher's exact test on per-sequence presence against a background set), aligns the Hamming-1 neighbours of the best seed to form a PPM, refines that PPM over two re-scan passes, accepts the motif if its Fisher's exact p-value passes `stop.pvalue`, and then masks the covered positions and repeats, up to `nmotifs` times per width. Once every width is done, the motifs are deduplicated across widths by coverage overlap, BH-adjusted, IC-trimmed at the flanks, and returned in p-value order.

Usage

```
motif_finder(sequences, bkg.sequences = NULL, min.width = 6L,
  max.width = 15L, nmotifs = 10L, hit.pvalue = 1e-04,
  stop.pvalue = 0.001, qvalue = 0.05, dedup.overlap = 0.5, RC = TRUE,
  shuffle.k = 2L, rng.seed = sample.int(1e+06, 1), pseudocount = 1L,
  nthreads = 1)
```

Arguments

sequences	XStringSet. DNA or RNA target sequences.
bkg.sequences	XStringSet or NULL. Background sequences. If NULL (default), target sequences are shuffled k-let-conserving via shuffle_sequences() with k = shuffle.k.
min.width	integer(1). Minimum motif width. Default 6L.
max.width	integer(1). Maximum motif width. Default 15L. Hard ceiling 30.
nmotifs	integer(1). Maximum number of motifs to discover per width before moving on. Default 10L.
hit.pvalue	numeric(1). P-value threshold for scoring a single hit during scanning. Default 1e-4.
stop.pvalue	numeric(1). Per-motif Fisher's exact p-value below which a candidate motif is accepted. Default 1e-3.
qvalue	numeric(1). Final BH-adjusted q-value cutoff. Motifs with qvalue above this are dropped. Default 0.05.
dedup.overlap	numeric(1). Cross-width dedup overlap fraction. When the intersection of two motifs' positional coverage divided by the smaller motif's coverage exceeds this, the higher-p-value motif is dropped. Default 0.5.
RC	logical(1). If TRUE (default), scan both strands.
shuffle.k	integer(1). K-let size for the background shuffle (only used when bkg.sequences = NULL). Default 2L.
rng.seed	integer(1). RNG seed for background shuffling. Default sample.int(1e6, 1); set explicitly for reproducible runs.
pseudocount	integer(1). Pseudocount used when converting per-position counts to log-odds scores. Default 1L.
nthreads	numeric(1). Number of threads. Parallelism is at the width level. nthreads = 0 uses all available threads.

Details

Algorithm and defaults are based on the STREME algorithm (Bailey 2021): seed enumeration via word counting with per-sequence Fisher's exact ranking, iterative PPM refinement on positive sequences, per-motif Fisher's exact significance against a shuffled or supplied background, and position masking between discoveries.

Each motif is the best of the thousands of words tried at its width, so its P-value is corrected for that search, and the q-values count every word tried at every width. Without these corrections, random sequences compared with shuffled ones give apparent motifs with very small q-values.

Motifs with qvalue > qvalue are dropped from the result. To see all discovered motifs regardless of significance, set qvalue = 1.

Value

A universalmotif_df (see to_df()) with one row per discovered motif, sorted by pvalue ascending. The standard to_df() columns (name, altname, family, organism, consensus, alphabet, type, nsites, pval, eval, motif, ...) are populated from each discovered motif. The discovery-specific stats are carried as additional columns:

- rank: 1-based discovery rank
- width: motif width (post IC-trim)
- seqs_pos: positive sequences with >=1 hit
- seqs_neg: background sequences with >=1 hit
- sites_pos: total target hits
- sites_neg: total background hits
- n_pos: number of positive sequences
- n_neg: number of background sequences
- pvalue: Fisher's exact one-sided p-value, multiplied by the number of words tried in the search that found the motif and capped at 1
- qvalue: Benjamini-Hochberg q-value, counting every word tried at every width as a test

Use to_list() on the returned object to recover a plain list of universalmotif S4 objects.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Bailey TL (2021). "STREME: accurate and versatile sequence motif discovery." *Bioinformatics*, **37**(18), 2834-2840. doi:10.1093/bioinformatics/btab203.

See Also

scan_sequences_lite(), enrich_motifs_lite(), shuffle_sequences(), create_motif(), to_df(), to_list()

Examples

```
## Not run:
library(universalmotif)
library(Biostrings)
set.seed(1)
planted <- DNAString("TTGACATA")
seqs <- create_sequences(seqnum = 100, seqlen = 200, rng.seed = 1)
## Plant the motif at random positions in 80% of sequences:
for (i in seq_len(80)) {
  pos <- sample.int(200 - 8, 1)
  seqs[[i]][pos:(pos + 7)] <- planted
}
motifs <- motif_finder(seqs, qvalue = 1, rng.seed = 1)
to_list(motifs)[[1]]

## End(Not run)
```

`motif_peaks` *CentriMo-style positional enrichment of motif hits.*

Description

`motif_peaks()` tests whether motif hits cluster non-uniformly along input sequences. It implements an analytical CentriMo-style test (Bailey & Machanick 2012): under the null that hit positions are uniformly distributed over `[1, seq.length]`, the count of hits falling inside a candidate window follows a binomial distribution. For each motif and each candidate window, a one-sided binomial p-value is computed; the best-scoring window per motif is reported, Bonferroni-corrected over the number of windows tested, then BH- adjusted across motifs.

Usage

```
motif_peaks(hits, seq.length = NULL, mode = c("central", "local"),
  seq.strand = NULL, qvalue = 0.1, min.window = 10L, max.window = NULL,
  window.step = 10L, position.step = 5L, nthreads = 1)
```

Arguments

`hits`	Motif hit table from `scan_sequences()` or `scan_sequences_lite()`. Accepted as either a `data.frame` or a `GRanges`. Required columns / metadata: `motif`, `score`, `start`, and either `end` (scan_sequences_lite) or `stop` (scan_sequences). For `data.frame` input, a `sequence` / `sequence.i` column identifies which sequence each hit belongs to. For `GRanges` input, the sequence is taken from `seqnames()`.
`seq.length`	`integer(1)`. Common length of the input sequences (in bases). Required for `data.frame` input. For `GRanges` input, taken from `seqlengths(hits)` if not provided.
`mode`	`character(1)`. `"central"` (default) tests only centre-of-sequence windows; `"local"` also varies the window centre across the sequence.
`seq.strand`	`NULL` or a named `character` vector mapping each sequence id (the `sequence` / `seqnames()` values in `hits`) to `"+"` or `"-"`. When supplied, hit positions on `"-"` sequences are reflected into the feature 5'->3' frame (`seq.length + 1 - center`), so that off-centre signals at the same feature-relative position reinforce across strands instead of splitting into mirror-image peaks. This only changes `mode = "local"` results; `mode = "central"` is symmetric about the centre and so is unaffected (the reflection is still applied, so the reported positions read in feature frame). Unnecessary when sequences were extracted with `Biostrings::getSeq()` from stranded ranges, since those are already oriented. Default `NULL` (strand-agnostic, the original behaviour).
`qvalue`	`numeric(1)`. BH-adjusted q-value cutoff for the final reported motifs. Default `0.1`. Set to 1 to return every motif.
`min.window, max.window`	`integer(1)`. Bounds of the window widths to test. `max.window = NULL` (default) uses `floor(seq.length / 2)`. `min.window` defaults to `10L`.
`window.step`	`integer(1)`. Step between consecutive window widths. Default `10L`.
`position.step`	`integer(1)`. Step between consecutive window centres in `mode = "local"`. Ignored in `mode = "central"`. Default `5L`.

nthreads numeric(1). Number of threads. nthreads = 0 uses all available cores. Default 1.

Details

Two modes are supported. mode = "central" (default) tests only windows centred on seq.length / 2, which is the appropriate hypothesis for centred input sequences (e.g. ChIP-seq peak summits). mode = "local" additionally varies the window centre, scanning the whole sequence for any positionally-enriched region; this is more permissive and uses a larger Bonferroni correction.

All sequences are assumed to be of equal length; the function does not enforce centring on a biological reference, but its results are only meaningful if such centring is in place upstream.

Value

A data.frame with one row per motif passing the q-value cutoff, sorted by pvalue ascending. Columns: motif, motif.i, mode, seq.length, nhits, best.window, best.center, hits.in, hits.out, expected.in, enrichment, log2.enrichment, pvalue, qvalue, centers (list-column of integer vectors of hit centres, one entry per sequence after best-hit-per-sequence dedup; consumed by plot_motif_peaks()).

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Bailey TL, Machanick P (2012). "Inferring direct DNA binding from ChIP-seq." *Nucleic Acids Research*, **40**(17):e128. doi:10.1093/nar/gks433.

See Also

scan_sequences_lite(), plot_motif_peaks()

Examples

```
## Pre-centred sequences (e.g. ChIP-seq peaks centred on their summit):
library(Biostrings)
set.seed(1)
seqs <- create_sequences(seqnum = 200, seqlen = 500, rng.seed = 1)
planted <- DNAString("TTGACATA")
for (i in seq_len(150)) {
  pos <- 246L + sample.int(8, 1) - 1L
  subseq(seqs[[i]], start = pos, width = 8) <- planted
}
m <- create_motif("TTGACATA", name = "test")
## scan_sequences() returns a table that motif_peaks() reads directly;
## pass seq.length since the data.frame does not carry it.
hits <- scan_sequences(m, seqs, threshold = 1e-3, threshold.type = "pvalue")
motif_peaks(as.data.frame(hits), seq.length = 500)

## Enrichment near a set of genomic coordinates. Given point
## coordinates in a genome (peak summits, TSSs, SNPs, and so on),
## expand each to a fixed-width window centred on the coordinate and
## scan those windows. "Enriched near the anchor" is then exactly the
## central hypothesis motif_peaks() already tests, so nothing beyond
```

```
## extracting the windows is required.
## Not run:
library(universalmotif)
library(GenomicRanges)
library(BSgenome.Athaliana.TAIR.TAIR9)

## A set of anchor coordinates (here, arbitrary points on Chr1).
anchors <- GRanges("Chr1",
                   IRanges(start = seq(1e5, 5e5, by = 1e4), width = 1))

## Expand to fixed 500 bp windows centred on each anchor. The
## equal-width filter matters because motif_peaks() compares positions
## across sequences, so a window clipped at a chromosome end would
## misplace its centre. For stranded anchors (e.g. TSSs), give
## `anchors` a strand and getSeq() will orient each window
## consistently up- or downstream of the anchor.
windows <- resize(anchors, width = 500, fix = "center")
windows <- trim(windows)
windows <- windows[width(windows) == 500]

seqs <- getSeq(Athaliana, windows)
names(seqs) <- paste0("anchor_", seq_along(seqs))

m    <- create_motif("TTGACATA", name = "example")
hits <- scan_sequences_lite(m, seqs, pvalue = 1e-3, return.granges = TRUE)
## seq.length is inferred from seqlengths() here, but pass it
## explicitly when the hit table doesn't carry equal seqlengths.
motif_peaks(hits, seq.length = 500)

## End(Not run)
```

`motif_proximity` *Test for motif enrichment near a set of genomic anchors.*

Description

`motif_proximity()` tests whether a motif's hits sit closer to a set of anchor coordinates than expected by chance, genome-wide. It is the genome-wide sibling of `motif_peaks()`: where `motif_peaks()` asks whether hits cluster in a central window of equal-length sequences, `motif_proximity()` asks whether hits cluster near anchors scattered across whole chromosomes. The default test mirrors the CentriMo-style binomial of `motif_peaks()`, but the success probability is the fraction of the scanned genome lying within a distance of an anchor rather than the fraction of a sequence lying in a window.

Usage

```
motif_proximity(hits = NULL, anchors, motifs = NULL, sequences = NULL,
  universe = NULL, method = c("binomial", "permutation", "ks"),
  count = c("hits", "anchors"), orientation = c("undirected", "upstream",
  "downstream"), scan.pvalue = 1e-04, RC = TRUE, dist.thresholds = NULL,
  n.perm = 1000L, perm.per.chrom = TRUE, ignore.strand = TRUE,
  qvalue = 0.1, nthreads = 1)
```

Arguments

hits	Motif hits from scan_sequences_lite() or scan_sequences(), as a GRanges (preferred) or a data.frame, carrying genomic coordinates plus motif and score. Supply this *or* (motifs + sequences), not both. data.frame input also requires universe (its seqlengths are unknown).
anchors	GRanges. The anchor coordinates to test proximity to. Anchor width matters: the near zone is the anchor widened by the distance on each side, so wide anchors enlarge the within-distance fraction and collapse the resolution of the test (see Caveats). For a point hypothesis (summits, TSSs, SNPs, motif centres) use 1 bp anchors, reducing regions first with e.g. GenomicRanges::resize(anchors, width = 1, fix = "center").
motifs, sequences	Alternative to hits: motifs (any format accepted by convert_motifs()) and the sequences to scan. sequences is an XStringSet (the scanned chromosomes) or a BSgenome; for a BSgenome only the chromosomes carrying anchors are extracted and scanned. Scanning is done internally with scan_sequences_lite().
universe	GRanges or NULL. The regions actually scanned (the null support). NULL (default) derives whole chromosomes from seqlengths(hits). **Pass this explicitly for partial or masked scans**, otherwise the test is anti-conservative.
method	character(1). "binomial" (default), "permutation", or "ks". See Details.
count	character(1). "hits" (default) or "anchors". See Details.
orientation	character(1). "undirected" (default; symmetric near zone) or "upstream" / "downstream" for a strand-aware, directional near zone built with GenomicRanges::flank(). Directional modes require stranded anchors and are not available for method = "ks".
scan.pvalue	numeric(1). Hit P-value threshold forwarded to scan_sequences_lite() when scanning internally. Default 1e-4.
RC	logical(1). Scan both strands when scanning internally. Default TRUE.
dist.thresholds	integer or NULL. Candidate distances d to test. NULL (default) builds a log-spaced grid capped at the smallest chromosome. Ignored by method = "ks".
n.perm	integer(1). Number of permutations for method = "permutation". Default 1000.
perm.per.chrom	logical(1). Randomise anchors within their own chromosome (TRUE, default, more conservative) or across the whole universe.
ignore.strand	logical(1). Match hits to anchors ignoring strand. Default TRUE. The near-zone *geometry* is always unstranded.
qvalue	numeric(1). BH q-value cutoff for the reported motifs. Default 0.1. Set to 1 to return every motif.
nthreads	numeric(1). Threads for internal scanning and the per-motif loop. 0 uses all cores. Default 1. For method = "permutation", exact reproducibility needs nthreads = 1 together with a prior set.seed(); parallel workers draw independent streams that are not bit-reproducible unless RNGkind("L'Ecuyer-CMRG") is in effect.

Details

Three nulls are available via method:

`"binomial"` (default) For each candidate distance `d`, the "near zone" is the set of genomic positions within `d` of any anchor; `f_d` is its fraction of the universe `U`. Under the null that hits are uniformly distributed over `U`, the count `k` of hits inside the near zone follows `Binomial(N, f_d)`. The best `d` per motif is reported, Bonferroni-corrected over the candidate distances, then BH-adjusted across motifs. Fast, but the uniform-hit null cannot correct composition bias.

`"permutation"` The anchors (not the hits) are randomised within `U` `n.perm` times; the binomial z-score is recomputed each time to build an empirical null. Randomising the anchors keeps the real hits (and their composition-driven clustering) fixed, so the test asks whether hits are closer to the real anchors than to random ones, which absorbs composition and mappability bias. This is the recommended method when composition bias is a concern. Uses the global RNG, so call `set.seed()` beforehand for reproducible p-values (see `nthreads`).

`"ks"` A threshold-free one-sided Kolmogorov-Smirnov test of the per-hit distance-to-nearest-anchor against its analytical null CDF (`f` as a function of `d`). P-values are approximate under the integer-distance ties; `"permutation"` is the exact alternative. Only available for `count = "hits"` and `orientation = "undirected"`.

The counting unit is set by `count`. `"hits"` (default) collapses overlapping same-motif hits to one best-scoring representative (the genome-wide analogue of the best-hit-per-sequence convention of `motif_peaks()`) and counts hits in the near zone. `"anchors"` counts the fraction of anchors that have at least one nearby hit; it has no clean analytical null, so `method = "permutation"` is recommended and `method = "binomial"` falls back to a Poisson occupancy approximation (anti-conservative when hits cluster).

Each hit is placed at its middle position, the lower one for even widths, for counting and for distances, since the null treats hits as single positions; counting a hit whenever any part of it reached the near zone would inflate counts for wide hits.

Value

A `data.frame` with one row per motif passing the q-value cutoff, sorted by `pvalue` ascending. Columns: `motif`, `motif.i`, `method`, `count`, `orientation`, `nhits`, `best.dist`, `hits.near`, `hits.far`, `frac.near`, `expected.near`, `enrichment`, `log2.enrichment`, `pvalue`, `qvalue`, `ks.D` (non-NA only for `method = "ks"`), and `dist.to.nearest` (list-column of per-hit signed distances to the nearest anchor, consumed by `plot_motif_proximity()`). For `count = "anchors"`, `nhits`, `hits.near` and `hits.far` count anchors rather than hits.

Caveats

The universe derived from `seqlengths(hits)` is correct only for whole, unmasked chromosome scans. For masked or partial scans (or for windowed sequences extracted with `Biostrings::getSeq()`), pass `universe` explicitly; otherwise `f_d` is overstated and the test is anti-conservative.

The `"binomial"` null assumes hits are uniformly distributed over the universe and cannot correct for composition or mappability bias. Prefer `method = "permutation"` when that bias is plausible.

Anchor width strongly affects power and resolution. The near zone is the anchor widened by `d` on each side, so its width is `width(anchor) + 2d` and the within-distance fraction `f_d` grows with anchor width; a hit inside an anchor region also has a distance of zero. Wide anchors therefore inflate the expected count (shrinking the enrichment toward 1), wash out the distance distribution, and weaken the test, while the biological signal is unchanged. For a point hypothesis (summits, TSSs, SNPs, motif centres) use 1 bp anchors, reducing regions with `GenomicRanges::resize(anchors, width = 1, fix = "center")` (or the called summit). Keep regions only when the question really is enrichment within or around intervals, and then scale `dist.thresholds` to the region width.

All inputs must share chromosome names. A mismatch (e.g. `chr1` versus `Chr1` versus `1`) raises an error; reconcile with `GenomeInfoDb::seqlevelsStyle()`.

`method = "permutation"` rebuilds the near zone for every distance threshold on every permutation, so its cost scales with `n.perm` times `length(dist.thresholds)`; for genome-scale anchor sets a run can take minutes per motif. Lower `n.perm`, shorten `dist.thresholds`, or use `method = "binomial"` / `"ks"` when that is prohibitive.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Bailey TL, Machanick P (2012). "Inferring direct DNA binding from ChIP-seq." *Nucleic Acids Research*, **40**(17):e128. doi:10.1093/nar/gks433.

See Also

`motif_peaks()`, `scan_sequences_lite()`, `plot_motif_proximity()`

Examples

```
## Not run:
library(universalmotif)
library(GenomicRanges)
## Treat three created sequences as small chromosomes.
set.seed(1)
chrs <- do.call(c, lapply(1:3, function(i)
  create_sequences(seqnum = 1, seqlen = 5000, rng.seed = i)))
names(chrs) <- paste0("chr", 1:3)

## Plant a motif near a set of anchors on chr1.
anchor.pos <- seq(500, 4500, by = 400)
planted <- Biostrings::DNAString("TTGACATA")
for (p in anchor.pos)
  subseq(chrs[["chr1"]], start = p + 5L, width = 8) <- planted
anchors <- GRanges("chr1", IRanges(anchor.pos, width = 1))

m <- create_motif("TTGACATA", name = "example")

## Either scan first and pass hits ...
hits <- scan_sequences_lite(m, chrs, pvalue = 1e-3, return.granges = TRUE)
motif_proximity(hits, anchors)

## ... or let motif_proximity() scan internally.
motif_proximity(motifs = m, sequences = chrs, anchors = anchors)

## A composition-robust permutation test (set.seed() for reproducibility).
set.seed(1)
motif_proximity(hits, anchors, method = "permutation", n.perm = 199)

## End(Not run)
```

`motif_pvalue` *Motif P-value and scoring utility*

Description

For calculating P-values and logodds scores from P-values for any number of motifs.

Usage

```
motif_pvalue(motifs, score, pvalue, bkg.probs, use.freq = 1, k = 8,
  nthreads = 1, rand.tries = 10, rng.seed = sample.int(10000, 1),
  allow.nonfinite = FALSE, method = c("dynamic", "exhaustive"))
```

Arguments

`motifs`	See `convert_motifs()` for acceptable motif formats.
`score`	`numeric`, `list` Get a P-value for a motif from a logodds score. See details for an explanation of how to vectorize the calculation for `method = "dynamic"`.
`pvalue`	`numeric`, `list` Get a logodds score for a motif from a P-value. See details for an explanation of how to vectorize the calculation for `method = "dynamic"`.
`bkg.probs`	`numeric`, `list` A vector background probabilities. If supplying individual background probabilities for each motif, a list of such vectors. If missing, retrieves the background from the motif `bkg` slot. With `use.freq = 1`, changing this background first rebases the motif's probabilities into a new PWM. For `method = "dynamic"`, it also weights the null distribution for both score and P-value queries; this is not a fixed-PWM null-only change. The exhaustive score-to-P-value calculation also uses these null weights, but exhaustive P-value-to-score inversion uses sequence counts rather than background-weighted probabilities; prefer the dynamic method for non-uniform backgrounds.
`use.freq`	`numeric(1)` By default uses the regular motif matrix; otherwise uses the corresponding `multifreq` matrix. Max is 3 when `method = "exhaustive"`.
`k`	`numeric(1)` For speed, scores/P-values can be approximated after subsetting the motif every `k` columns when `method = "exhaustive"`. If `k` is a value equal or higher to the size of input motif(s), then the calculations are exact. The default, 8, is recommended to those looking for a good tradeoff between speed and accuracy for jobs requiring repeated calculations. Note that this is ignored when `method = "dynamic"`, as subsetting is not required.
`nthreads`	`numeric(1)` Run `motif_pvalue()` in parallel with `nthreads` threads. `nthreads = 0` uses all available threads. The dynamic method parallelises across motifs.
`rand.tries`	`numeric(1)` When `ncol(motif) < k` and `method = "exhaustive"`, an approximation is used. This involves randomly approximating the overall motif score distribution. To increase accuracy, the distribution is approximated `rand.tries` times and the final scores averaged. Note that this is ignored when `method = "dynamic"`, as subsetting is not required.
`rng.seed`	`numeric(1)` In order to allow `motif_pvalue()` to perform C++ level parallelisation, it must work independently from R. This means it cannot communicate with R to get/set the R RNG state. To get around this, the RNG seed used by the C++ function can be set with `rng.seed`. To make sure each thread gets a

different seed however, the seed is multiplied with the iteration count. For example: when working with two motifs, the second motif gets the following seed: `rng.seed * 2`. The default is to pick a random number as chosen by `sample()`, which effectively makes `motif_pvalue()` dependent on the R RNG state. Note that this is ignored when `method = "dynamic"`, as the random subsetting is only used for `method = "exhaustive"`.

`allow.nonfinite`

`logical(1)` If `FALSE`, then apply a pseudocount if non-finite values are found in the PWM. Note that if the motif has a pseudocount greater than zero and the motif is not currently of type PWM, then this parameter has no effect as the pseudocount will be applied automatically when the motif is converted to a PWM internally. Note that this option is incompatible with `method = "dynamic"`. A message will be printed if a pseudocount is applied. To disable this, set `options(pseudocount.warning=FALSE)`.

`method` `character(1)` One of `c("dynamic", "exhaustive")`. Algorithm used for calculating P-values. The `"exhaustive"` method involves finding all possible motif matches at or above the specified score using a branch-and-bound algorithm, which can be computationally intensive (Hartman et al., 2013). Additionally, the computation must be repeated for each hit. The `"dynamic"` method calculates the distribution of possible motif scores using a much faster dynamic programming algorithm, and can be recycled for multiple scores (Grant et al., 2011). The only disadvantage is the inability to use `allow.nonfinite = TRUE`.

Details

Regarding vectorization:

A note regarding vectorizing the calculation when `method = "dynamic"` (no vectorization is possible with `method = "exhaustive"`): to avoid performing the P-value/score calculation repeatedly for individual motifs, provide the `score`/`pvalue` arguments as a list, with each entry corresponding to the scores/P-values to be calculated for the respective motifs provided to `motifs`. If you simply provide a list of repeating motifs and a single numeric vector of corresponding input scores/P-values, then `motif_pvalue()` will not vectorize. See the Examples section.

The dynamic method:

One of the algorithms available to `motif_pvalue()` to calculate scores or P-values is the dynamic programming algorithm used by FIMO (Grant et al., 2011). In this method, a small range of possible scores from the possible miminum and maximum is created and the cumulative probability of each score in this distribution is incrementally calculated using the logodds scores and the background probabilities. This distribution of scores and associated P-values can be used to calculate P-values or scores for any input, any number of times. This method scales well with large motifs, and `multifreq` representations. The only downside is that it is incompatible with `allow.nonfinite = TRUE`, as this would not allow for the creation of the initial range of scores. Although described for a different purpose, the basic premise of the dynamic programming algorithm is also described in Gupta et al. (2007).

Each PWM entry is multiplied by 1000 and truncated towards zero before scores are summed. Dynamic P-values are upper-tail probabilities for this integerised distribution, which can differ from the distribution of the original real-valued scores. Query scores are rounded up to the next score grid point, with floating-point round-trip error handled at grid boundaries. For a requested P-value, the returned score is an inclusive threshold whose upper-tail probability is at most that P-value. If no attainable score satisfies the cutoff, the threshold is `Inf` and sequence scanning returns no hits for that motif.

The exhaustive method:

Calculating P-values exhaustively for motifs can be very computationally intensive. This is due to how P-values must be calculated: for a given score, all possible sequences which score equal or higher must be found, and the probability for each of these sequences (based on background probabilities) summed. For a DNA motif of length 10, the number of possible unique sequences is 4^10 = 1,048,576. Finding all possible sequences higher than a given score can be done very efficiently and quickly with a branch-and-bound algorithm, but as the motif length increases even this calculation becomes impractical. To get around this, the P-value calculation can be approximated.

In order to calculate P-values for longer motifs, this function uses the approximation proposed by Hartmann et al. (2013), where the motif is subset, P-values calculated for the subsets, and finally combined for a total P-value. The smaller the size of the subsets, the faster the calculation; but also, the bigger the approximation. This can be controlled by setting `k`. In fact, for smaller motifs (< 13 positions) calculating exact P-values can be done individually in reasonable time by setting `k = 12`.

To calculate a score from a P-value, all possible scores are calculated and the `(1 - pvalue) * 100` nth percentile score returned. When `k < ncol(motif)`, the complete set of scores is instead approximated by randomly adding up all possible scores from each subset. Note that this approximation can actually be potentially quite expensive at times and even slower than the exact version; for jobs requiring lots of repeat calculations, a bit of benchmarking beforehand can be useful to find the optimal settings.

Please note that bugs are more likely to occur when using the exhaustive method, as the algorithm contains several times more code compared to the dynamic method. Unless you have a strong need to use `allow.nonfinite = TRUE` then avoid using this method.

Value

`numeric`, `list` A vector or list of vectors of scores/P-values.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Grant CE, Bailey TL, Noble WS (2011). "FIMO: scanning for occurrences of a given motif." *Bioinformatics*, **27**, 1017-1018.

Gupta S, Stamatoyannopoulos JA, Bailey TL, Noble WS (2007). "Quantifying similarity between motifs." *Genome Biology*, **8**, R24.

Hartmann H, Guthohrlein EW, Siebert M, Soding SLJ (2013). "P-value-based regulatory motif discovery using positional weight matrices." *Genome Research*, **23**, 181-194.

See Also

`get_matches()`, `get_scores()`, `motif_range()`, `motif_score()`, `prob_match()`, `prob_match_bkg()`, `score_match()`

Examples

```
if (R.Version()$arch != "i386") {

## P-value/score calculations are performed using the PWM version of the
## motif
data(examplemotif)
```

```
## Get a minimum score based on a P-value
motif_pvalue(examplemotif, pvalue = 0.001)

## Get the probability of a particular sequence hit
motif_pvalue(examplemotif, score = 0)

## The calculations can be performed for multiple motifs
motif_pvalue(c(examplemotif, examplemotif), pvalue = c(0.001, 0.0001))

## Compare score thresholds and P-value:
scores <- motif_score(examplemotif, c(0.6, 0.7, 0.8, 0.9))
motif_pvalue(examplemotif, scores)

## Calculate the probability of getting a certain match or better:
TATATAT <- score_match(examplemotif, "TATATAT")
TATATAG <- score_match(examplemotif, "TATATAG")
motif_pvalue(examplemotif, TATATAT)
motif_pvalue(examplemotif, TATATAG)

## Get all possible matches by P-value:
get_matches(examplemotif, motif_pvalue(examplemotif, pvalue = 0.001))
## A stricter cutoff cannot be met: the threshold is Inf, with no matches.
get_matches(examplemotif, motif_pvalue(examplemotif, pvalue = 0.0001))

## Vectorize the calculation for multiple motifs and scores/P-values:
m <- create_motif()
motif_pvalue(c(examplemotif, m), list(1:5, 2:3))
## The non-vectorized equivalent:
motif_pvalue(
  c(rep(list(examplemotif), 5), rep(list(m), 2)), c(1:5, 2:3)
)
}
```

`motif_rc` *Get the reverse complement of a DNA or RNA motif.*

Description

For any motif, change the `motif` slot to it's reverse complement. If the `multifreq` slot is filled, then it is also applied. No other slots are affected.

Usage

```
motif_rc(motifs, ignore.alphabet = FALSE)
```

Arguments

`motifs`	See `convert_motifs()` for acceptable formats
`ignore.alphabet`	`logical(1)` If `TRUE`, then `motif_rc()` throws an error when it detects a non-DNA/RNA motif. If `FALSE`, it will proceed regardless.

Value

See convert_motifs() for available output formats.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

Examples

```
jaspar <- read_jaspar(system.file("extdata", "jaspar.txt",
                                  package = "universalmotif"))
jaspar.rc <- motif_rc(jaspar)
```

---

motif_tree *Generate* **ggplot2** *motif trees with* **ggtree**.

Description

For more powerful motif tree functions, see the **motifStack** package. The motif_tree() function compares motifs with compare_motifs() to create a distance matrix, which is used to generate a phylogeny. This can be plotted with ggtree::ggtree(). The purpose of this function is simply to combine the compare_motifs() and ggtree::ggtree() steps into one. For more control over tree creation, it is recommend to do these steps separately. See the "Motif comparisons and P-values" vignette for such a workthrough. This function requires the **ape** and **ggtree** packages to be installed separately. For a leaner alternative using compare_motifs_lite(), see motif_tree_lite().

Usage

```
motif_tree(motifs, layout = "circular", linecol = "family",
  labels = "none", tipsize = "none", legend = TRUE,
  branch.length = "none", db.scores, method = "EUCL", use.type = "PPM",
  min.overlap = 6, min.position.ic = 0, tryRC = TRUE, min.mean.ic = 0,
  relative_entropy = FALSE, progress = FALSE, nthreads = 1,
  score.strat = "a.mean", ...)
```

Arguments

motifs	list, dist See convert_motifs() for available formats. Alternatively, the resulting comparison matrix from compare_motifs() (run as.dist(results) beforehand; if the comparison was performed with a similarity metric, make sure to convert to distances first).
layout	character(1) One of c('rectangular', 'slanted', 'fan', 'circular', 'radial', 'equal_angle', 'daylight'). See ggtree::ggtree().
linecol	character(1) universalmotif slot to use to colour lines (e.g. 'family'). Not available for dist input (see examples for how to add it manually). See ggtree::ggtree().
labels	character(1) universalmotif slot to use to label tips (e.g. 'name'). For dist input, only 'name' is available. See ggtree::ggtree(). Note: drawing tip labels uses ggtree::geom_tiplab(align = TRUE), which fails to render under **ggplot2** >= 4.0 with **ggtree** <= 3.12 (ggtree's internal empty() calls the removed ggplot2:::is.waive()); update **ggtree** if you need tip labels.

tipsize — character(1) universalmotif slot to use to control tip size (e.g. 'icscore'). Not available for dist input (see examples for how to add it manually). See ggtree::ggtree().

legend — logical(1) Show legend for line colour and tip size. See ggtree::ggtree().

branch.length — character(1) If 'none', draw a cladogram. See ggtree::ggtree().

db.scores — data.frame See compare_motifs().

method — character(1) One of PCC, EUCL, SW, KL, ALLR, BHAT, HELL, SEUCL, MAN, ALLR_LL, WEUCL, WPCC. See details.

use.type — character(1)c('PPM', 'ICM'). The latter allows for taking into account the background ative_entropy = TRUE'). See compare_motifs().

min.overlap — numeric(1) Minimum overlap required when aligning the motifs. Setting this to a number higher then the width of the motifs will not allow any overhangs. Can also be a number between 0 and 1, representing the minimum fraction that the motifs must overlap.

min.position.ic — numeric(1) Minimum information content required between individual alignment positions for it to be counted in the final alignment score. It is recommended to use this together with normalise.scores = TRUE, as this will help punish scores resulting from only a fraction of an alignment.

tryRC — logical(1) Try the reverse complement of the motifs as well, report the best score.

min.mean.ic — numeric(1) Minimum mean information content between the two motifs for an alignment to be scored. This helps prevent scoring alignments between low information content regions of two motifs. Note that this can result in some comparisons failing if no alignment passes the mean IC threshold. Use average_ic() to filter out low IC motifs to get around this if you want to avoid getting NAs in your output.

relative_entropy — logical(1) Change the ICM calculation affecting min.position.ic and min.mean.ic. See convert_type().

progress — logical(1) Show message regarding current step.

nthreads — numeric(1) Run compare_motifs() in parallel with nthreads threads. nthreads = 0 uses all available threads.

score.strat — character(1) How to handle column scores calculated from motif alignments. "sum": add up all scores. "a.mean": take the arithmetic mean. "g.mean": take the geometric mean. "median": take the median. "wa.mean", "wg.mean": weighted arithmetic/geometric mean. "fzt": Fisher Z-transform. Weights are the total information content shared between aligned columns.

... — **ggtree** params. See ggtree::ggtree().

Details

See compare_motifs() for more info on comparison parameters.

Value

ggplot object.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Wickham H (2009). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. ISBN 978-0-387-98140-6, <URL: http://ggplot2.org>.

Yu G, Smith D, Zhu H, Guan Y, Lam TT (2017). "ggtree: an R package for visualization and annotation of phylogenetic trees with their covariates and other associated data." *Methods in Ecology and Evolution*, **8**, 28-36. doi: 10.1111/2041-210X.12628.

See Also

motifStack::motifStack(), compare_motifs(), ggtree::ggtree(), ggplot2::ggplot(), motif_tree_lite()

Examples

```
jaspar <- read_jaspar(system.file("extdata", "jaspar.txt",
                                  package = "universalmotif"))
if (requireNamespace("ggtree", quietly = TRUE)) {
jaspar.tree <- motif_tree(jaspar, linecol = "none", labels = "name",
                          layout = "rectangular")
}

## Not run:
## When inputting a dist object, the linecol and tipsize options are
## not available. To add these manually:

library(MotifDb)
library(ggtree)
library(ggplot2)

motifs <- filter_motifs(MotifDb, organism = "Athaliana")[1:50]
comparison <- compare_motifs(motifs, method = "PCC", score.strat = "a.mean")
comparison <- as.dist(1 - comparison)
mot.names <- attr(comparison, "Labels")
tree <- motif_tree(comparison)

annotations <- data.frame(label = mot.names,
                          icscore = sapply(motifs, function(x) x["icscore"]),
                          family = sapply(motifs, function(x) x["family"]))

tree <- tree %<+% annotations +
          geom_tippoint(aes(size = icscore)) +
          aes(colour = family) +
          theme(legend.position = "right",
                legend.title = element_blank())

## End(Not run)
```

motif_tree_lite *Generate ggtree-based motif trees using* compare_motifs_lite().

Description

motif_tree_lite() is the leaner counterpart of motif_tree(): it builds the distance matrix via compare_motifs_lite() (mean Pearson correlation, with built-in p-value / q-value machinery) instead of compare_motifs(). The distance for hclust() is derived from the mean Pearson correlation matrix as (1 - score) / 2, mapping similarity in [-1, 1] to a symmetric distance in [0, 1]. The tree-construction and rendering steps (hclust() followed by ape::as.phylo() followed by ggtree::ggtree()) match motif_tree() exactly, so the visual conventions, the linecol / labels / tipsize slot-mapping arguments, and the dist-input short-circuit all behave the same.

Usage

```
motif_tree_lite(motifs, layout = "rectangular", linecol = "family",
  labels = "none", tipsize = "none", legend = TRUE,
  branch.length = "none", min.overlap = 6, tryRC = TRUE, nthreads = 1,
  progress = FALSE, tiplogo = TRUE, tiplogo.align = TRUE,
  tiplogo.width = 1, tiplogo.height = 0.9, tiplogo.offset = NULL, ...)
```

Arguments

motifs	list or dist. See convert_motifs() for accepted motif formats. Alternatively, a pre-built dist object (e.g. the result of as.dist((1 - compare_motifs_lite(...)) / 2)) skips the comparison step entirely and goes straight to tree construction.
layout	character(1). One of c('rectangular', 'slanted', 'fan', 'circular', 'radial', 'equal_angle', 'daylight'). Passed through to ggtree::ggtree(). Defaults to "rectangular" so that the tip logos (see tiplogo) are visible; tip logos are only drawn on the "rectangular" and "slanted" layouts.
linecol	character(1). Motif slot used to colour the branches (e.g. "family"). Not used when motifs is a dist.
labels	character(1). Motif slot used to label the tips (e.g. "name"). For dist input, only "name" is meaningful. Note: drawing tip labels uses ggtree::geom_tiplab(align = TRUE), which fails to render under **ggplot2** >= 4.0 with **ggtree** <= 3.12 (ggtree's internal empty() calls the removed ggplot2:::is.waive()); update **ggtree** if you need tip labels. The tip logos (tiplogo) are unaffected.
tipsize	character(1). Motif slot used to size the tips (e.g. "icscore"). Not used when motifs is a dist.
legend	logical(1). Show legend for line colour and tip size.
branch.length	character(1). If "none", draw a cladogram. Passed through to ggtree::ggtree().
min.overlap	integer(1). Minimum overlap in columns for the pairwise alignment to count. Forwarded to compare_motifs_lite(). Default 6.
tryRC	logical(1). Also test reverse-complement alignments. Forwarded to compare_motifs_lite(). Default TRUE.
nthreads	numeric(1). Threads passed to compare_motifs_lite(). nthreads = 0 uses all available threads.
progress	logical(1). Print progress messages.

tiplogo	logical(1). Draw a sequence logo at each tip using geom_logo(). On by default. Only drawn on the "rectangular" and "slanted" layouts (other layouts warn and skip), and not available for dist input. Default TRUE.
tiplogo.align	logical(1). Align the tip logos to a common, zero-padded column frame (via view_motifs_lite()) so that homologous positions line up across tips. DNA/RNA only; falls back to unaligned per-tip logos otherwise. Default TRUE.
tiplogo.width	numeric(1). Width of one motif position, in tree x-axis units, for the tip logos. Default 1.
tiplogo.height	numeric(1). Total height of a tip logo, in tree y-axis units, centred on each tip. Default 0.9.
tiplogo.offset	numeric(1) or NULL. Gap between the tips and the start of the logos, in tree x-axis units. If NULL, a small fraction of the tree width is used. Default NULL.
...	Additional arguments passed through to ggtree::ggtree().

Details

The PCC-to-distance conversion d = (1 - score) / 2 is symmetric (the comparison matrix from compare_motifs_lite() is symmetric in matrix mode), so as.dist() and hclust() accept it directly. Anti-correlated motif pairs land at the maximum distance of 1; identical motifs land at 0.

For more control over tree construction, run compare_motifs_lite() directly with matrix.out = "score", convert to a dist object yourself, and pass that dist to motif_tree_lite(). The function detects the dist input and skips the comparison step.

Value

A ggplot (ggtree) object.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

motif_tree(), compare_motifs_lite(), merge_similar_lite(), geom_logo(), view_motifs_lite(), ggtree::ggtree()

Other lite motif functions: compare_motifs_lite(), enrich_motifs_lite(), merge_motifs_lite(), merge_similar_lite(), scan_sequences_lite(), view_motifs_lite()

Examples

```
## Not run:
library(universalmotif)
jaspar <- read_jaspar(system.file("extdata", "jaspar.txt",
                                  package = "universalmotif"))
if (requireNamespace("ggtree", quietly = TRUE)) {
  motif_tree_lite(jaspar, linecol = "none", labels = "name",
              layout = "rectangular")
}

## Equivalent two-step form, useful when you want to inspect the
## distance matrix before drawing the tree:
if (requireNamespace("ggtree", quietly = TRUE)) {
```

```
  score.mat <- compare_motifs_lite(jaspar, matrix.out = "score")
  d <- as.dist((1 - score.mat) / 2)
  motif_tree_lite(d, labels = "name")
}

## End(Not run)
```

---

plot_comparison *Draw motif similarities as a clustered heatmap.*

Description

Motifs are compared all against all with compare_motifs_lite(), or the comparison is given as matrix, a square matrix such as compare_motifs_lite() returns without compare.to. Rows are queries and columns targets, both in the order of a hierarchical clustering drawn as a dendrogram beside the heatmap. Scores are drawn as they are, and P- and q-values as their -log10, with the diagonal left blank since it compares each motif with itself.

Usage

```
plot_comparison(motifs = NULL, matrix = NULL, labels = "name",
  value = c("score", "pvalue", "qvalue"), min.overlap = 6L, RC = TRUE,
  cluster = TRUE, method = "complete", dendrogram = c("left", "top",
  "both", "none"), cut = NULL, cut.height = NULL,
  colours = PLOT_COMPARISON_BLUES, limits = NULL, annotate = NULL,
  logos = NULL, text.size = 10, nthreads = 1)
```

Arguments

motifs	See convert_motifs() for accepted motif formats. The motifs to compare, or the motifs matrix was made from, which give the labels and logos.
matrix	matrix A precomputed square matrix of the kind chosen by value. Its column names become the labels when no motifs are given. Scores are clustered as correlations, so a matrix of other similarity scores needs a tree of its own as cluster, or no clustering.
labels	character A motif slot to label rows and columns with, such as "name" or "altname", or one label per motif.
value	character(1) What is drawn: "score", the mean column correlation, or "pvalue" or "qvalue".
min.overlap	integer(1) Passed to compare_motifs_lite(). The default of 6 follows motif_tree_lite(), so the scores can differ slightly from those of compare_motifs_lite() with its default of 5.
RC	logical(1) Passed to compare_motifs_lite().
cluster	logical(1) or hclust Order the motifs by clustering them with method, or keep them in input order with FALSE. A tree from stats::hclust() can be given instead. Motifs are clustered on their scores, with distances of (1 - score) / 2, so that the order is that of motif_tree_lite() with the same min.overlap. A matrix of P- or q-values given as matrix without motifs is clustered on the mean -log10 value of both directions, scaled to distances from 0 to 1.

method	character(1) The linkage passed to stats::hclust().
dendrogram	character(1) Where to draw the tree: "left", "top", "both" or "none".
cut, cut.height	numeric(1) Show where cutting the tree would split the motifs into groups, as a dashed line across the dendrogram and a box around each group of two or more motifs on the heatmap. cut is in the units clustered on, a score such as 0.8 (or a P- or q-value for such a matrix given without motifs), and with complete linkage every motif in a group is then at least that similar to every other. A tree given as cluster is taken to be one of scores; cut.height instead cuts at a height of the tree itself, as stats::cutree() does.
colours	character The colours of the gradient, from low to high. The default is the ColorBrewer "Blues" palette.
limits	numeric(2) The range of the colour scale, which by default spans the values drawn, from 0 for P- and q-values. Values outside it take the colour at the nearest end.
annotate	logical(1) Write each value in its cell, by default for up to 20 motifs.
logos	logical(1) Draw each motif's logo beside its row label, by default when motifs are given and there are up to 60 of them.
text.size	numeric(1) The largest font size of labels, in points. Labels shrink with the cells of larger heatmaps and are left out once they would be unreadable.
nthreads	numeric(1) Passed to compare_motifs_lite().

Details

Everything is drawn in a single panel with a fixed aspect ratio, where each cell is one unit across, so that the dendrograms, logos and labels stay aligned with the rows and columns whatever the size of the device. Text sizes are fixed in points, however, and are chosen for cells of about 0.45 inches for up to 15 motifs, shrinking to about 0.15 inches for 80 or more. The size of device this needs, labels and colour bar included, is given by attr(p, "size"), which can be passed on to ggplot2::ggsave() as in the examples.

Value

A ggplot object, with the width and height in inches it is drawn for as its "size" attribute (see Details).

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

compare_motifs_lite(), motif_tree_lite(), geom_motif()

Examples

```
motifs <- read_jaspar(system.file("extdata", "jaspar.txt",
                                  package = "universalmotif"))
p <- plot_comparison(motifs)
p
## Not run:
ggplot2::ggsave("comparison.pdf", p, width = attr(p, "size")[1],
  height = attr(p, "size")[2])
```

```
## End(Not run)

## P-values, with the tree cut at a mean column correlation of 0.4:
plot_comparison(motifs, value = "pvalue", dendrogram = "both", cut = 0.4)
```

---

plot_coocc *Plot motif co-occurrence results as a heatmap.*

Description

Draw the results of motif_coocc() as a heatmap with a cell for every pair of motifs, filled the same on both sides of the diagonal with the log2 odds ratio of the pair (on a scale centred on zero) or with its -log10 q-value or P-value. Pairs with q-values at or below cutoff are marked with a dot.

Usage

```
plot_coocc(results, value = c("odds_ratio", "qvalue", "pvalue"),
  cutoff = 0.1, order = "cluster", colours = NULL, text.size = 9)
```

Arguments

results	data.frame returned by motif_coocc(), with the motif_a, motif_b, odds_ratio, pvalue and qvalue columns.
value	character(1) One of c("odds_ratio", "qvalue", "pvalue"), the value the cells are filled with.
cutoff	numeric(1) The q-value at or below which pairs are marked, in (0, 1].
order	character "cluster" to order the motifs by average-linkage clustering of their rows of shown values, "input" to keep them in the order they first appear in results, or the names of the motifs in the order to draw them, which can leave some out.
colours	character or NULL The colours of the fill gradient, from lowest to highest. By default a diverging blue to red scale for odds ratios and a sequential blue scale for q-values and P-values.
text.size	numeric(1) The font size of the motif names and the legends, in points.

Details

Untested pairs (those with NA P-values, which motif_coocc() gives to pairs that co-occur in fewer than min.coocc sequences and to self pairs) and the diagonal are filled grey, as are tested pairs whose odds ratio is undefined. Odds ratios of zero or infinity are drawn at the ends of the colour scale, whose limits are set by the largest finite log2 odds ratio.

The dots are drawn in white on dark cells and in black on light ones. When clustering, untested pairs count as having no association (a log2 odds ratio or -log10 P-value of zero), infinite log2 odds ratios as the largest finite one, and a missing diagonal as the strongest association, so that motifs found together end up next to each other.

Value

A ggplot object.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

motif_coocc(), plot_enrichment()

Examples

```
motifs <- list(create_motif("TTGACCAA", name = "W-box", nsites = 20),
  create_motif("CACGTGGC", name = "G-box", nsites = 20),
  create_motif("TATAAATA", name = "TATA-box", nsites = 20),
  create_motif("CCGGAAGT", name = "ETS", nsites = 20))
seqs <- create_sequences(seqnum = 200, seqlen = 200, rng.seed = 1)
## Plant W-box and G-box sites together, and TATA-box sites alone.
set.seed(1)
seqs[1:60] <- implant_motifs(motifs[[1]], seqs[1:60])
seqs[1:60] <- implant_motifs(motifs[[2]], seqs[1:60])
seqs[61:120] <- implant_motifs(motifs[[3]], seqs[61:120])
res <- motif_coocc(motifs, seqs, pvalue = 1e-4, nthreads = 1)
plot_coocc(res)

## Fill by q-value, with the motifs in a given order.
plot_coocc(res, value = "qvalue", order = c("W-box", "G-box", "TATA-box"))
```

---

plot_enrichment *Plot motif enrichment results as a volcano plot.*

Description

Draw the results of enrich_motifs_lite() or enrich_motifs() with each motif as a point at its log2 enrichment and -log10 q-value (or P-value with y = "pvalue"), sized by the number of target sequences with hits. Motifs at or below cutoff are coloured, a dashed line marks the cutoff, and the most significant motifs are labelled, optionally with their logos beside the labels.

Usage

```
plot_enrichment(results, y = c("qvalue", "pvalue"), cutoff = 0.1,
  labels = 10, size.by = "target.seq.hits", motifs = NULL,
  logo.height = NULL, colour = "#255C99", text.size = 9)
```

Arguments

results	data.frame or DataFrame returned by enrich_motifs_lite() or enrich_motifs(). It needs a motif column, either a log2.enrichment or a target.enrichment column, and the P-value or q-value column chosen with y (pvalue/qvalue from enrich_motifs_lite(), Pval/Qval from enrich_motifs()).
y	character(1) One of c("qvalue", "pvalue"), the value drawn on the vertical axis and compared to cutoff.
cutoff	numeric(1) The q-value or P-value at or below which motifs are significant, in (0, 1].

labels	numeric(1) or character The number of most significant motifs to label, or the names of the motifs to label whether they are significant or not. Labels are moved apart so that they do not cover each other or other points.
size.by	character(1) or NULL A column of results to size the points by, or NULL for points of one size. The names of the hit count columns of enrich_motifs_lite() and enrich_motifs() can be used for either table (for example "target.site.hits" also finds the target.hits column of enrich_motifs()).
motifs	See convert_motifs() for accepted motif formats. The motifs given to the enrichment function, to draw a logo beside each label. Motifs are matched to the rows of results by the names of the list when it has them, and otherwise by motif name (with duplicates renamed by make.unique(), as enrich_motifs_lite() does).
logo.height	numeric(1) or NULL The height of the logos in points, by default twice text.size.
colour	character(1) The colour of significant points.
text.size	numeric(1) The font size of the labels and legend, in points.

Details

Motifs without any target hits have an enrichment of zero, and so an infinitely negative log2 enrichment; these are drawn as triangles at the left edge of the plot instead. Likewise motifs with hits in the targets but none in the background (enrich_motifs() can give these an infinite enrichment) are drawn as triangles at the right edge. P-values and q-values too small to be stored as doubles are drawn at the smallest representable value.

The labels are placed when the plot is drawn, so that they stay next to their points and apart from each other whatever size the plot is drawn at. Each label in turn, from the most significant, tries spots at growing distances around its point and takes the one that covers the fewest of the labels placed before it, the other points and the edge of the panel. Labels that end up away from their points are joined to them by a thin line.

enrich_motifs_lite() only returns motifs with q-values of 0.1 or less by default; use qvalue = 1 to plot every motif, as the plot is easier to read with the motifs that fall short of the cutoff in it too.

Value

A ggplot object.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

enrich_motifs_lite(), enrich_motifs(), geom_motif(), plot_coocc()

Examples

```
data(ArabidopsisPromoters)
data(ArabidopsisMotif)
motifs <- list(ArabidopsisMotif,
  create_motif("TTGACY", name = "W-box", nsites = 20),
  create_motif("CACGTG", name = "G-box", nsites = 20),
  create_motif("TATAWAWA", name = "TATA-box", nsites = 20),
  create_motif("CCGGAAGT", name = "ETS", nsites = 20))
```

```
bkg <- shuffle_sequences(ArabidopsisPromoters, k = 2, rng.seed = 1)
res <- enrich_motifs_lite(motifs, ArabidopsisPromoters, bkg,
  pvalue = 1e-3, qvalue = 1, nthreads = 1)
plot_enrichment(res, labels = 3, motifs = motifs)

## Label motifs by name, and draw P-values with points of one size.
plot_enrichment(res, y = "pvalue", cutoff = 0.001,
  labels = c("W-box", "ETS"), size.by = NULL)
```

`plot_hits` *Draw motif hits along sequences.*

Description

Each sequence is drawn as a thin horizontal line (a track), and each hit from `scan_sequences()` or `scan_sequences_lite()` is drawn on it from its start to its end as an arrow pointing along its strand, or as a block when it has none. Hits that overlap are stacked in lanes below the line so that none hide another, and are coloured by motif.

Usage

```
plot_hits(hits, sequences = NULL, ranges = NULL, names = NULL,
  max.tracks = 25, motifs = NULL, colours = NULL, shade.by = NULL,
  size.by = NULL, labels = TRUE, legend = TRUE, text.size = 9)
```

Arguments

`hits`	`DataFrame`, `data.frame` or `GRanges` Hits from `scan_sequences()` or `scan_sequences_lite()`, with `motif`, `start` and `stop` (or `end`) columns and a `sequence` column (the sequence names of a `GRanges`). A `strand` column sets the direction of the arrows. Positions are 1-based and inclusive; hits on the reverse strand may have their start after their stop, as `scan_sequences()` reports them.
`sequences`	`XStringSet` or named `numeric` The sequences scanned, or their lengths. Every sequence is then drawn at its full length in the order given, including those without hits. Otherwise only the sequences with hits are drawn, each up to its last hit. Unnamed sequences are named by their position in the set, as `scan_sequences()` names them. Not used together with `ranges`.
`ranges`	`GRanges` or `data.frame` Genomic ranges to draw, for hits found on chromosomes (for example by scanning a `BSgenome`). A `data.frame` needs `seqnames` (or `chrom`), `start` and `end` columns, with positions 1-based and inclusive. Each range becomes a track showing the hits that overlap it, labelled by a `name` column (or the names of a `GRanges`) when there is one and as `chrom:start-end` otherwise. The x-axis shows genomic positions for a single range, and positions from the start of each range for several.
`names`	`character` Draw only the tracks with these names, in this order.
`max.tracks`	`numeric(1)` Draw at most this many tracks, or all of them with `NULL`. A note above the plot says when tracks were left out.

motifs — character, universalmotif or list The motifs to draw, by name or as the motifs scanned (anything convert_motifs() accepts). This sets their order in the legend, and motifs given as objects have their logos added to it. Duplicated names are made unique as scan_sequences_lite() makes them.

colours — character A vector of colours named by motif. By default motifs take the default **ggplot2** hues, spread over the motifs in the order they were scanned (or given in motifs).

shade.by, size.by — character(1) A numeric column of hits to set the opacity or the height of the hits by, from the smallest value to the largest, such as "score" or "pvalue". P-values, Q-values and E-values are taken as -log10(), so that stronger hits are always darker and taller.

labels — logical(1) Name the motif on each hit long enough to hold it.

legend — logical(1) Draw a legend of the motifs (and of shade.by and size.by) beside the plot.

text.size — numeric(1) Size in points of the track names, axis text and legend. Labels on hits are a little smaller.

Details

Arrow heads, the narrowest hit drawn, the gap kept between hits in a lane and which hits are wide enough to be labelled are worked out for a plotting area about 6.5 inches wide, with each lane about 0.17 inches tall. The plot reads best at roughly those proportions, so a height of about `0.17 * (number of lanes) + 1` inches suits it when saving with ggplot2::ggsave().

With ranges, a hit that runs past the end of a range is cut at that end, and a hit falling in several overlapping ranges is drawn in each of them.

Value

A ggplot object.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

scan_sequences(), scan_sequences_lite(), geom_motif()

Examples

```
data(ArabidopsisPromoters)
data(ArabidopsisMotif)
motifs <- c(ArabidopsisMotif, create_motif("CACGTG", name = "G-box"))
hits <- scan_sequences(motifs, ArabidopsisPromoters[1:4], threshold = 1e-4,
  threshold.type = "pvalue", nthreads = 1)
plot_hits(hits, ArabidopsisPromoters[1:4], motifs = motifs)

## Lite hits, shaded by P-value and without labels:
hits2 <- scan_sequences_lite(motifs, ArabidopsisPromoters[1:4],
  pvalue = 1e-3, return.granges = FALSE)
plot_hits(hits2, ArabidopsisPromoters[1:4], shade.by = "pvalue",
  labels = FALSE)
```

plot_match_bkg *Visualise composition match between target and background sequences.*

Description

Companion to match_bkg(). Overlays density curves for the target and matched-background sets in a faceted ggplot, useful for visually verifying that the binned matching landed where it was supposed to. Called with two XStringSets it plots GC fraction and sequence length; called with the indices data.frame from match_bkg(..., return.indices = TRUE) it plots every matched axis, including any covariates.

Usage

```
plot_match_bkg(sequences = NULL, bkg = NULL, by = NULL, bins = 30L,
  indices = NULL)
```

Arguments

sequences	XStringSet or NULL. The target sequences passed to match_bkg(). Ignored when indices is supplied.
bkg	XStringSet or NULL. The matched background sequences returned by match_bkg(). Ignored when indices is supplied.
by	character or NULL. Axes to plot. For the sequences/bkg form, a subset of c("gc", "length") (default both). For the indices form, a subset of the available axis names ("gc", "length", and each covariate name); NULL (default) plots them all.
bins	integer(1). Ignored when geom = "density"; kept for API parity with other v2 plotting helpers. Default 30L.
indices	data.frame or NULL. The output of match_bkg(..., return.indices = TRUE). When supplied, every target.<axis> / universe.<axis> column pair (apart from the index column) becomes a density panel, so covariate matches can be inspected alongside GC and length. Default NULL.

Value

A ggplot object.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

match_bkg()

Examples

```
data(ArabidopsisPromoters)
target   <- ArabidopsisPromoters[1:10]
universe <- ArabidopsisPromoters[11:50]
bkg <- match_bkg(target, universe, n.per.target = 2)

## GC and length density of the target vs. its matched background:
plot_match_bkg(target, bkg)
```

`plot_motif_peaks` *Plot the position distribution of motif hits.*

Description

Companion to `motif_peaks()`. Returns a `ggplot` faceted by motif, showing a histogram of best-hit centre positions and a shaded rectangle marking the most-enriched window.

Usage

```
plot_motif_peaks(peaks, motifs = NULL, ncol = 2L, bins = 50L,
  fill.window = "#ffe4a3")
```

Arguments

`peaks`	`data.frame` returned by `motif_peaks()`. Must carry the `centers` list-column.
`motifs`	`character` or `NULL`. Subset to these motifs (by name). `NULL` (default) plots every row of `peaks`.
`ncol`	`integer(1)`. Number of facet columns. Default `2`.
`bins`	`integer(1)`. Histogram bin count. Default `50`.
`fill.window`	`character(1)`. Colour of the best-window shade. Default `"#ffe4a3"` (a pale yellow).

Value

A `ggplot` object.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

`motif_peaks()`

Examples

```
library(Biostrings)
set.seed(1)
seqs <- create_sequences(seqnum = 200, seqlen = 500, rng.seed = 1)
planted <- DNAString("TTGACATA")
for (i in seq_len(150)) {
  pos <- 246L + sample.int(8, 1) - 1L
  subseq(seqs[[i]], start = pos, width = 8) <- planted
}
m <- create_motif("TTGACATA", name = "test")
hits <- scan_sequences(m, seqs, threshold = 1e-3, threshold.type = "pvalue")
peaks <- motif_peaks(as.data.frame(hits), seq.length = 500, qvalue = 1)
plot_motif_peaks(peaks)
```

---

plot_motif_proximity *Plot the distance-to-anchor distribution of motif hits.*

Description

Companion to motif_proximity(). Returns a ggplot faceted by motif, showing a histogram of per-hit distances to the nearest anchor (signed when the anchors are stranded, so upstream and downstream sit on opposite sides of zero) with the most-enriched near zone shaded.

Usage

```
plot_motif_proximity(proximity, motifs = NULL, ncol = 2L, bins = 50L,
  fill.zone = "#ffe4a3", log.x = FALSE)
```

Arguments

proximity	data.frame returned by motif_proximity(). Must carry the dist.to.nearest list-column.
motifs	character or NULL. Subset to these motifs (by name). NULL (default) plots every row of proximity.
ncol	integer(1). Number of facet columns. Default 2.
bins	integer(1). Histogram bin count. Default 50.
fill.zone	character(1). Colour of the best-zone shade. Default "#ffe4a3" (a pale yellow).
log.x	logical(1). Plot log10 of the absolute distance instead of the signed distance. Default FALSE.

Value

A ggplot object.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

motif_proximity()

---

plot_score_distribution

Plot the distribution of motif scores.

Description

Draw the distribution of a motif's scores in random sequences, with the score cutoffs that P-values give, to help choose a cutoff for scan_sequences().

Usage

```
plot_score_distribution(motifs, pvalues = 1e-04, bkg.probs, log.y = FALSE,
  hits = NULL, bins = 100, ncol = 2)
```

Arguments

motifs	See convert_motifs() for acceptable motif formats.
pvalues	numeric One or several P-value cutoffs to mark, or NULL for none. A motif too short or too degenerate to reach a cutoff gets a warning instead of a line.
bkg.probs	numeric, list Background probabilities, in any of the forms motif_pvalue() takes: a single vector for all motifs (named by letter or in alphabet order) or a list with one vector per motif. If missing, each motif's own bkg slot is used.
log.y	logical(1) Draw probabilities on a log scale, which shows the upper tail where cutoffs usually fall.
hits	Scores of real sites to mark as ticks along the bottom of each panel: the output of scan_sequences() or scan_sequences_lite() (a DataFrame, data.frame or GRanges with motif and score columns), a named list of scores with one entry per motif name, or for a single motif a numeric vector of scores.
bins	numeric(1) Number of bins the distribution is gathered into.
ncol	numeric(1) Number of panels per row when drawing several motifs.

Details

The distribution is the exact one that motif_pvalue() works from with method = "dynamic", gathered into bins bins of equal width between the lowest and highest possible scores, so that the height of each bin is the probability that a random site scores within it. Since scores lie on a grid of steps of 0.001, the bin edges are placed on that grid and the probability of each bin is the difference between the P-values of its two edges; the probabilities thus add up to one. Scores are the same logodds scores reported by motif_pvalue() and scan_sequences().

Each cutoff in pvalues is drawn as a vertical line at the lowest score with at most that P-value, labelled with the P-value, the score and the score as a percentage of the maximum possible score (as in the score.pct column of scan_sequences()). When a single motif is drawn, the top axis also shows scores as a percentage of the maximum; since this percentage differs between motifs, the top axis is left out when several motifs are drawn as panels.

Value

A ggplot object, with one panel per motif.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

motif_pvalue(), motif_range(), scan_sequences(), scan_sequences_lite()

Examples

```
data(ArabidopsisMotif)
data(ArabidopsisPromoters)
plot_score_distribution(ArabidopsisMotif, pvalues = c(1e-3, 1e-4))

## Mark the scores of hits along the bottom, on a log scale:
hits <- scan_sequences(ArabidopsisMotif, ArabidopsisPromoters,
  threshold = 1e-4, threshold.type = "pvalue", nthreads = 1)
plot_score_distribution(ArabidopsisMotif, hits = hits, log.y = TRUE)

## Several motifs, scored against an AT-rich background:
motifs <- read_meme(system.file("extdata", "meme_full.txt",
  package = "universalmotif"))
plot_score_distribution(motifs[c(1, 3)], pvalues = c(1e-3, 1e-5),
  bkg.probs = c(0.32, 0.18, 0.18, 0.32), log.y = TRUE)
```

---

read_cisbp *Import CIS-BP motifs.*

Description

Import CIS-BP formatted motifs. See http://cisbp.ccbr.utoronto.ca/index.php. Assumed to be DNA motifs.

Usage

```
read_cisbp(file, skip = 0)
```

Arguments

file	character(1) File name.
skip	numeric(1) If not zero, will skip however many desired lines in the file before starting to read.

Details

CIS-BP motifs can be formatted with or without additional header metadata. Motifs without any header start at instances of the word "Pos", whereas motifs with a header start at instances of the word "TF".

Value

`list` universalmotif objects.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Weirauch MT, Yang A, Albu M, Cote AG, Montenegro-Montero A, Drewe P, Najafabadi HS, Lambert SA, Mann I, Cook K, Zheng H, Goity A, van Bakel H, Lozano JC, Galli M, Lewsey MG, Huang E, Mukherjee T, Chen X, Reece-Hoyes JS, Govindarajan S, Shaulsky G, Walhout AJ, Bouget FY, Ratsch G, Larrondo LF, Ecker JR, Hughes TR (2014). "Determination and inference of eukaryotic transcription factor sequence specificity." *Cell*, **158**, 1431-1443.

See Also

Other read_motifs: `read_homer()`, `read_jaspar()`, `read_matrix()`, `read_meme()`, `read_motifs()`, `read_transfac()`, `read_uniprobe()`

Examples

```
cisbp <- read_cisbp(system.file("extdata", "cisbp.txt",
                                package = "universalmotif"))
```

---

read_homer *Import HOMER motifs.*

Description

Import HOMER formatted motifs. See http://homer.ucsd.edu/homer/motif/. Assumed to be DNA motifs. Note that HOMER motifs come with a pre-determined logodds threshold; if you wish to re-create HOMER's motif scanning, then use it in `scan_sequences()` (see examples).

Usage

```
read_homer(file, skip = 0)
```

Arguments

`file`	`character(1)` File name.
`skip`	`numeric(1)` If not zero, will skip however many desired lines in the file before starting to read.

Value

`list` universalmotif objects.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Heinz S, Benner C, Spann N, Bertolino E, Lin YC, Laslo P, Cheng JX, Murre C, Singh H, Glass CK (2010). "Simple combinations of lineage-determining transcription factors prime cis-regulatory elements required for macrophage and B cell identities." *Molecular Cell*, **38**, 576-589.

See Also

Other read_motifs: `read_cisbp()`, `read_jaspar()`, `read_matrix()`, `read_meme()`, `read_motifs()`, `read_transfac()`, `read_uniprobe()`

Examples

```
data(ArabidopsisPromoters)
homer <- read_homer(system.file("extdata", "homer.txt",
                                package = "universalmotif"))
thresholds <- homer |> to_df() |> with(logodds.threshold) |> as.numeric()
scan_sequences(homer, ArabidopsisPromoters,
  threshold = thresholds, threshold.type = "logodds.abs")
```

`read_jaspar` *Import JASPAR motifs.*

Description

Import JASPAR formatted motifs. See http://jaspar.genereg.net/. Can be either DNA, RNA, or AA motifs.

Usage

```
read_jaspar(file, skip = 0)
```

Arguments

`file`	`character(1)` File name.
`skip`	`numeric(1)` If not zero, will skip however many desired lines in the file before starting to read.

Value

`list` universalmotif objects.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Khan A, Fornes O, Stigliani A, Gheorghe M, Castro-Mondragon JA, van der Lee R, Bessy A, Cheneby J, Kulkarni SR, Tan G, Baranasic D, Arenillas DJ, Sandelin A, Vandepoele K, Lenhard B, Ballester B, Wasserman WW, Parcy F, Mathelier A (2018). "JASPAR 2018: update of the open-access database of transcription factor binding profiles and its web framework." *Nucleic Acids Research*, **46**, D260-D266.

See Also

Other read_motifs: `read_cisbp()`, `read_homer()`, `read_matrix()`, `read_meme()`, `read_motifs()`, `read_transfac()`, `read_uniprobe()`

Examples

```
jaspar <- read_jaspar(system.file("extdata", "jaspar.txt",
                                  package = "universalmotif"))
```

`read_matrix` *Import motifs from raw matrices.*

Description

Import simply formatted motifs.

Usage

```
read_matrix(file, skip = 0, type, positions = "columns",
  alphabet = "DNA", sep = "", headers = TRUE, rownames = FALSE,
  comment = NULL)
```

Arguments

`file`	`character(1)` File name.
`skip`	`numeric(1)` If not zero, will skip however many desired lines in the file before starting to read.
`type`	`character(1)` One of `c('PCM', 'PPM', 'PWM', 'ICM', 'CWM')`. If missing, will try and guess which one (note that `'CWM'` is never auto-detected; a signed matrix without explicit `type = "CWM"` is read as `'PWM'`, since the two have identical matrix shapes and CWM semantics need to be opted into explicitly).
`positions`	`character(1)` One of `c('columns', 'rows')`. Partial matching allowed. Indicate whether each position within a motif is represented as a row or a column in the file.
`alphabet`	`character(1)` One of `c('DNA', 'RNA', 'AA')`, or a string of letters.
`sep`	`character(1)` Indicates how individual motifs are separated. Set as `NULL` if there are no seperating lines between motifs (the default is to assume a blank line).
`headers`	`logical(1)`, `character(1)` Indicating if and how to read names.
`rownames`	`logical(1)` Are there alphabet letters present as rownames?
`comment`	`NULL`, `character(1)` Character denoting lines to be considered comments.

Value

`list` universalmotif objects.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

Other read_motifs: read_cisbp(), read_homer(), read_jaspar(), read_meme(), read_motifs(), read_transfac(), read_uniprobe()

Examples

```
hocomoco <- system.file("extdata", "hocomoco.txt", package = "universalmotif")
hocomoco <- read_matrix(hocomoco, headers = ">", positions = "rows")

## Reading a TF-MoDISco-style CWM dump: signed real values, no
## column-sum constraint. Opt in via type = "CWM".
## Not run:
cwms <- read_matrix("modisco_cwms.txt", headers = ">",
                    positions = "rows", type = "CWM")

## End(Not run)
```

---

read_meme *Import MEME motifs.*

Description

Import MEME formatted motifs, as well as original motif sequences. See http://meme-suite.org/doc/meme-format.html. Both 'full' and 'minimal' formats are supported. DREME and STREME motifs can also be imported, but note that readsites and readsites.meta arguments do nothing.

Usage

```
read_meme(file, skip = 0, readsites = FALSE, readsites.meta = FALSE,
  readsites.meta.tidy = FALSE, CWM = FALSE)
```

Arguments

file	character(1) File name.
skip	numeric(1) If not zero, will skip however many desired lines in the file before starting to read.
readsites	logical(1) If TRUE, the motif sites will be read as well.
readsites.meta	logical(1) If readsites = TRUE, then additionally read site positions and P-values.
readsites.meta.tidy	logical(1) If readsites.meta = TRUE, merge the position site information for all motifs into a single tidy data.frame.
CWM	logical(1) Treat each letter-probability matrix: block as a Contribution Weight Matrix (CWM) rather than a PPM. With CWM = TRUE, the parsed values pass through verbatim (signed real values, no column-sum or non-negativity check) and each resulting motif is tagged type = "CWM". This is the intended path for MEME files produced by TF-MoDISco or similar contribution-attribution pipelines, where the file format is MEME-shaped but the values are not probabilities. Defaults to FALSE (standard MEME-spec PPM parsing). See convert_type() for what a CWM is and how it converts onward.

Details

Please note that the typical number precision limit in R is around 1e-308. This means that motif P-values in MEME files below this limit are rounded automatically to 0. To get around this, the E-value is also stored as a string in the `extrainfo` slot. If you require a numeric value for analysis, use the `log_string_pval()` function to get the log of the string-formatted p-value.

Value

`list` universalmotif objects. If `readsites = TRUE`, a list comprising of a sub-list of motif objects and a sub-list of motif sites will be returned. If `readsites.meta = TRUE`, then two additional list items will be present, one containing site positions and P-values, and another containing combined sequence p-values. If `readsites.meta.tidy = TRUE`, an additional list entry named `sites.meta.tidy` will be added containing a `data.frame`.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Bailey TL, Boden M, Buske FA, Frith M, Grant CE, Clementi L, Ren J, Li WW, Noble WS (2009). "MEME SUITE: tools for motif discovery and searching." *Nucleic Acids Research*, **37**, W202-W208.

See Also

Other read_motifs: `read_cisbp()`, `read_homer()`, `read_jaspar()`, `read_matrix()`, `read_motifs()`, `read_transfac()`, `read_uniprobe()`

Examples

```
meme.minimal <- read_meme(system.file("extdata", "meme_minimal.txt",
                                      package = "universalmotif"))
meme.full <- read_meme(system.file("extdata", "meme_full.txt",
                                   package = "universalmotif"))
## Get numeric p-value:
log_string_pval(meme.minimal[[1]]["extrainfo"]["eval.string"])

## Read a TF-MoDISco-style MEME file as CWMs (no normalisation,
## values may be negative, type tag = "CWM"):
## Not run:
modisco <- read_meme("modisco_cwms.meme", CWM = TRUE)

## End(Not run)
```

read_motifs *Import universalmotif formatted motifs.*

Description

Import motifs created from `write_motifs()`. For optimal storage of `universalmotif` class motifs, consider using `saveRDS()` and `readRDS()`. Currently the `universalmotif` format is YAML-based, but this is subject to change.

Usage

```
read_motifs(file, skip = 0, progress = FALSE, BP = FALSE)
```

Arguments

`file`	`character(1)` File name.
`skip`	`numeric(1)` If not zero, will skip however many desired lines in the file before starting to read.
`progress`	`logical(1)` Show progress.
`BP`	`logical(1)` Allows for the use of **BiocParallel** within `read_motifs()`. See `BiocParallel::register()` to change the default backend.

Value

`list` universalmotif objects.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

Other read_motifs: `read_cisbp()`, `read_homer()`, `read_jaspar()`, `read_matrix()`, `read_meme()`, `read_transfac()`, `read_uniprobe()`

Examples

```
motif <- create_motif("CARGTG", name = "MyMotif")
tmp <- tempfile()
write_motifs(motif, tmp)
motif2 <- read_motifs(tmp)
```

`read_transfac` *Import TRANSFAC motifs.*

Description

Import TRANSFAC formatted motifs. Assumed to be DNA motifs, type PCM. See `system.file("extdata", "transfac.txt", pacakge="universalmotif")` for an example motif.

Usage

```
read_transfac(file, skip = 0)
```

Arguments

`file`	`character(1)` File name.
`skip`	`numeric(1)` If not zero, will skip however many desired lines in the file before starting to read.

Details

A few TRANSFAC tags are recognized, including AC, ID, NA, HC and OS. HC will be set to the family slot and OS to the organism slot. If AC, ID and NA are present, then AC will be set as the motif name and NA as the alternate name. If AC is absent, then ID is set as the name. If ID is also absent, then NA is set as the motif name.

Value

`list` universalmotif objects.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Wingender E, Dietze P, Karas H, Knuppel R (1996). "TRANSFAC: A Database on Transcription Factors and Their DNA Binding Sites." *Nucleic Acids Research*, **24**, 238-241.

See Also

Other read_motifs: `read_cisbp()`, `read_homer()`, `read_jaspar()`, `read_matrix()`, `read_meme()`, `read_motifs()`, `read_uniprobe()`

Examples

```
transfac <- read_transfac(system.file("extdata", "transfac.txt",
                                      package = "universalmotif"))
```

`read_uniprobe` *Import UNIPROBE motifs.*

Description

Import UNIPROBE formatted motifs. Assumed DNA.

Usage

```
read_uniprobe(file, skip = 0)
```

Arguments

`file`	`character(1)` File name.
`skip`	`numeric(1)` If not zero, will skip however many desired lines in the file before starting to read.

Value

`list` universalmotif objects.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Hume MA, Barrera LA, Gisselbrecht SS, Bulyk ML (2015). "UniPROBE, update 2015: new tools and content for the online database of protein-binding microarray data on protein-DNA interactions." *Nucleic Acids Research*, **43**, D117-D122.

See Also

Other read_motifs: `read_cisbp()`, `read_homer()`, `read_jaspar()`, `read_matrix()`, `read_meme()`, `read_motifs()`, `read_transfac()`

Examples

```
uniprobe.minimal <- read_uniprobe(system.file("extdata", "uniprobe_minimal.txt",
                                              package = "universalmotif"))
uniprobe.full <- read_uniprobe(system.file("extdata", "uniprobe_full.txt",
                                           package = "universalmotif"))
```

reexports *Objects exported from other packages*

Description

These objects are imported from other packages. Follow the links below to see their documentation.

BiocGenerics as.data.frame, cbind, colnames, ncol, normalize, nrow, rownames, subset

MatrixGenerics colMeans, colSums, rowMeans, rowSums

run_meme *Run MEME from within R.*

Description

De novo motif discovery via MEME. For a detailed description of the command, see http://meme-suite.org/doc/meme.html. For a brief description of the command parameters, call run_meme() without any arguments. Parameters in run_meme() which are directly taken from the MEME program are tagged with [MEME]. This function requires that the **processx** package be installed separately.

Usage

```
run_meme(target.sequences, output = NULL, overwrite.dir = FALSE,
  control.sequences = NULL, weights = NULL, text = FALSE, brief = 1000,
  objfun = "classic", test = NULL, use_llr = FALSE, shuf = 2,
  hsfrac = NULL, cefrac = NULL, searchsize = NULL, norand = FALSE,
  csites = 1000, seed = 0, alph = NULL, revcomp = FALSE, pal = FALSE,
  mod = "zoops", nmotifs = 3, evt = NULL, nsites = NULL,
  minsites = NULL, maxsites = NULL, wnsites = 0.8, w = NULL,
  minw = 8, maxw = 50, allw = NULL, nomatrim = FALSE, wg = 11,
  ws = 1, noendgaps = FALSE, bfile = NULL, markov_order = 0,
  psp = NULL, maxiter = 50, distance = 0.001, prior = NULL, b = NULL,
  plib = NULL, spfuzz = NULL, spmap = NULL, cons = NULL, p = NULL,
  maxsize = NULL, maxtime = NULL, wd = getwd(), logfile = paste0(wd,
  "/memerun.log"), readsites = TRUE, echo = FALSE, verbose = 1,
  timeout = Inf, bin = getOption("meme.bin"))
```

Arguments

target.sequences	XStringSet List of sequences to get motifs from.
output	character(1) Name of the output folder. If NULL, MEME output will be deleted.
overwrite.dir	logical(1) If output is set but already exists, allow over-writing.
control.sequences	XStringSet List of negative sequences. Only used if objfun = c("de", "se").
weights	numeric Vector of numbers between 0 and 1, representing sequence weights.

text	logical(1) [MEME]
brief	numeric(1) [MEME]
objfun	character(1) [MEME]
test	character(1) [MEME]
use_llr	logical(1) [MEME]
shuf	numeric(1) [MEME]
hsfrac	numeric(1) [MEME]
cefrac	numeric(1) [MEME]
searchsize	numeric(1) [MEME]
norand	logical(1) [MEME]
csites	numeric(1) [MEME]
seed	numeric(1) [MEME]
alph	character(1) [MEME] Note: custom alphabet definition files can be created using meme_alph().
revcomp	logical(1) [MEME]
pal	logical(1) [MEME]
mod	character(1) [MEME]
nmotifs	numeric(1) [MEME]
evt	numeric(1) [MEME]
nsites	numeric(1) [MEME]
minsites	numeric(1) [MEME]
maxsites	numeric(1) [MEME]
wnsites	numeric(1) [MEME]
w	numeric(1) [MEME]
minw	numeric(1) [MEME]
maxw	numeric(1) [MEME]
allw	numeric(1) [MEME]
nomatrim	logical(1) [MEME]
wg	numeric(1) [MEME]
ws	numeric(1) [MEME]
noendgaps	logical(1) [MEME]
bfile	character(1) [MEME]
markov_order	numeric(1) [MEME]
psp	character(1) [MEME]
maxiter	numeric(1) [MEME]
distance	numeric(1) [MEME]
prior	character(1) [MEME]
b	numeric(1) [MEME]
plib	character(1) [MEME]
spfuzz	numeric(1) [MEME]

spmap	character(1) [MEME]
cons	character(1) [MEME]
p	numeric(1) [MEME]
maxsize	numeric(1) [MEME]
maxtime	numeric(1) [MEME]
wd	character(1) Working directory to run MEME in.
logfile	character(1) File to dump MEME stderr. If NULL, no logs will be saved.
readsites	logical(1) Read sites from MEME output (from read_meme()).
echo	logical(1) Dump MEME output to console.
verbose	numeric(1) Set verbose = 0 to quiet run_meme().
timeout	numeric(1) Stop MEME program past timeout (seconds). See processx::run().
bin	character(1) Location of MEME binary. Alternatively, set this via options(meme.bin = '/path/to/meme/bin').

Value

list The output file is read with read_meme().

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Bailey TL, Elkan C (1994). "Fitting a mixture model by expectation maximization to discover motifs in biopolymers." *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*, **2**, 28-36.

See Also

read_meme(), create_sequences(), shuffle_sequences(), processx::run()

Examples

```
## Not run:
## To check that you are properly linking to the binary:
run_meme()

## End(Not run)
```

sample_sites *Generate binding sites from a motif.*

Description

Given probabilities for a sequence as represented by a motif, generate random sequences with the same length as the motif.

Usage

```
sample_sites(motif, n = 100, use.freq = 1)
```

Arguments

motif	See convert_motifs() for acceptable formats.
n	numeric(1) Number of sites to generate.
use.freq	numeric(1) If one, use regular motif matrix. Otherwise, use respective multifreq matrix.

Value

XStringSet object.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

create_sequences(), create_motif(), add_multifreq(), implant_motifs()

Examples

```
motif <- create_motif()
sites <- sample_sites(motif)
```

scan_sequences *Scan sequences for matches to input motifs.*

Description

For sequences of any alphabet, scan them using the PWM matrices of a set of input motifs. Note: for regular DNA/RNA scanning, see scan_sequences_lite() for a faster and simpler alternative.

Usage

```
scan_sequences(motifs, sequences, threshold = 1e-04,
  threshold.type = c("pvalue", "qvalue", "logodds", "logodds.abs"),
  RC = FALSE, use.freq = 1, verbose = 0, nthreads = 1,
  motif_pvalue.k = 8, use.gaps = TRUE, allow.nonfinite = FALSE,
  warn.NA = TRUE, calc.pvals = TRUE, return.granges = FALSE,
  no.overlaps = FALSE, no.overlaps.by.strand = FALSE,
  no.overlaps.strat = c("score", "order"), respect.strand = FALSE,
  motif_pvalue.method = c("dynamic", "exhaustive"),
  calc.qvals = calc.pvals, calc.qvals.method = c("fdr", "BH",
  "bonferroni"))
```

Arguments

motifs	See convert_motifs() for acceptable motif formats.
sequences	XStringSet Sequences to scan. Alphabet should match motif.
threshold	numeric(1) See details.
threshold.type	character(1) One of c('pvalue', 'qvalue', 'logodds', 'logodds.abs'). See details.
RC	logical(1) If TRUE, check reverse complement of the input sequences. Only available for DNA/RNA.
use.freq	numeric(1) The default, 1, uses the motif matrix (from the motif['motif'] slot) to search for sequences. If a higher number is used, then the matching k-let matrix from the motif['multifreq'] slot is used. See add_multifreq().
verbose	numeric(1) Describe progress, from none (0) to verbose (3).
nthreads	numeric(1) Run scan_sequences() in parallel with nthreads threads. nthreads = 0 uses all available threads. Note that no speed up will occur for jobs with only a single motif and sequence.
motif_pvalue.k	numeric(1) Control motif_pvalue() approximation. See motif_pvalue(). Only used if motif_pvalue.method = "exhaustive".
use.gaps	logical(1) Set this to FALSE to ignore motif gaps, if present.
allow.nonfinite	logical(1) If FALSE, then apply a pseudocount if non-finite values are found in the PWM. Note that if the motif has a pseudocount greater than zero and the motif is not currently of type PWM, then this parameter has no effect as the pseudocount will be applied automatically when the motif is converted to a PWM internally. This value is set to FALSE by default in order to stay consistent with pre-version 1.8.0 behaviour. Also note that this parameter is not compatible with motif_pvalue.method = "dynamic". A message will be printed if a pseudocount is applied. To disable this, set options(pseudocount.warning=FALSE).
warn.NA	logical(1) Whether to warn about the presence of non-standard letters in the input sequence, such as those in masked sequences.
calc.pvals	logical(1) Calculate P-values for each hit. This is a convenience option which simply gives motif_pvalue() the input motifs and the scores of each hit. Be careful about setting this to TRUE if you anticipate getting thousands of hits and are using motif_pvalue.method = "exhaustive": expect to wait a few seconds or minutes for the calculations to finish. Increasing the nthreads value can help greatly here. See Details for more information on P-value calculation. If motif_pvalue.method = "dynamic", then this is usually not an issue.

`return.granges` `logical(1)` Return the results as a `GRanges` object. Requires the `GenomicRanges` package to be installed.

`no.overlaps` `logical(1)` Remove overlapping hits from the same motifs. Overlapping hits from different motifs are preserved. Please note that the current implementation of this feature can add significantly to the run time for large inputs.

`no.overlaps.by.strand`
`logical(1)` Whether to discard overlapping hits from the opposite strand (`TRUE`), or to only discard overlapping hits on the same strand (`FALSE`).

`no.overlaps.strat`
`character(1)` One of `c("score", "order")`. The former option keeps the highest scoring overlapping hit (and the first of these within ties), and the latter simply keeps the first overlapping hit.

`respect.strand` `logical(1)` If motifs are DNA/RNA, then setting this option to `TRUE` will make `scan_sequences()` only scan the strands of the input sequences as indicated in the motif `strand` slot.

`motif_pvalue.method`
`character(1)` One of `c("dynamic", "exhaustive")`. Algorithm used for calculating P-values. The `"exhaustive"` method involves finding all possible motif matches at or above the specified score using a branch-and-bound algorithm, which can be computationally intensive (Hartman et al., 2013). Additionally, the computation must be repeated for each hit. The `"dynamic"` method calculates the distribution of possible motif scores using a much faster dynamic programming algorithm, and can be recycled for multiple scores (Grant et al., 2011). The only disadvantage is the inability to use `allow.nonfinite = TRUE`. See `motif_pvalue()` for details.

`calc.qvals` `logical(1)` Whether to also calculate adjusted P-values. Only valid if `calc.pvals = TRUE`.

`calc.qvals.method`
`character(1)` One of `c("fdr", "BH", "bonferroni")`. The method for calculating adjusted P-values. These are described in depth in the Sequence Searches vignette. Also see Noble (2009).

Details

Logodds scoring:

Similar to `Biostrings::matchPWM()`, the scanning method uses logodds scoring. (To see the scoring matrix for any motif, simply run `convert_type(motif, "PWM")`. For a `multifreq` scoring matrix: `apply(motif["multifreq"][["2"]], 2, ppm_to_pwm)`). In order to score a sequence, at each position within a sequence of length equal to the length of the motif, the scores for each base are summed. If the score sum is above the desired threshold, it is kept.

Thresholds:

If `threshold.type = 'logodds'`, then the `threshold` value is multiplied by the maximum possible motif scores. To calculate the maximum possible scores a motif (of type PWM) manually, run `motif_score(motif, 1)`. If `threshold.type = 'pvalue'`, then threshold logodds scores are generated using `motif_pvalue()`. Finally, if `threshold.type = 'logodds.abs'`, then the exact values provided will be used as thresholds. Finally, if `threshold.type = 'qvalue'`, then the threshold is calculated as if `threshold.type = 'pvalue'` and the final set of hits are filtered based on their calculated Q-value. (Note: this means that the `thresh.score` column will be incorrect!) This is done since most Q-values cannot be calculated prior to scanning. If you are

running a very large job, it may be wise to use a P-value threshold followed by manually filtering by Q-value; this will avoid the scanning have to parse the larger number of hits from the internally-lowered threshold.

Non-standard letters:
Non-standard letters (such as "N", "+", "-", ".", etc in DNAString objects) will be safely ignored, resulting only in a warning and a very minor performance cost. This can used to scan masked sequences. See Biostrings::mask() for masking sequences (generating MaskedXString objects), and Biostrings::injectHardMask() to recover masked XStringSet objects for use with scan_sequences(). There is also a provided wrapper function which performs both steps: mask_seqs().

Value

DataFrame, GRanges with each row representing one hit. If the input sequences are DNAStringSet or RNAStringSet, then an additional column with the strand is included. Function args are stored in the metadata slot. If return.granges = TRUE then a GRanges object is returned.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Grant CE, Bailey TL, Noble WS (2011). "FIMO: scanning for occurrences of a given motif." *Bioinformatics*, **27**, 1017-1018.

Hartmann H, Guthohrlein EW, Siebert M, Soding SLJ (2013). "P-value-based regulatory motif discovery using positional weight matrices." *Genome Research*, **23**, 181-194.

Noble WS (2009). "How does multiple testing work?" *Nature Biotechnology*, **27**, 1135-1137.

See Also

add_multifreq(), Biostrings::matchPWM(), enrich_motifs(), motif_pvalue(), scan_sequences_lite()

Examples

```
## any alphabet can be used
## Not run:
set.seed(1)
alphabet <- paste(c(letters), collapse = "")
motif <- create_motif("hello", alphabet = alphabet)
sequences <- create_sequences(alphabet, seqnum = 1000, seqlen = 100000)
scan_sequences(motif, sequences)

## End(Not run)

## Sequence masking:
if (R.Version()$arch != "i386") {
library(Biostrings)
data(ArabidopsisMotif)
data(ArabidopsisPromoters)
seq <- mask_seqs(ArabidopsisPromoters, "AAAAA")
scan_sequences(ArabidopsisMotif, seq)
# A warning regarding the presence of non-standard letters will be given,
# but can be safely ignored in this case.
```

```
}
```

`scan_sequences_lite` *Faster minimalist motif scanner.*

Description

`scan_sequences_lite()` is a deliberately pared-down counterpart to `scan_sequences()`. It exposes a single threshold (a P-value), always scans both strands by default, always computes a per-hit P-value, and returns either a `GRanges` (preferred, when GenomicRanges is installed) or a `data.frame`. Use `scan_sequences()` when you need any of multifreq scoring, gapped motifs, q-values, exhaustive P-values, `respect.strand`, `allow.nonfinite`, or `threshold.types` other than a P-value.

Usage

```
scan_sequences_lite(motifs, sequences, pvalue = 1e-04, RC = TRUE,
  nthreads = 1, return.granges = NULL, no.overlaps = FALSE,
  no.overlaps.by = c("pvalue", "score"), no.overlaps.by.strand = FALSE,
  no.overlaps.by.motif = FALSE)
```

Arguments

`motifs`	See `convert_motifs()` for accepted motif formats. DNA or RNA only – amino-acid and custom alphabet motifs are rejected.
`sequences`	`XStringSet`. DNA or RNA sequences to scan; the alphabet must match the motif alphabet.
`pvalue`	`numeric(1)`. P-value cutoff for reporting hits. Default `1e-4`.
`RC`	`logical(1)`. If `TRUE` (default), scan both strands.
`nthreads`	`numeric(1)`. Number of threads. `nthreads = 0` uses all available threads. Parallelisation happens inside `scan_sequences_cpp` over the motif x sequence cross-product.
`return.granges`	`logical(1)` or `NULL`. When `NULL` (default), returns a `GRanges` if the `GenomicRanges` package is installed, otherwise a `data.frame`. Set `TRUE` / `FALSE` to force one or the other.
`no.overlaps`	`logical(1)`. If `TRUE`, drop overlapping hits within each `(sequence, motif, strand)` group using a greedy algorithm (see `dedup_hits()`). Default `FALSE`.
`no.overlaps.by`	`character(1)`. Which column to use as the priority for breaking ties between overlapping hits. `"pvalue"` (default) keeps the lowest p-value; `"score"` keeps the highest log-odds score. Only used when `no.overlaps = TRUE`.
`no.overlaps.by.strand`	`logical(1)`. If `TRUE`, overlapping hits on opposite strands compete with each other. If `FALSE` (default), each strand is deduplicated independently.
`no.overlaps.by.motif`	`logical(1)`. If `TRUE`, overlapping hits from different motifs compete with each other. If `FALSE` (default), each motif is deduplicated independently.

Details

The scanner re-uses the very same C++ core as `scan_sequences()`; the speed advantage simply comes from skipping the R-side bookkeeping for the features this function does not support.

P-values are computed via the dynamic-programming algorithm of `motif_pvalue()` (FIMO-style; Grant et al. 2011).

Value

Either a `GRanges` or a `data.frame`.

Common fields (both shapes): `motif`, `motif.i`, `sequence.i`, `score`, `score.pct`, `match`, `pvalue`.

Coordinate fields:

- `GRanges`: `seqnames` (sequence name), `start`, `end`, `strand`.
- `data.frame`: `sequence`, `start`, `end`, `strand`.

Coordinates are always 1-based, inclusive, with `start` <= `end` regardless of strand. For hits on the `-` strand the `match` column is the reverse complement of the sequence substring; i.e. the sequence as matched against the motif.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Grant CE, Bailey TL, Noble WS (2011). "FIMO: scanning for occurrences of a given motif." *Bioinformatics*, **27**(7), 1017-1018.

See Also

`scan_sequences()`, `motif_pvalue()`, `convert_motifs()`

Other lite motif functions: `compare_motifs_lite()`, `enrich_motifs_lite()`, `merge_motifs_lite()`, `merge_similar_lite()`, `motif_tree_lite()`, `view_motifs_lite()`

Examples

```
library(universalmotif)
motifs <- create_motif(c("TATAAA", "CACGTG"))
seqs <- create_sequences(seqnum = 5, seqlen = 200, rng.seed = 1)
hits <- scan_sequences_lite(motifs, seqs, pvalue = 1e-3, return.granges = FALSE)
head(hits)
```

`sequence_complexity` *Calculate sequence complexity.*

Description

Calculate sequence complexity using either the Wootton-Federhen, Trifonov, or DUST algorithms.

Usage

```
sequence_complexity(seqs, window.size = 20,
  window.overlap = round(window.size/2), method = c("WoottonFederhen",
  "WoottonFederhenFast", "Trifonov", "TrifonovFast", "DUST"),
  trifonov.max.word.size = 7, nthreads = 1, return.granges = FALSE)
```

Arguments

`seqs`	`XStringSet` Input sequences.
`window.size`	`numeric` Window size. If a number between 0 and 1 is provided, the value is calculated as the number multiplied by the sequence length.
`window.overlap`	`numeric` Overlap between windows. If a number between 0 and 1 is provided, the value is calculated as the number multiplied by the sequence length.
`method`	`character(1)` Choose one of the available methods for calculating sequence complexity. See details.
`trifonov.max.word.size`	`numeric(1)` The maximum word size within each window used to calculate complexity using `method = c("Trifonov", "TrifonovFast")`. In other words, the Trifonov method involves counting the number of possible different sub-words in a window at different sizes up to the values provided by this option. It also involves calculating the product of ever increasing sequences of numbers and so in order to reduce the computations involed this can be limited to a specific maximum sub-word size.
`nthreads`	`numeric(1)` Run `sequence_complexity()` in parallel with `nthreads` threads. `nthreads = 0` uses all available threads.
`return.granges`	`logical(1)` Return the results as a `GRanges` object. Requires the `GenomicRanges` package to be installed.

Details

The Wootton-Federhen (Wootton and Federhen, 1993) and Trifonov (Trifonov, 1990) algorithms as well as their faster approximations are well described within Orlov and Potapov (2004). These algorithms score less complex sequences closer to 0, and more complex ones closer to 1. Please note that the 'fast' approximation versions of the two methods are not actually faster within `sequence_complexity()`, and so speed should not be a major consideration when choosing which method to use within the `universalmotif` package. The DUST algorithm implementation is described in Morgulis et al. (2006). In this case, less complex sequences score higher, and more complex ones closer to 0.

Briefly, the Wootton-Federhen complexity score is a reflection of the numbers of each unique letter found in the window (e.g. for DNA, the more of all four letters can be found in the window the higher the score). An increasing Trifonov score is a relection of the numbers of increasingly larger k-mers (e.g. the count of possible 1-mers, 2-mers, 3-mers, ..., until `trifonov.max.word.size`).

Finally, the DUST score approaches 0 as the count of unique 3-mers increases. (See the final section in the examples to see how different types of sequence compositions affect the methods.)

Please note that the authors of the different methods recommend various window sizes and complexity thresholds. The authors of DUST for example, suggest using a window size of 64 and a threshold of 2 for low complexity. Wootton and Federhen suggest a window size of 40, though show that 10 and 20 can be appropriate as well (for amino acid sequences). Keep in mind however that these algorithms were implemented at a time when computers were much slower; perhaps the authors would suggest different window sizes today. One thing to note is that the Wootton-Federhen algorithm has a hard limit due to the need to caculate the product from `1:window.size`. This can end up calculating values which are greater than what a double can hold (e.g. try `prod(1:500)`). Its approximation does not suffer from this though, as it skips calculating the product.

In terms of speed, the Wootton-Federhen algorithms are fastest, with DUST being 1-3 times slower and the Trifonov algorithms being several times slower (though the exact amount depends on the max word size).

Value

`DataFrame`, `GRanges` with each row getting a complexity score for each window in each input sequence.

Author(s)

Benjamin Jean-Marie Tremblay, `<benjamin.tremblay@uwaterloo.ca>`

References

Morgulis A, Gertz EM, Schaffer AA, Agarwala R (2006). "A fast and symmetric DUST implementation to mask low-complexity DNA sequences." *Journal of Computational Biology*, **13**, 1028-1040.

Orlov YL, Potapov VN (2004). "Complexity: an internet resource for analysis of DNA sequence complexity." *Nucleic Acids Research*, **32**, W628-W633.

Trifonov EN (1990). "Making sense of the human genome." In Sarma RH, Sarma MH (Eds), *Structure & Methods* Adenine Press, Albany, **1**, 69-77.

Wootton JC, Federhen S (1993). "Statistics of local complexity in amino acid sequences and sequence databases." *Computers & Chemistry*, **17**, 149-163.

See Also

`calc_complexity()`, `count_klets()`, `get_bkg()`, `mask_ranges()`, `mask_seqs()`

Examples

```
## Feel free to play around with different toy sequences to get a feel for
## how the different methods perform

library(Biostrings)
test.seq <- DNAStringSet(c("AAAAAAAAAAA", "ATGACTGATGC"))

sequence_complexity(test.seq, method = "WoottonFederhen")
sequence_complexity(test.seq, method = "WoottonFederhenFast")
sequence_complexity(test.seq, method = "Trifonov")
sequence_complexity(test.seq, method = "TrifonovFast")
sequence_complexity(test.seq, method = "DUST")
```

```
## You could also use this in conjuction with mask_ranges() to hide
## low complexity regions from scanning, de novo motif discovery, etc

if (requireNamespace("GenomicRanges", quietly = TRUE)) {
data(ArabidopsisPromoters)

# Calculate complexity in 20 bp windows, sliding every 1 bp
to.mask <- sequence_complexity(ArabidopsisPromoters, method = "DUST",
  window.size = 20, window.overlap = 19, return.granges = TRUE)

# Get the ranges with a complexity score greater than 3.5
to.mask <- to.mask[to.mask$complexity > 3.5]

# See what the low complexity regions look like
ArabidopsisPromoters[IRanges::reduce(to.mask)]

# Mask them with the default '-' character:
mask_ranges(ArabidopsisPromoters, to.mask)
}

## To demonstrate how the methods work, consider:
## (These examples use the calc_complexity() utility which utilizes
## the same algorithms and works on character vectors, but lacks
## the ability to use sliding windows.)
a <- "ACGT"

# For Wootton-Federhen, it can be easily shown it is only dependent
# on the counts of individual letters (though do note that the
# original paper discusses this method in the context of amino acid
# sequences and not DNA):
calc_complexity("AAACCCGGGTTT", alph = a)  # 0.7707
calc_complexity("AACCGGTTACGT", alph = a)  # 0.7707
calc_complexity("ACGTACGTACGT", alph = a)  # 0.7707

# As letters start to see drops in counts, the scores go down too:
calc_complexity("AAAACCCCGGGG", alph = a)  # 0.6284
calc_complexity("AAAAAACCCCCC", alph = a)  # 0.4105
calc_complexity("AAAAAAAAAACC", alph = a)  # 0.2518

# Trifonov on the other hand is greatly affected by the number
# of higher order combinations:
calc_complexity("AAACCCGGGTTT", c = "Trifonov", alph = a)  # 0.6364
calc_complexity("AACCGGTTACGT", c = "Trifonov", alph = a)  # 0.7273

# This next one may seem surprising, but it indeed scores very low.
# This is because although it has many of each individual letter,
# the number of higher order letter combinations in fact is quite
# low due to this particular repeating pattern!
calc_complexity("ACGTACGTACGT", c = "Trifonov", alph = a)  # 0.01231

# By extension, this means it scores sequences with fewer
# counts of individual letters lower too.
calc_complexity("AAAACCCCGGGG", c = "Trifonov", alph = a)  # 0.2386
calc_complexity("AAAAAACCCCCC", c = "Trifonov", alph = a)  # 0.0227
calc_complexity("AAAAAAAAAACC", c = "Trifonov", alph = a)  # 0.0011

# As for DUST, it considers the number of 3-mers in the sequence.
```

```
# The higher the numbers of 3-mers, the lower the score.
# (0 = the max possible number of DNA 3-mers for the window size)
calc_complexity("AAACCCGGGTTT", c = "DUST", alph = a)  # 0
calc_complexity("AACCGGTTACGT", c = "DUST", alph = a)  # 0
calc_complexity("ACGTACGTACGT", c = "DUST", alph = a)  # 0.8889
calc_complexity("AAAACCCCGGGG", c = "DUST", alph = a)  # 0.333
calc_complexity("ACGACGACGACG", c = "DUST", alph = a)  # 1.333
calc_complexity("AAAAAACCCCCC", c = "DUST", alph = a)  # 1.333
# Similarly to Trifonov, the next one also scores as less complex
# compared to the previous one:
calc_complexity("ACACACACACAC", c = "DUST", alph = a)  # 2.222
calc_complexity("AAAAAAAAAACC", c = "DUST", alph = a)  # 3.111
calc_complexity("AAAAAAAAAAAC", c = "DUST", alph = a)  # 4
calc_complexity("AAAAAAAAAAAA", c = "DUST", alph = a)  # 5

# Just to show once more why the seemingly more complex sequences
# such as "ACACACACACAC" score as less complex than "AAAAAACCCCCC"
# for the Trifonov and DUST methods:
count_klets("ACACACACACAC", k = 3)  # Only 2 possible 3-mers
count_klets("AAAAAACCCCCC", k = 3)  # Now 4 possible 3-mers!
```

`shuffle_motifs` *Shuffle motifs by column.*

Description

Given a set of motifs, shuffle the columns to create new motifs. Currently does not support keeping the 'multifreq' slot. Only the 'bkg', 'nsites', 'strand', and 'bkgsites' slots will be preserved. Uses the same shuffling methods as `shuffle_sequences()`. When shuffling more than one motif, all motif columns are merged into a single pool and shuffled together, finally returning them as motifs of identical lengths as the input motifs. To instead shuffle motifs individually, call `shuffle_motifs()` using `lapply()`.

Usage

```
shuffle_motifs(motifs, k = 2, method = "linear")
```

Arguments

`motifs`	See `convert_motifs()` for acceptable formats.
`k`	`numeric(1)` K-let size.
`method`	`character(1)` Currently only 'linear' is accepted.

Value

Motifs. See `convert_motifs()` for available output formats.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

shuffle_sequences()

Examples

```
motif <- create_motif("CARGTGNN", nsites = 100)
motif.shuffled <- shuffle_motifs(motif, k = 2)
```

---

shuffle_sequences *Shuffle input sequences.*

Description

Given a set of input sequences, shuffle the letters within those sequences with any k-let size.

Usage

```
shuffle_sequences(sequences, k = 1, method = "euler", nthreads = 1,
  rng.seed = sample.int(10000, 1), window = FALSE, window.size = 0.1,
  window.overlap = 0.01)
```

Arguments

sequences	XStringSet Set of sequences to shuffle. Works with any set of characters.
k	numeric(1) K-let size.
method	character(1) One of c('euler', 'markov', 'linear'). Only relevant if k > 1. See details.
nthreads	numeric(1) Run shuffle_sequences() in parallel with nthreads threads. nthreads = 0 uses all available threads. Note that no speed up will occur for jobs with only a single sequence.
rng.seed	numeric(1) Set random number generator seed. Since shuffling can occur simultaneously in multiple threads using C++, it cannot communicate with the regular R random number generator state and thus requires an independent seed. Each individual sequence in an XStringSet object will be given the following seed: rng.seed * index. The default is to pick a random number as chosen by sample(), which effectively is making shuffle_sequences() dependent on the R RNG state.
window	logical(1) Shuffle sequences iteratively over windows instead of all at once.
window.size	numeric(1) Window size. Can be a fraction less than one, or an integer representing the actual window size.
window.overlap	numeric(1) Overlap between windows. Can be a fraction less than one, or an integer representing the actual overlap size.

Details

markov method:

If `method = 'markov'`, then the Markov model is used to generate sequences which will maintain (on average) the k-let frequencies. Please note that this method is not a 'true' shuffling, and for short sequences (e.g. <100bp) this can result in slightly more dissimilar sequences versus true shuffling. See Fitch (1983) for a discussion on the topic.

euler method:

If `method = 'euler'`, then the sequence shuffling method proposed by Altschul and Erickson (1985) is used. As opposed to the 'markov' method, this one preserves exact k-let frequencies. This is done by creating a k-let edge graph, then following a random Eulerian walk through the graph. Not all walks will use up all available letters however, so the cycle-popping algorithm proposed by Propp and Wilson (1998) is used to find a random Eulerian path. A side effect of using this method is that the starting and ending sequence letters will remain unshuffled.

linear method:

If `method = 'linear'`, then the input sequences are split linearly every k letters. For example, for `k = 3` 'ACAGATAGACCC' becomes 'ACA GAT AGA CCC'; after which these 3-lets are shuffled randomly.

Single-letter shuffling:

Do note however, that the `method` parameter is only relevant for `k > 1`. For `k = 1`, a simple shuffling is performed using the `shuffle` function from the C++ standard library.

Value

`XStringSet` The input sequences will be returned with identical names and lengths.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Altschul SF, Erickson BW (1985). "Significance of Nucleotide Sequence Alignments: A Method for Random Sequence Permutation That Preserves Dinucleotide and Codon Usage." *Molecular Biology and Evolution*, **2**, 526-538.

Fitch WM (1983). "Random sequences." *Journal of Molecular Biology*, **163**, 171-176.

Propp JG, Wilson DW (1998). "How to get a perfectly random sample from a generic markov chain and generate a random spanning tree of a directed graph." *Journal of Algorithms*, **27**, 170-217.

See Also

`create_sequences()`, `scan_sequences()`, `enrich_motifs()`, `shuffle_motifs()`

Examples

```
if (R.Version()$arch != "i386") {
sequences <- create_sequences()
sequences.shuffled <- shuffle_sequences(sequences, k = 2)
}
```

switch_alph *Switch between DNA and RNA alphabets.*

Description

Convert a motif from DNA to RNA, or RNA to DNA.

Usage

```
switch_alph(motifs)
```

Arguments

motifs	See convert_motifs() for acceptable formats.

Value

The DNA/RNA version of the motifs. See convert_motifs() for acceptable output formats.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

create_motif()

Examples

```
DNA.motif <- create_motif()
RNA.motif <- switch_alph(DNA.motif)
```

tidy-motifs *Tidy manipulation of motifs.*

Description

Tidy manipulation of motifs.

Usage

```
to_df(motifs, extrainfo = TRUE)

update_motifs(motif_df, extrainfo = TRUE, force = FALSE)

to_list(motif_df, extrainfo = TRUE, force = FALSE)

requires_update(motifs, extrainfo = TRUE)
```

Arguments

motifs	List of motifs.
extrainfo	Use the extrainfo slot in the tidy data.frame. The column names will be taken from the character vectors themselves, and unnamed elements will be assigned a unique name. To add elements to the slot, simply create new columns in the data.frame. Note that these will be coerced into characters. If extrainfo is not set to TRUE in to_df(), then the contents of the slot will not be transferred to the data.frame. If extrainfo is not set to TRUE in update_motifs() or to_list(), then the extra columns will be discarded.
motif_df	Motif data.frame generated by to_df().
force	Whether to coerce non-character data types into characters for inclusion in extrainfo. If force is FALSE (the default), columns which are not of type "character", "numeric", or "integer" (for example, list columns, or logical values), will not be added to the motif extrainfo slot, but will be passed onto the returned universalmotif_df unchanged. Setting force = TRUE coerces these values into a character, adding them to the extrainfo slot, and updating the universalmotif_df columns to reflect this coercion. In other words, forcing inclusion of these data is destructive and will change the column values. Use with caution.

Details

To turn off the informative messages/warnings when printing the object to the console, set options(universalmotif_df

Value

For to_df(): a data.frame with the exposed slots as columns.

For update_motifs(): the updated data.frame.

For requires_update(): TRUE if the motifs are out of date, FALSE if otherwise. Note that this function uses identical() to check for this, which can be quite slow for large datasets. It is usually just as fast to simply run update_motifs() in such cases.

For to_list(): a list of motifs.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

Examples

```
## Not run:
library(universalmotif)
library(dplyr)

m <- c(create_motif(name = "motif A"), create_motif(name = "motif B"))

# Change the names of the motifs using the tidy way:
m <- m %>%
   to_df() %>%
   mutate(name = paste0(name, "-2")) %>%
   to_list()

# Add your own metadata to be stored in the extrainfo slot:
m_df <- to_df(m)
```

```
m_df$MyMetadata <- c("Info_1", "Info_2")
m <- to_list(m_df, extrainfo = TRUE)

## End(Not run)
```

`trim_cwm` *Trim CWM motifs by absolute column sum.*

Description

Trim the low-contribution edge columns of one or more Contribution Weight Matrix (CWM) motifs. CWMs do not have a probabilistic interpretation, so `trim_motifs()`'s information-content threshold is not the natural cutoff; instead, `trim_cwm()` drops edge columns whose absolute column sum (`sum_i |cwm[i,j]|`) falls below a threshold.

Usage

```
trim_cwm(motifs, trim.threshold = 0.3, abs.threshold = NULL,
  trim.from = c("both", "left", "right"))
```

Arguments

`motifs`	A CWM motif or list of CWM motifs (`type = "CWM"`).
`trim.threshold`	`numeric(1)`. Fraction-of-peak cutoff, used when `abs.threshold` is `NULL`. Default `0.3`.
`abs.threshold`	`numeric(1)` or `NULL`. Absolute-value cutoff on `sum_i |cwm[i,j]|`. When non-`NULL`, takes precedence over `trim.threshold`. Default `NULL`.
`trim.from`	`character(1)`. One of `"both"`, `"left"`, or `"right"`. Direction of edge trimming.

Details

Two threshold modes are supported. By default, `trim_cwm()` uses the TF-MoDISco-lite fraction-of-peak rule: drop edge columns whose absolute column sum is below `trim.threshold * max_j(sum_i |cwm[i,j]|)`, with `trim.threshold = 0.3` matching TF-MoDISco-lite's `--trim_threshold` default. When `abs.threshold` is supplied (non-`NULL`), the function switches to an absolute-value cutoff: edge columns with absolute column sum below `abs.threshold` are dropped, ignoring `trim.threshold`.

In both modes the trimmer walks inward only from the chosen edges (`trim.from`), matching `trim_motifs()`'s edge-only semantics; interior low-contribution columns are kept.

Value

A trimmed CWM motif (single input) or list of trimmed CWM motifs (list input). The `type = "CWM"` tag is preserved.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

trim_motifs(), create_motif(), convert_type()

Examples

```
library(universalmotif)

## Synthetic CWM with a strong centre and weak flanks.
m <- matrix(c(0.02, -0.01,  0.01,  0.00,
              0.90, -0.30, -0.30, -0.30,
             -0.20,  0.80, -0.20, -0.20,
             -0.30, -0.30, -0.30,  0.90,
              0.03,  0.00, -0.02,  0.01),
            nrow = 4, byrow = FALSE,
            dimnames = list(c("A","C","G","T"), NULL))
cwm <- create_motif(m, type = "CWM", name = "demo")

## Default: drop edge columns where |colsum| < 0.3 * peak.
trimmed <- trim_cwm(cwm)
ncol(trimmed["motif"])

## Explicit absolute-value cutoff instead.
trim_cwm(cwm, abs.threshold = 0.5)

## One-sided trim.
trim_cwm(cwm, trim.from = "left")
```

---

trim_motifs *Trim motifs.*

Description

Remove edges of motifs with low information content. Currently does not trim `multifreq` representations.

Usage

```
trim_motifs(motifs, min.ic = 0.25, trim.from = c("both", "left", "right"))
```

Arguments

motifs	See convert_motifs() for acceptable formats.
min.ic	numeric(1) Minimum allowed information content. See convert_type() for a discussion on information content.
trim.from	character(1) Control the direction of trimming.

Value

Motifs See convert_motifs() for available output formats.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

create_motif(), convert_type()

Examples

```
jaspar <- read_jaspar(system.file("extdata", "jaspar.txt",
                                  package = "universalmotif"))
jaspar.trimmed <- trim_motifs(jaspar)
```

---

universalmotif-class *universalmotif: Motif class.*

Description

Container for motif objects. See create_motif() for creating motifs as well as a more detailed description of the slots. For a brief description of available methods, see examples.

Usage

```
## S4 method for signature 'universalmotif'
x[i]

## S4 replacement method for signature 'universalmotif'
x[i] <- value

## S4 method for signature 'universalmotif'
initialize(.Object, name, altname, family, organism,
  motif, alphabet = "DNA", type, icscore, nsites, pseudocount = 1, bkg,
  bkgsites, consensus, strand = "+-", pval, qval, eval, multifreq, extrainfo,
  gapinfo)

## S4 method for signature 'universalmotif'
show(object)

## S4 method for signature 'universalmotif'
as.data.frame(x)

## S4 method for signature 'universalmotif'
subset(x, select)

## S4 method for signature 'universalmotif'
normalize(object)

## S4 method for signature 'universalmotif'
rowMeans(x)
```

```
## S4 method for signature 'universalmotif'
colMeans(x)

## S4 method for signature 'universalmotif'
colSums(x)

## S4 method for signature 'universalmotif'
rowSums(x)

## S4 method for signature 'universalmotif'
nrow(x)

## S4 method for signature 'universalmotif'
ncol(x)

## S4 method for signature 'universalmotif'
colnames(x)

## S4 method for signature 'universalmotif'
rownames(x)

## S4 method for signature 'universalmotif'
cbind(..., deparse.level = 0)
```

Arguments

x	universalmotif Motif.
i	character Slot.
value	Object to replace slot with.
.Object	universalmotif Final motif.
name	character(1) Motif name.
altname	character(1) Alternate motif name.
family	character(1) Transcription factor family.
organism	character(1) Species of origin.
motif	matrix Each column represents a position in the motif.
alphabet	character(1) One of c('DNA', 'RNA', 'AA'), or a combined string representing the letters.
type	character(1) One of c('PCM', 'PPM', 'PWM', 'ICM').
icscore	numeric(1) Total information content. Automatically generated.
nsites	numeric(1) Number of sites the motif was constructed from.
pseudocount	numeric(1) Correction to be applied to prevent -Inf from appearing in PWM matrices.
bkg	numeric A vector of probabilities, each between 0 and 1. If higher order backgrounds are provided, then the elements of the vector must be named.
bkgsites	numeric(1) Total number of sites used to find the motif.
consensus	character(1) Consensus string. Automatically generated for 'DNA', 'RNA', and 'AA' alphabets.
strand	character(1) Whether the motif is specific to a certain strand.

pval	numeric(1) P-value associated with motif.
qval	numeric(1) Adjusted P-value associated with motif.
eval	numeric(1) E-value associated with motif.
multifreq	list See add_multifreq().
extrainfo	character Any other extra information, represented as a named character vector.
gapinfo	universalmotif_gapped(1) Gapped motif information.
object	universalmotif Motif.
select	numeric Columns to keep.
...	universalmotif Motifs.
deparse.level	Unused.

Value

A motif object of class universalmotif.

Slots

name character(1)

altname character(1)

family character(1)

organism character(1)

motif matrix

alphabet character(1)

type character(1)

icscore numeric(1) Generated automatically.

nsites numeric(1)

pseudocount numeric(1)

bkg numeric 0-order probabilities must be provided for all letters.

bkgsites numeric(1)

consensus character Generated automatically.

strand character(1)

pval numeric(1)

qval numeric(1)

eval numeric(1)

multifreq list

extrainfo character

gapinfo universalmotif_gapped(1)

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

Examples

```
## [
## Access the slots.
motif <- create_motif()
motif["motif"]
# you can also access multiple slots at once, released as a list
motif[c("motif", "name")]

## [<-
## Replace the slots.
motif["name"] <- "new name"
# some slots are protected
# motif["consensus"] <- "AAAA"  # not allowed

## c
## Assemble a list of motifs.
c(motif, motif)

## as.data.frame
## Represent a motif as a data.frame. The actual motif matrix is lost.
## Necessary for `summarise_motifs`.
as.data.frame(motif)

## subset
## Subset a motif matrix by column.
subset(motif, 3:7)  # extract motif core

## normalize
## Apply the pseudocount slot (or `1`, if the slot is set to zero) to the
## motif matrix.
motif2 <- create_motif("AAAAA", nsites = 100, pseudocount = 1)
normalize(motif2)

## rowMeans
## Calculate motif rowMeans.
rowMeans(motif)

## colMeans
## Calculate motif colMeans.
colMeans(motif)

## colSums
## Calculate motif colSums
colSums(motif)

## rowSums
## Calculate motif rowSums.
rowSums(motif)

## nrow
## Count motif rows.
nrow(motif)

## ncol
## Count motif columns.
ncol(motif)
```

```
## colnames
## Get motif colnames.
colnames(motif)

## rownames
## Get motif rownames.
rownames(motif)

## cbind
## Bind motifs together to create a new motif.
cbind(motif, motif2)
```

`universalmotif-pkg` *universalmotif: A Toolkit for Sequence Motif Analysis*

Description

A collection of utility functions for working with motifs.

Author(s)

Maintainer: Benjamin Jean-Marie Tremblay <benjamin.tremblay@uwaterloo.ca> (ORCID)

Other contributors:

- Sabrina Nystrom (ORCID) [contributor]

See Also

Useful links:

- https://bioconductor.org/packages/universalmotif/
- Report bugs at https://github.com/bjmt/universalmotif/issues

---

`utilities` *Utility functions.*

Description

Utility functions have been split into two categories: those related to motifs ?'utils-motif', and those related to sequences ?'utils-sequence'.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

utils-motif, utils-sequence

utils-motif *Motif-related utility functions.*

Description

Motif-related utility functions.

Usage

```
add_gap(motif, gaploc = ncol(motif)%/%2, mingap = 1, maxgap = 5)

average_ic(motifs, average = c("a.mean", "g.mean", "median", "fzt"))

compare_columns(x, y, method, bkg1 = rep(1/length(x), length(x)),
  bkg2 = rep(1/length(y), length(y)), nsites1 = 100, nsites2 = 100)

consensus_to_ppm(letter)

consensus_to_ppmAA(letter)

get_consensus(position, alphabet = "DNA", type = "PPM", pseudocount = 1)

get_consensusAA(position, type = "PPM", pseudocount = 0)

get_matches(motif, score, allow.nonfinite = FALSE)

get_scores(motif, allow.nonfinite = FALSE)

icm_to_ppm(position)

motif_range(motif, use.freq = 1, allow.nonfinite = FALSE)

motif_score(motif, threshold = c(0, 1), use.freq = 1,
  allow.nonfinite = FALSE, threshold.type = c("total", "fromzero"))

log_string_pval(pval)

pcm_to_ppm(position, pseudocount = 0)

position_icscore(position, bkg = numeric(), type = "PPM",
  pseudocount = 1, nsites = 100, relative_entropy = FALSE,
  schneider_correction = FALSE)

ppm_to_icm(position, bkg = numeric(), schneider_correction = FALSE,
  nsites = 100, relative_entropy = FALSE)

ppm_to_pcm(position, nsites = 100)

ppm_to_pwm(position, bkg = numeric(), pseudocount = 1, nsites = 100,
  smooth = TRUE)
```

```
prob_match(motif, match, allow.zero = TRUE)

prob_match_bkg(bkg, match)

pwm_to_ppm(position, bkg = numeric())

round_motif(motif, pct.tolerance = 0.05)

score_match(motif, match, allow.nonfinite = FALSE)

summarise_motifs(motifs, na.rm = TRUE)

ungap(motif, delete = FALSE)
```

Arguments

motif	Motif object to calculate scores from, or add/remove gap, or round.
gaploc	numeric Motif gap locations. The gap occurs immediately after every position value. If missing, uses round(ncol(motif) / 2).
mingap	numeric Minimum gap size. Must have one value for every location. If missing, set to 1.
maxgap	numeric Maximum gap size. Must have one value for every location. If missing, set to 5.
motifs	list A list of universalmotif motifs.
average	character(1) One of c("a.mean", "g.mean", "median", "fzt"). How to calculate the average motif information content.
x	numeric First column for comparison.
y	numeric Second column for comparison.
method	character(1) Column comparison metric. See compare_motifs() for details.
bkg1	numeric Vector of background probabilities for the first column. Only relevant if method = "ALLR".
bkg2	numeric Vector of background probabilities for the second column. Only relevant if method = "ALLR".
nsites1	numeric(1) Number of sites for the first column. Only relevant if method = "ALLR".
nsites2	numeric(1) Number of sites for the second column. Only relevant if method = "ALLR".
letter	character(1) Any DNA, RNA, or AA IUPAC letter. Ambiguity letters are accepted.
position	numeric A numeric vector representing the frequency or probability for each alphabet letter at a specific position.
alphabet	character(1) One of c('DNA', 'RNA').
type	character(1) One of c('PCM', 'PPM', 'PWM' 'ICM').
pseudocount	numeric(1) Used to prevent zeroes in motif matrix.
score	numeric(1) Logodds motif score.

allow.nonfinite
: logical(1) If FALSE, then apply a pseudocount if non-finite values are found in the PWM. Note that if the motif has a pseudocount greater than zero and the motif is not currently of type PWM, then this parameter has no effect as the pseudocount will be applied automatically when the motif is converted to a PWM internally. This value is set to FALSE by default in order to stay consistent with pre-version 1.8.0 behaviour. A message will be printed if a pseudocount is applied. To disable this, set options(pseudocount.warning=FALSE).

use.freq
: numeric(1) Use regular motif or the respective multifreq representation.

threshold
: numeric(1) Any number of numeric values between 0 and 1 representing score percentage.

threshold.type
: character For "total", a threshold of zero represents the minimum possible score. This means the range of scores that can be extracted is from the minimum to the maximum possible scores. For "fromzero", a threshold of zero is a score of zero. This means the range of scores is from zero to the maximum. The "total" threshold type can only be used if no non-finite values are present in the PWM.

pval
: character(1) String-formatted p-value.

bkg
: numeric Should be the same length as the alphabet length.

nsites
: numeric(1) Number of sites motif originated from.

relative_entropy
: logical(1) Calculate information content as relative entropy or Kullback-Leibler divergence.

schneider_correction
: logical(1) Apply sample size correction.

smooth
: logical(1) Apply pseudocount correction.

match
: character Sequence string to calculate score from.

allow.zero
: logical(1) If FALSE, apply a pseudocount if zero values are found in the background frequencies.

pct.tolerance
: numeric(1) or character(1) The minimum tolerated proportion each letter must represent per position in order not to be rounded off, either as a numeric value from 0 to 1 or a percentage written as a string from "0%" to "100%".

na.rm
: logical Remove columns where all values are NA.

delete
: logical(1) Clear gap information from motif. If FALSE, then it can be reactivated simply with add_gap(motif).

Value

For consensus_to_ppm() and consensus_to_ppmAA(): a numeric vector of length 4 and 20, respectively.

For get_consensus() and get_consensusAA(): a character vector of length 1.

For get_matches(): a character vector of motif matches. A score of Inf (for example, from an unattainable motif_pvalue() cutoff) returns character(0).

For motif_range(): a named numeric vector of motif scores.

For motif_score(): a named numeric vector of motif scores.

For log_string_pval(): a numeric vector of length 1.

For position_icscore(): a numeric vector of length 1.

For ppm_to_icm(), icm_to_ppm(), pcm_to_ppm(), ppm_to_pcm(), ppm_to_pwm(), and pwm_to_ppm(): a numeric vector with length equal to input numeric vector.

For prob_match(): a numeric vector of probabilities.

For round_motif(): the input motif, rounded.

For score_match(): a numeric vector with the match motif score.

For summarise_motifs(): a data.frame with columns representing the universalmotif slots.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

create_motif()

Examples

```
data(examplemotif)
examplemotif0 <- examplemotif
examplemotif0["pseudocount"] <- 0

################################################################################
## add_gap
## Add gap information to a motif.
m <- create_motif()
# Add a gap size 5-8 between positions 4 and 5:
m <- add_gap(m, gaploc = 4, mingap = 5, maxgap = 8)

################################################################################
## average_ic
## Calculate the average information content for a list of motifs.
m <- create_motif()
average_ic(m, "fzt")

################################################################################
## compare_columns
## Compare two numeric vectors using the metrics from compare_motifs()
compare_columns(c(0.5, 0.1, 0.1, 0.2), c(0.7, 0.1, 0.1, 0.1), "PCC")

################################################################################
## consensus_to_ppm
## Do the opposite of get_consensus. Note that loss of information is
## inevitable. Generates a sequence matrix.
sapply(c("A", "G", "T", "B"), consensus_to_ppm)

################################################################################
## consensus_to_ppmAA
## Do the opposite of get_consensusAA and generate a motif matrix.
sapply(c("V", "A", "L"), consensus_to_ppmAA)

################################################################################
## get_consensus
## Get a consensus string from a DNA/RNA motif.
m <- create_motif()["motif"]
apply(m, 2, get_consensus)
```

```
#########################################################################
## get_consensusAA
## Get a consensus string from an amino acid motif. Unless each position
## is clearly dominated by a single amino acid, the resulting string will
## likely be useless.
m <- create_motif(alphabet = "AA")["motif"]
apply(m, 2, get_consensusAA, type = "PPM")

#########################################################################
## get_match
## Get all possible motif matches above input score
get_matches(examplemotif, 0)
get_matches(examplemotif0, 0, allow.nonfinite = TRUE)

#########################################################################
## get_scores
## Get all possible scores for a motif
length(get_scores(examplemotif))
get_scores(examplemotif)
get_scores(examplemotif0, allow.nonfinite = TRUE)

#########################################################################
## icm_to_ppm
## Do the opposite of ppm_to_icm.
m <- create_motif(type = "ICM")["motif"]
apply(m, 2, icm_to_ppm)

#########################################################################
## motif_range
## Calculate the range of possible logodds scores for a motif
motif_range(examplemotif)
motif_range(examplemotif, allow.nonfinite = TRUE)

#########################################################################
## motif_score
## Calculate motif score from different thresholds
m <- normalize(examplemotif)
motif_score(m, c(0, 0.8, 1))
motif_score(examplemotif0, c(0, 0.8, 1), allow.nonfinite = TRUE,
   threshold.type = "fromzero")

#########################################################################
## log_string_pval
## Get the log of a string-formatted p-value
log_string_pval("1e-200")

#########################################################################
## pcm_to_ppm
## Go from a count type motif to a probability type motif.
m <- create_motif(type = "PCM", nsites = 50)["motif"]
apply(m, 2, pcm_to_ppm, pseudocount = 1)

#########################################################################
## position_icscore
## Similar to ppm_to_icm, except this calculates the position sum.
m <- create_motif()["motif"]
```

```
apply(m, 2, position_icscore, type = "PPM", bkg = rep(0.25, 4))

########################################################################
## ppm_to_icm
## Convert one column from a probability type motif to an information
## content type motif.
m <- create_motif(nsites = 100, pseudocount = 0.8)["motif"]
apply(m, 2, ppm_to_icm, nsites = 100, bkg = rep(0.25, 4))

########################################################################
## ppm_to_pcm
## Do the opposite of pcm_to_ppm.
m <- create_motif()["motif"]
apply(m, 2, ppm_to_pcm, nsites = 50)

########################################################################
## ppm_to_pwm
## Go from a probability type motif to a weight type motif.
m <- create_motif()["motif"]
apply(m, 2, ppm_to_pwm, nsites = 100, bkg = rep(0.25, 4))

########################################################################
## prob_match, prob_match_bkg
## Calculate probability of a particular match based on background
## frequencies
prob_match(examplemotif, "TATATAT")
## Since this motif has a uniform background, the probability of
## finding any motif hit within the sequence is equal
prob_match(examplemotif, "TATATAG")
m <- examplemotif
m["bkg"] <- c(0.3, 0.2, 0.2, 0.3)
prob_match(m, "TATATAT")
## The prob_match_bkg alternative allows you to simply pass along the
## background frequencies
prob_match_bkg(c(A=0.3, C=0.2, G=0.2, T=0.3), c("TATATAT", "TATATAG"))

########################################################################
## pwm_to_ppm
## Do the opposite of ppm_to_pwm.
m <- create_motif(type = "PWM")["motif"]
apply(m, 2, pwm_to_ppm, bkg = rep(0.25, 4))

########################################################################
## Note that not all type conversions can be done directly; for those
## type conversions which are unavailable, universalmotif just chains
## together others (i.e. from PCM -> ICM => pcm_to_ppm -> ppm_to_icm)

########################################################################
## round_motif
## Round down letter scores to 0
m <- create_motif()
## Remove letters from positions which are less than 5% of the total
## position:
round_motif(m, pct.tolerance = 0.05)

########################################################################
## score_match
```

```
## Calculate score of a particular match
score_match(examplemotif, "TATATAT")
score_match(examplemotif, "TATATAG")
score_match(examplemotif0, "TATATAT", allow.nonfinite = TRUE)
score_match(examplemotif0, "TATATAG", allow.nonfinite = TRUE)

######################################################################
## summarise_motifs
## Create a data.frame of information based on a list of motifs.
m1 <- create_motif()
m2 <- create_motif()
m3 <- create_motif()
summarise_motifs(list(m1, m2, m3))

######################################################################
## ungap
## Unset motif's gap status. Does not delete actual gap data unless
## delete = TRUE.
m <- create_motif()
m <- add_gap(m, 3, 2, 4)
m <- ungap(m)
# Restore gap data:
m <- add_gap(m)
```

---

utils-sequence *Sequence-related utility functions.*

Description

Sequence-related utility functions.

Usage

```
calc_complexity(string, complexity.method = c("WoottonFederhen",
  "WoottonFederhenFast", "Trifonov", "TrifonovFast", "DUST"), alph = NULL,
  trifonov.max.word.size = 7)

calc_windows(n, window = 1, overlap = 0, return.incomp = TRUE)

count_klets(string, k = 1, alph)

get_klets(lets, k = 1)

mask_ranges(seqs, ranges, letter = "-")

mask_seqs(seqs, pattern, RC = FALSE, letter = "-")

meme_alph(core, file = stdout(), complements = NULL, ambiguity = NULL,
  like = NULL, alph.name = NULL, letter.names = NULL, colours = NULL)

shuffle_string(string, k = 1, method = c("euler", "linear", "markov"),
  rng.seed = sample.int(10000, 1))
```

```
slide_fun(string, FUN, FUN.VALUE, window = 1, overlap = 0,
  return.incomp = TRUE)

window_string(string, window = 1, overlap = 0, return.incomp = TRUE,
  nthreads = 1)
```

Arguments

string	character(1) A character vector containing a single string, with the exception of calc_complexity() where string can be a length greater than one.
complexity.method	character(1) Complexity algorithm. See sequence_complexity().
alph	character(1) A single character string with the desired sequence alphabet. If missing, it is "ACGT" when the strings hold only those letters and N, "ACGU" likewise for RNA, and otherwise every letter found in the strings, so that all strings are scored against the same alphabet.
trifonov.max.word.size	integer(1) Maximum word size for use in the Trifonov complexity methods. See sequence_complexity().
n	integer(1) Total size from which to calculate sliding windows.
window	integer(1) Window size to slide along.
overlap	integer(1) Overlap size between windows.
return.incomp	logical(1) Whether to return the last window if it is smaller then the requested window size.
k	integer(1) K-let size.
lets	character A character vector where each element will be considered a single unit.
seqs	XStringSet Sequences to mask. Cannot be BStringSet.
ranges	GRanges The ranges to mask. Must be a GRanges object from the GenomicRanges package, on sequences named in seqs.
letter	character(1) Character to use for masking.
pattern	character(1) Pattern to mask.
RC	logical(1) Whether to mask the reverse complement of the pattern.
core	character(1) Core alphabet symbols. If complements are also provided, then only half of the letters should be provided to this argument.
file	Output file.
complements	character(1), NULL Complementary letters to the core symbols.
ambiguity	character(1), NULL A named vector providing ambiguity codes for the custom alphabet.
like	character(1), NULL How to classify the custom alphabet. If not NULL, then one of c("DNA", "RNA", "PROTEIN").
alph.name	character(1), NULL Custom alphabet name.
letter.names	character, NULL Named vector of core symbol names.
colours	character, NULL Named vector of core symbol colours. MEME requires hex colours.

method	character(1) Shuffling method. One of c("euler", "linear", "markov"). See shuffle_sequences().
rng.seed	numeric(1) Set random number generator seed. Since shuffling in shuffle_sequences() can occur simultaneously in multiple threads using C++, it cannot communicate with the regular R random number generator state and thus requires an independent seed. Since shuffle_string() uses the same underlying code as shuffle_sequences(), it also requires a separate seed even if it is run in serial.
FUN	closure The function to apply per window. (See ?vapply.)
FUN.VALUE	The expected return type for FUN. (See ?vapply.)
nthreads	integer(1) Number of threads to use. Zero uses all available threads.

Value

For calc_complexity(): A vector of numeric values.

For calc_windows(): A data.frame with columns start and stop.

For count_klets(): A data.frame with columns lets and counts.

For get_klets(): A character vector of k-lets.

For mask_ranges(): The masked XStringSet object.

For mask_seqs(): The masked XStringSet object.

For meme_alph(): NULL, invisibly.

For shuffle_string(): A single character string.

For slide_fun(): A vector with type FUN.VALUE.

For window_string(): A character vector.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

create_sequences(), get_bkg(), sequence_complexity(), shuffle_sequences()

Examples

```
########################################################################
## calc_complexity
## Calculate complexity for abitrary strings
calc_complexity("GTGCCCCGCGGGAACCCCGC", c = "WoottonFederhen")
calc_complexity("GTGCCCCGCGGGAACCCCGC", c = "WoottonFederhenFast")
calc_complexity("GTGCCCCGCGGGAACCCCGC", c = "Trifonov")
calc_complexity("GTGCCCCGCGGGAACCCCGC", c = "TrifonovFast")
calc_complexity("GTGCCCCGCGGGAACCCCGC", c = "DUST")

########################################################################
## calc_windows
## Calculate window coordinates for any value 'n'.
calc_windows(100, 10, 5)

########################################################################
## count_klets
## Count k-lets for any string of characters
```

```
count_klets("GCAAATGTACGCAGGGCCGA", k = 2)
## The default 'k' value (1) counts individual letters
count_klets("GCAAATGTACGCAGGGCCGA")

#########################################################################
## get_klets
## Generate all possible k-lets for a set of characters
get_klets(c("A", "C", "G", "T"), 3)
## Note that each element in 'lets' is considered a single unit;
## see:
get_klets(c("AA", "B"), k = 2)

#########################################################################
## mask_ranges
## Mask arbitrary ranges
if (requireNamespace("GenomicRanges", quietly = TRUE)) {
ranges <- GenomicRanges::GRanges("A", IRanges::IRanges(1, 5))
seq <- Biostrings::DNAStringSet(c(A = "ATGACTGATTACTTATA"))
mask_ranges(seq, ranges, "-")
}

#########################################################################
## mask_seqs
## Mask repetitive seqeuences
data(ArabidopsisPromoters)
mask_seqs(ArabidopsisPromoters, "AAAAAA")

#########################################################################
## meme_alph
## Create MEME custom alphabet definition files
meme_alph("ACm", complements = "TGM", alph.name = "MethDNA",
  letter.names = c(A = "Adenine", C = "Cytosine", G = "Guanine",
    T = "Thymine", m = "Methylcytosine", M = "mC:Guanine"),
  like = "DNA", ambiguity = c(N = "ACGTmM"))

#########################################################################
## shuffle_string
## Shuffle any string of characters
shuffle_string("ASDADASDASDASD", k = 1)

#########################################################################
## slide_fun
## Apply a function to a character vector along sliding windows
FUN <- function(x) grepl("[GC]", x)
data.frame(
  Window = window_string("ATGCATCTATGCA", 2, 1),
  HasGC = slide_fun("ATGCATCTATGCA", FUN, logical(1), 2, 1)
)

#########################################################################
## window_string
## Get sliding windows for a string of characters
window_string("ABCDEFGHIJ", 2, 1)
```

view_logo *Plot logos from numeric matrices.*

Description

This function provides the plotting capabilities of view_motifs() without requiring universalmotif-class objects. Instead, it takes a numeric matrix with row names as input. Additionally, columns can be of any height and letters can be a mix of different character lengths.

Usage

```
view_logo(x, fontDF = NULL, fill = "black", colour.scheme = NULL,
  min.height = 0.01, x.spacer = 0.04, y.spacer = 0.01,
  sort.positions = FALSE, sort.positions.decreasing = TRUE,
  fit.to.height = NULL, flip.neg = FALSE)
```

Arguments

x	A numeric matrix with row names. The row names can be a mix of different character lengths.
fontDF	data.frame or DataFrame Polygon data for letters used for plotting, as generated by the createPolygons() function from the gglogo package. See the fontDFroboto data object (which is used by default when fontDF = NULL). See Examples for how to generate your own font set. Expected columns: x, y, order, group; additional columns will be ignored.
fill	character A single colour to fill all letters with. Ignored if colour.scheme is provided.
colour.scheme	character A named character vector of colour names. Provide colours for individual letters, even if the row names are made up of multiple characters.
min.height	numeric(1) Minimum height for a letter to be plotted. The number is taken as the fraction of the total height of the plot. The default value is to not show letters which take up 1% or less of the vertical space. For smaller figures it is recommended to increase this value, and vice versa for larger figures.
x.spacer	numeric(1) Add horizontal spacing between letters. The number is taken as the fraction of the width of an individual position. Increasing this value is recommended for letters made up of multiple characters.
y.spacer	numeric(1) Add vertical spacing between letters. The number is taken as the fraction nof the total height of the plot.
sort.positions	logical(1) Sort letters vertically per position by height.
sort.positions.decreasing	logical(1) Sort in decreasing or increasing order based on letter height.
fit.to.height	numeric(1) Normalize the per position height to this value. If NULL, no normalization is applied. Note that this parameter is ignored if use.type = c("PWM", "ICM").
flip.neg	logical(1) Flip letters with negative heights.

Value

A ggplot object. If you wish to plot the data yourself from polygon paths, access them using $data on the output object. The theme theme_void() is applied to the object; apply your own theme or adjust specific plot parameters with theme() to change this.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

view_motifs()

Examples

```
## Feel free to mix and match row name character lengths and column sums.
data(examplemotif)
toplot <- examplemotif["motif"]
toplot[4] <- 2
toplot[20] <- -0.5
rownames(toplot)[1] <- "AA"
view_logo(toplot)
```

---

view_motifs *Plot motif logos.*

Description

Show sequence logo. If given a list of more than one motif, then the motifs are aligned with the first in the list. For a leaner alternative in line with compare_motifs_lite(), see view_motifs_lite().

Usage

```
view_motifs(motifs, use.type = "ICM", method = "ALLR", tryRC = TRUE,
  min.overlap = 6, min.mean.ic = 0.25, relative_entropy = FALSE,
  normalise.scores = FALSE, min.position.ic = 0, score.strat = "sum",
  return.raw = FALSE, dedup.names = TRUE, show.positions = TRUE,
  show.positions.once = TRUE, sort.by = c("none", "ic", "similarity"),
  show.names = TRUE, names.pos = c("top", "right"), use.freq = 1,
  colour.scheme = NULL, fontDF = NULL, min.height = 0.01, x.spacer = if
  (use.freq == 1) 0.04 else 0.1, y.spacer = 0.01,
  sort.positions = !use.type %in% c("PCM", "PPM"),
  sort.positions.decreasing = TRUE, text.size = 16, fit.to.height = if
  (use.type == "PPM") 1 else NULL, RC.text = " [RC]", flip.neg = FALSE,
  ...)
```

Arguments

motifs	See convert_motifs() for acceptable motif formats.
use.type	character(1) One of c('PCM', 'PPM', 'PWM', 'ICM').
method	character(1) One of PCC, EUCL, SW, KL, ALLR, BHAT, HELL, SEUCL, MAN, ALLR_LL, WEUCL, WPCC. See details.
tryRC	logical(1) Try the reverse complement of the motifs as well, report the best score.
min.overlap	numeric(1) Minimum overlap required when aligning the motifs. Setting this to a number higher then the width of the motifs will not allow any overhangs. Can also be a number between 0 and 1, representing the minimum fraction that the motifs must overlap.
min.mean.ic	numeric(1) Minimum mean information content between the two motifs for an alignment to be scored. This helps prevent scoring alignments between low information content regions of two motifs. Note that this can result in some comparisons failing if no alignment passes the mean IC threshold. Use average_ic() to filter out low IC motifs to get around this if you want to avoid getting NAs in your output.
relative_entropy	logical(1) Change the ICM calculation affecting min.position.ic and min.mean.ic. See convert_type().
normalise.scores	logical(1) Favour alignments which leave fewer unaligned positions, as well as alignments between motifs of similar length. Similarity scores are multiplied by the ratio of aligned positions to the total number of positions in the larger motif, and the inverse for distance scores.
min.position.ic	numeric(1) Minimum information content required between individual alignment positions for it to be counted in the final alignment score. It is recommended to use this together with normalise.scores = TRUE, as this will help punish scores resulting from only a fraction of an alignment.
score.strat	character(1) How to handle column scores calculated from motif alignments. "sum": add up all scores. "a.mean": take the arithmetic mean. "g.mean": take the geometric mean. "median": take the median. "wa.mean", "wg.mean": weighted arithmetic/geometric mean. "fzt": Fisher Z-transform. Weights are the total information content shared between aligned columns.
return.raw	logical(1) Instead of returning a plot, return the aligned named matrices used to generate the plot. This can be useful if you wish to use view_motifs() alignment capabilities for custom plotting uses. Alignment is performed by adding empty columns to the left or right of motifs to generate matrices of equal length.
dedup.names	logical(1) Plotting motifs with duplicated names is not allowed. Setting this to TRUE allows the names to be modified for plotting.
show.positions	logical(1) Show x-axis position tick labels.
show.positions.once	logical(1) When plotting multiple motifs, show x-axis position tick labels only once. If FALSE, then x-axis tick labels are specific to each motif.
sort.by	character(1). Display order for multi-motif plots. One of "none" (default), "ic", or "similarity". Regardless of this setting, alignment is always performed with the highest-IC motif as the anchor; sort.by controls only the order in which the aligned motifs are displayed.

- `"none"`: the input order is preserved for display, while alignment still uses the highest-IC motif as the anchor.
- `"ic"`: motifs are shown in descending information content, so the highest-IC motif (the alignment anchor) appears first.
- `"similarity"`: after alignment, motifs are reordered by hierarchical clustering on Pearson correlation between aligned columns, so visually similar motifs are adjacent. Useful when plotting a large heterogeneous set. Ignored when only one motif is provided.

`show.names` `logical(1)` Add motif names when plotting multiple motifs.

`names.pos` `character(1)` Motif name locations. Either above (`top`) or to the right (`right`) of the logos.

`use.freq` `numeric(1)` Plot higher order motifs from the `multifreq` slot.

`colour.scheme` `character` A named character vector of colour names. Default colours are provided for DNA, RNA, and AA motifs if left `NULL`.

`fontDF` `data.frame` or `DataFrame` Polygon data for letters used for plotting, as generated by the `createPolygons()` function from the `gglogo` package. See the `fontDFroboto` data object (which is used by default when `fontDF = NULL`). See `Examples` for how to generate your own font set. Expected columns: `x`, `y`, `order`, `group`; additional columns will be ignored.

`min.height` `numeric(1)` Minimum height for a letter to be plotted. The number is taken as the fraction of the total height of the plot. The default value is to not show letters which take up 1% or less of the vertical space. For smaller figures it is recommended to increase this value, and vice versa for larger figures.

`x.spacer` `numeric(1)` Add horizontal spacing between letters. The number is taken as the fraction of the width of an individual position. Increasing this value is recommended for plotting multifreq motifs.

`y.spacer` `numeric(1)` Add vertical spacing between letters. The number is taken as the fraction nof the total height of the plot.

`sort.positions` `logical(1)` Sort letters vertically per position by height.

`sort.positions.decreasing` `logical(1)` Sort in decreasing or increasing order based on letter height.

`text.size` `numeric(1)` Text size for labels.

`fit.to.height` `numeric(1)` Normalize the per position height to this value. If `NULL`, no normalization is applied. Note that this parameter is ignored if `use.type = c("PWM", "ICM")`.

`RC.text` `character(1)` The text to display alongside the name of motifs shown as their reverse complement.

`flip.neg` `logical(1)` Flip letters with negative heights.

`...` Unused. Was previously in place to allow extra args to be given to `ggseqlogo::ggseqlogo`, however `universalmotif` now implements its own motif plotting code directly with `ggplot2`.

Details

See `compare_motifs()` for more info on comparison parameters.

See `view_logo()` to plot from a numeric matrix with arbitrary values instead of a motif object.

Note: `score.strat = "a.mean"` is NOT recommended, as `view_motifs()` will not discriminate between two alignments with equal mean scores, even if one alignment is longer than the other.

Note: if you want to plot the motifs yourself, you can set `return.raw=TRUE` to get the numeric motif matrices and calculate the polygon paths on your own or access the polygon path data directly from the final `ggplot` object using `$data`.

Value

A `ggplot` object. If `return.raw = TRUE`, a list of matrices.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

`compare_motifs()`, `add_multifreq()`, `view_logo()`, `view_motifs_lite()`

Examples

```
## Plotting multifreq motifs:
data(examplemotif2)
view_motifs(examplemotif2, use.freq = 2)

## Generate your own letter set:
## Not run:

library(gglogo)  # install from CRAN first if needed
fontDFtimes <- createPolygons(LETTERS, "Times", 800, scale = TRUE)
view_motifs(examplemotif2, fontDF = fontDFtimes)

## Note: setting `scale = TRUE` is necessary to properly align letters
## vertically, but this has the effect of horizontally stretching out
## letters which shouldn't be stretched (such as "I"). If you need to plot
## letters which have been badly horizontally scaled, you can fix them
## manually as demonstrated here:

# Retrieve the x-coordinates for the desired letter:
tofix <- fontDFtimes$x[fontDFtimes$group == "I"]
# Scale the letter x-coordinates:
tofix <- tofix * 0.35
# Remember to center the letter around 0.5 again:
tofix <- tofix + (1 - max(tofix)) / 2
# Apply the fix:
fontDFtimes$x[fontDFtimes$group == "I"] <- tofix
view_motifs(create_motif("AIG", alphabet = "AA"), fontDF = fontDFtimes)


## End(Not run)
```

`view_motifs_lite` *Plot motif logos using* `compare_motifs_lite()` *alignment (v2).*

Description

`view_motifs_lite()` is the leaner counterpart of `view_motifs()`: it aligns the input motifs using the same Pearson-correlation backend used by `compare_motifs_lite()` and `merge_motifs_lite()`. The rendering (logo polygons, `facet_wrap()` layout, `sort.by` permutation, `names.pos` placement) is identical to `view_motifs()`.

Usage

```
view_motifs_lite(motifs, use.type = "ICM", tryRC = TRUE, min.overlap = 6,
  relative_entropy = FALSE, nthreads = 1, return.raw = FALSE,
  dedup.names = TRUE, show.positions = TRUE, show.positions.once = TRUE,
  sort.by = c("none", "ic", "similarity"), show.names = TRUE,
  names.pos = c("top", "right"), use.freq = 1, colour.scheme = NULL,
  fontDF = NULL, min.height = 0.01, x.spacer = if (use.freq == 1) 0.04
  else 0.1, y.spacer = 0.01, sort.positions = !use.type %in% c("PCM",
  "PPM"), sort.positions.decreasing = TRUE, text.size = 16,
  fit.to.height = if (use.type == "PPM") 1 else NULL, RC.text = " [RC]",
  flip.neg = FALSE)
```

Arguments

`motifs`	See `convert_motifs()` for accepted motif formats. DNA or RNA only.
`use.type`	`character(1)` One of `c('PCM', 'PPM', 'PWM', 'ICM')`. Controls the matrix type used for display.
`tryRC`	`logical(1)` Also test reverse-complement alignments. Default `TRUE`.
`min.overlap`	`integer(1)` Minimum overlap (in columns) for an alignment to be accepted. Default `6`.
`relative_entropy`	`logical(1)` If `TRUE`, compute information content using the motif's background frequencies (relative-entropy ICM). Used only for display when `use.type = "ICM"`.
`nthreads`	`numeric(1)` Number of threads for comparison. `nthreads = 0` uses all available threads.
`return.raw`	`logical(1)` Instead of returning a plot, return the aligned named matrices used to generate the plot.
`dedup.names`	`logical(1)` Allow duplicate motif names to be silently disambiguated for plotting.
`show.positions`	`logical(1)` Show x-axis position tick labels.
`show.positions.once`	`logical(1)` When plotting multiple motifs, show x-axis position tick labels only once.
`sort.by`	`character(1)`. Display order for multi-motif plots. One of `"none"` (default), `"ic"`, or `"similarity"`. Alignment is always performed with the highest-IC motif as the anchor; `sort.by` controls only the order in which the aligned motifs are shown.
`show.names`	`logical(1)` Add motif names when plotting multiple motifs.
`names.pos`	`character(1)` Motif name locations, either `"top"` or `"right"`.
`use.freq`	`numeric(1)` Plot higher-order motifs from the `multifreq` slot.

colour.scheme	character Named character vector of colour names. Default colours are provided for DNA, RNA, and AA motifs if NULL.
fontDF	data.frame or DataFrame Polygon data for letters.
min.height	numeric(1) Minimum letter height to show, as a fraction of total plot height.
x.spacer	numeric(1) Horizontal spacing between letters.
y.spacer	numeric(1) Vertical spacing between letters.
sort.positions	logical(1) Sort letters vertically per position by height.
sort.positions.decreasing	logical(1) Sort direction.
text.size	numeric(1) Text size for labels.
fit.to.height	numeric(1) Normalise per-position height.
RC.text	character(1) Text appended to motif names shown as their reverse complement.
flip.neg	logical(1) Flip letters with negative heights.

Details

DNA and RNA only. For amino-acid or custom-alphabet motifs use view_motifs().

The alignment step uses the same backend as merge_motifs_lite() and merge_similar_lite(). The highest-information-content input motif is chosen as the anchor, every other motif is aligned against it once, motifs whose best alignment lands on the reverse complement strand are reverse-complemented before display, and all motifs are placed in a shared column frame with zero-padding on either side as needed. The padded matrices then flow through the same logo-rendering pipeline used by view_motifs().

See view_motifs() for the original version, which supports Euclidean / similarity metrics and arbitrary alphabets.

Value

A ggplot object. If return.raw = TRUE, a list of matrices.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

view_motifs(), view_logo(), compare_motifs_lite(), merge_motifs_lite(), merge_similar_lite()

Other lite motif functions: compare_motifs_lite(), enrich_motifs_lite(), merge_motifs_lite(), merge_similar_lite(), motif_tree_lite(), scan_sequences_lite()

Examples

```
m1 <- create_motif("TTGACATA", name = "a")
m2 <- create_motif("CTTGACAT", name = "b")
m3 <- create_motif("TGACATAT", name = "c")
view_motifs_lite(list(m1, m2, m3))
view_motifs_lite(list(m1, m2, m3), sort.by = "similarity",
             names.pos = "right")
```

write_homer *Export motifs in HOMER format.*

Description

Convert DNA motifs to HOMER format and write to file. See http://homer.ucsd.edu/homer/motif/.

Usage

```
write_homer(motifs, file, logodds_threshold = NULL, overwrite = FALSE,
  append = FALSE, threshold = 0.8, threshold.type = c("logodds",
  "logodds.abs", "pvalue"))
```

Arguments

motifs	See convert_motifs() for acceptable formats.
file	character(1) File name.
logodds_threshold	Deprecated. If set, read_homer() will behave like pre-version 1.12.0 of the universalmotif package for backwards compatibility (though a warning will be printed).
overwrite	logical(1) Overwrite existing file.
append	logical(1) Add to an existing file.
threshold	numeric(1) Stringency required for HOMER to match a motif. See scan_sequences() for how to use this argument. Can be a single value to be recycled for all motifs, or a vector of equal length to the number of motifs.
threshold.type	character(1) How the threshold value should be used to obtain the final threshold value in the written motif. See scan_sequences() for how to use this. Thresholds are written on HOMER's own score scale: the natural log odds of the probabilities as written to the file, floored at 0.001, against a uniform background. P-value thresholds are raised by up to 0.001 per position so that every site HOMER accepts is within the P-value.

Value

NULL, invisibly.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Heinz S, Benner C, Spann N, Bertolino E, Lin YC, Laslo P, Cheng JX, Murre C, Singh H, Glass CK (2010). "Simple combinations of lineage-determining transcription factors prime cis-regulatory elements required for macrophage and B cell identities." *Molecular Cell*, **38**, 576-589.

See Also

read_homer()

Other write_motifs: write_jaspar(), write_matrix(), write_meme(), write_motifs(), write_transfac()

Examples

```
motif <- create_motif()
write_homer(motif, tempfile())
```

---

write_jaspar *Export motifs in JASPAR format.*

Description

Convert motifs to JASPAR format and write to file. See http://jaspar.genereg.net/.

Usage

```
write_jaspar(motifs, file, overwrite = FALSE, append = FALSE)
```

Arguments

motifs	See convert_motifs() for acceptable formats.
file	character(1) File name.
overwrite	logical(1) Overwrite existing file.
append	logical(1) Add to an existing file.

Value

NULL, invisibly.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Khan A, Fornes O, Stigliani A, Gheorghe M, Castro-Mondragon JA, van der Lee R, Bessy A, Cheneby J, Kulkarni SR, Tan G, Baranasic D, Arenillas DJ, Sandelin A, Vandepoele K, Lenhard B, Ballester B, Wasserman WW, Parcy F, Mathelier A (2018). "JASPAR 2018: update of the open-access database of transcription factor binding profiles and its web framework." *Nucleic Acids Research*, **46**, D260-D266.

See Also

read_jaspar()

Other write_motifs: write_homer(), write_matrix(), write_meme(), write_motifs(), write_transfac()

Examples

```
transfac <- read_transfac(system.file("extdata", "transfac.txt",
                                      package = "universalmotif"))
write_jaspar(transfac, tempfile())
```

`write_matrix` *Export motifs as raw matrices.*

Description

Write motifs as simple matrices with optional headers to file.

Usage

```
write_matrix(motifs, file, positions = "columns", rownames = FALSE, type,
  sep = "", headers = TRUE, overwrite = FALSE, append = FALSE,
  digits = 6)
```

Arguments

`motifs`	See `convert_motifs()` for acceptable formats.
`file`	`character(1)` File name.
`positions`	`character(1)` One of `c('columns', 'rows')`. Partial matching allowed.
`rownames`	`logical(1)` Include alphabet letters as rownames.
`type`	`character(1)` One of `c('PCM', 'PPM', 'PWM', 'ICM', 'CWM')`. If missing, the motif is written with whatever type it is currently stored as, so a CWM motif (signed real values, no shared column sums) round-trips through `write_matrix()` / `read_matrix(type = "CWM")` verbatim. Note that `type = "CWM"` here only works with CWM motifs; you cannot ask `write_matrix()` to convert a PPM (or any other type) into a CWM (the underlying `convert_type()` step refuses any `*` -> `CWM` direction).
`sep`	`character(1)` Indicates how to separate individual motifs. Set as `NULL` to have no seperating lines between motifs (the default is to use a blank line).
`headers`	`logical(1)`, `character(1)` Indicating if and how to write names.
`overwrite`	`logical(1)` Overwrite existing file.
`append`	`logical(1)` Add to an existing file.
`digits`	`numeric(1)` Number of digits to use for motif positions.

Value

`NULL`, invisibly.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

read_matrix()

Other write_motifs: write_homer(), write_jaspar(), write_meme(), write_motifs(), write_transfac()

Examples

```
motif <- create_motif()
write_matrix(motif, tempfile(), headers = ">")

## CWM round-trip: write a CWM with its raw signed values, then
## read it back with type = "CWM" preserved.
cwm.mat <- matrix(c( 0.1, -0.2,  0.8, -0.1,
                    -0.3,  0.9, -0.2,  0.1,
                     0.7, -0.1, -0.1, -0.4,
                    -0.2, -0.1, -0.1,  0.9),
                  nrow = 4, byrow = FALSE,
                  dimnames = list(c("A","C","G","T"), NULL))
cwm <- create_motif(cwm.mat, type = "CWM", name = "modisco_demo")
f <- tempfile()
write_matrix(cwm, f, headers = ">", rownames = TRUE)
re <- read_matrix(f, headers = ">", rownames = TRUE, type = "CWM")
re["type"]
```

---

write_meme *Export motifs in MEME format.*

Description

Convert motifs to minimal MEME format and write to file. See http://meme-suite.org/doc/meme-format.html.

Usage

```
write_meme(motifs, file, version = 5, bkg, strand, overwrite = FALSE,
  append = FALSE, CWM = FALSE)
```

Arguments

motifs	See convert_motifs() for acceptable formats.
file	character(1) File name.
version	numeric(1) MEME version.
bkg	numeric Background letter frequencies. If missing, will use background frequencies from motif objects (if they are identical); else background frequencies will be set to freq = 1/length(alphabet)
strand	character If missing, will use strand from motif objects (if identical); otherwise will default to "+ -"
overwrite	logical(1) Overwrite existing file.
append	logical(1) Add to an existing file. Motifs will be written in minimal format, so it is recommended to only use this if the existing file is also a minimal MEME format file.

CWM — logical(1) If TRUE, write Contribution Weight Matrix (CWM) motifs verbatim into the letter-probability matrix: block: signed real values, no normalisation (matching what TF-MoDISco-lite emits). All input motifs must be of type = "CWM". If FALSE (default), CWM motifs are first converted to PPM via the |cwm| / sum route so the output stays strictly MEME-spec-compliant for downstream consumers (FIMO, AME, CentriMo). Read the file back with read_meme(CWM = TRUE) to recover the original CWM matrix bit-identical. See convert_type() for the full CWM semantics.

Value

NULL, invisibly.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Bailey TL, Boden M, Buske FA, Frith M, Grant CE, Clementi L, Ren J, Li WW, Noble WS (2009). "MEME SUITE: tools for motif discovery and searching." *Nucleic Acids Research*, **37**, W202-W208.

See Also

read_meme()

Other write_motifs: write_homer(), write_jaspar(), write_matrix(), write_motifs(), write_transfac()

Examples

```
transfac <- read_transfac(system.file("extdata", "transfac.txt",
                                      package = "universalmotif"))
write_meme(transfac, tempfile())

## CWM round-trip: write a CWM with its raw signed values, then
## recover it with read_meme(..., CWM = TRUE).
cwm.mat <- matrix(c( 0.1, -0.2,  0.8, -0.1,
                    -0.3,  0.9, -0.2,  0.1,
                     0.7, -0.1, -0.1, -0.4,
                    -0.2, -0.1, -0.1,  0.9),
                  nrow = 4, byrow = FALSE,
                  dimnames = list(c("A","C","G","T"), NULL))
cwm <- create_motif(cwm.mat, type = "CWM", name = "modisco_demo")
f <- tempfile(fileext = ".meme")
write_meme(cwm, f, CWM = TRUE, overwrite = TRUE)
re <- read_meme(f, CWM = TRUE)
re["type"]
```

write_motifs *Export motifs in universalmotif format.*

Description

Write motifs as universalmotif objects to file. For optimal storage of `universalmotif` class motifs, consider using `saveRDS()` and `readRDS()`. Currently the `universalmotif` format is YAML-based, but this is subject to change.

Usage

```
write_motifs(motifs, file, minimal = FALSE, multifreq = TRUE,
  progress = FALSE, overwrite = FALSE, append = FALSE, BP = FALSE)
```

Arguments

`motifs`	See `convert_motifs()` for acceptable formats.
`file`	`character(1)` File name.
`minimal`	`logical(1)` Only write essential motif information. Gap definitions are always retained. Use `saveRDS()` for lossless storage of all metadata and full matrix precision.
`multifreq`	`logical(1)` Write `multifreq` slot, if present.
`progress`	`logical(1)` Show progress.
`overwrite`	`logical(1)` Overwrite existing file.
`append`	`logical(1)` Add to an existing motif file. Package version in existing motif file must be greater than 1.2.0.
`BP`	`logical(1)` Allows for the use of **BiocParallel** within `write_motifs()`. See `BiocParallel::register()` to change the default backend.

Value

`NULL`, invisibly.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

See Also

Other write_motifs: `write_homer()`, `write_jaspar()`, `write_matrix()`, `write_meme()`, `write_transfac()`

Examples

```
motif <- create_motif("CARGTG", name = "MyMotif")
write_motifs(motif, tempfile())
```

`write_transfac` *Export motifs in TRANSFAC format.*

Description

Convert motifs to TRANSFAC format and write to file.

Usage

```
write_transfac(motifs, file, overwrite = FALSE, append = FALSE,
  name.tag = "ID", altname.tag = "NA")
```

Arguments

`motifs`	See `convert_motifs()` for acceptable formats.
`file`	`character(1)` File name.
`overwrite`	`logical(1)` Overwrite existing file.
`append`	`logical(1)` Add to an existing file.
`name.tag`	`character(1)` The tag to use when writing the motifs name slot.
`altname.tag`	`character(1)` The tag to use when writing the motifs altname slot. Note that no tag will be written if the slot is empty.

Details

If the family slot of a motif is not empty, then its contents will included using the HC tag. Similarly for the organism slot using the tag OS. The default name and alternate name tags are ID and NA, respectively, though these can be set manually.

Value

`NULL`, invisibly.

Author(s)

Benjamin Jean-Marie Tremblay, <benjamin.tremblay@uwaterloo.ca>

References

Wingender E, Dietze P, Karas H, Knuppel R (1996). "TRANSFAC: A Database on Transcription Factors and Their DNA Binding Sites." *Nucleic Acids Research*, **24**, 238-241.

See Also

`read_transfac()`

Other write_motifs: `write_homer()`, `write_jaspar()`, `write_matrix()`, `write_meme()`, `write_motifs()`

Examples

```
jaspar <- read_jaspar(system.file("extdata", "jaspar.txt",
                                  package = "universalmotif"))
write_transfac(jaspar, tempfile())
```

Index

∗ **datasets**
ArabidopsisMotif, 6
ArabidopsisPromoters, 6
examplemotif, 32
examplemotif2, 32
fontDFroboto, 34
geom_motif, 35
JASPAR2018_CORE_DBSCORES, 41
∗ **internal**
reexports, 95
∗ **lite motif functions**
compare_motifs_lite, 10
enrich_motifs_lite, 30
merge_motifs_lite, 50
merge_similar_lite, 53
motif_tree_lite, 73
scan_sequences_lite, 102
view_motifs_lite, 133
∗ **read_motifs**
read_cisbp, 86
read_homer, 87
read_jaspar, 88
read_matrix, 89
read_meme, 90
read_motifs, 92
read_transfac, 93
read_uniprobe, 94
∗ **write_motifs**
write_homer, 136
write_jaspar, 137
write_matrix, 138
write_meme, 139
write_motifs, 141
write_transfac, 142
[,universalmotif-method *(universalmotif-class)*, 114
[<-,universalmotif-method *(universalmotif-class)*, 114

AAStringSet, *22*
add_gap *(utils-motif)*, 119
add_multifreq, 4
add_multifreq(), *5*, *21*, *22*, *28*, *29*, *98*, *99*, *101*, *116*, *133*
annotate_logo *(geom_motif)*, 35
annotate_motif *(geom_motif)*, 35
ape::as.phylo(), *73*
ArabidopsisMotif, 6
ArabidopsisPromoters, *6*, 6
as.data.frame, *95*
as.data.frame *(reexports)*, 95
as.data.frame,universalmotif-method *(universalmotif-class)*, 114
average_ic *(utils-motif)*, 119
average_ic(), *7*, *42*, *48*, *52*, *71*, *131*

BiocParallel::register(), *92*, *141*
Biostrings::DNAStringSet(), *44*
Biostrings::getSeq(), *60*, *64*
Biostrings::injectHardMask(), *101*
Biostrings::mask(), *101*
Biostrings::matchPWM(), *100*, *101*
Biostrings::oligonucleotideFrequency(), *37*
Biostrings::readBStringSet(), *44*
BStringSet, *22*

calc_complexity *(utils-sequence)*, 125
calc_complexity(), *105*, *126*, *127*
calc_windows *(utils-sequence)*, 125
calc_windows(), *127*
cbind, *95*
cbind *(reexports)*, 95
cbind,universalmotif-method *(universalmotif-class)*, 114
colMeans, *95*
colMeans *(reexports)*, 95
colMeans,universalmotif-method *(universalmotif-class)*, 114
colnames, *95*
colnames *(reexports)*, 95
colnames,universalmotif-method *(universalmotif-class)*, 114
colSums, *95*
colSums *(reexports)*, 95
colSums,universalmotif-method *(universalmotif-class)*, 114
compare_columns *(utils-motif)*, 119

compare_motifs, 7
compare_motifs(), *8*, *10–12*, *41–43*, *48, 49*, *51, 52*, *70–73*, *120*, *132, 133*
compare_motifs_lite, 10, *32*, *51*, *54*, *74*, *103*, *135*
compare_motifs_lite(), *7*, *10*, *50, 51*, *53, 54*, *70*, *73–76*, *130*, *134, 135*
consensus_to_ppm *(*utils-motif*)*, 119
consensus_to_ppm(), *121*
consensus_to_ppmAA *(*utils-motif*)*, 119
consensus_to_ppmAA(), *121*
convert_motifs, 13
convert_motifs(), *4, 5*, *7*, *10, 11*, *15*, *17–19*, *27*, *30*, *33*, *35, 36*, *39*, *48–52*, *55*, *63*, *66*, *69, 70*, *73*, *75*, *79*, *81*, *85*, *98*, *102, 103*, *107*, *110*, *113*, *131*, *134*, *136–139*, *141, 142*
convert_motifs,AsIs-method *(*convert_motifs*)*, 13
convert_motifs,ICMatrix-method *(*convert_motifs*)*, 13
convert_motifs,list-method *(*convert_motifs*)*, 13
convert_motifs,matrix-method *(*convert_motifs*)*, 13
convert_motifs,Motif-method *(*convert_motifs*)*, 13
convert_motifs,MotifList-method *(*convert_motifs*)*, 13
convert_motifs,pcm-method *(*convert_motifs*)*, 13
convert_motifs,pfm-method *(*convert_motifs*)*, 13
convert_motifs,PFMatrix-method *(*convert_motifs*)*, 13
convert_motifs,PWM-method *(*convert_motifs*)*, 13
convert_motifs,pwm-method *(*convert_motifs*)*, 13
convert_motifs,PWMatrix-method *(*convert_motifs*)*, 13
convert_motifs,TFFMFirst-method *(*convert_motifs*)*, 13
convert_motifs,universalmotif-method *(*convert_motifs*)*, 13
convert_motifs,XMatrixList-method *(*convert_motifs*)*, 13
convert_type, 16
convert_type(), *7*, *21, 22*, *48*, *52*, *71*, *90*, *113, 114*, *131*, *138*, *140*
count_klets *(*utils-sequence*)*, 125
count_klets(), *105*, *127*
create_motif, 20
create_motif(), *25*, *42*, *59*, *98*, *110*, *113, 114*, *122*
create_motif,AAStringSet-method *(*create_motif*)*, 20
create_motif,BStringSet-method *(*create_motif*)*, 20
create_motif,character-method *(*create_motif*)*, 20
create_motif,DNAStringSet-method *(*create_motif*)*, 20
create_motif,matrix-method *(*create_motif*)*, 20
create_motif,missing-method *(*create_motif*)*, 20
create_motif,numeric-method *(*create_motif*)*, 20
create_motif,RNAStringSet-method *(*create_motif*)*, 20
create_sequences, 24
create_sequences(), *22*, *25*, *38*, *40*, *97, 98*, *109*, *127*

dedup_hits, 26
dedup_hits(), *55*, *102*
DNAString, *101*
DNAStringSet, *6*, *22*, *101*

enrich_motifs, 27
enrich_motifs(), *30–32*, *43*, *78, 79*, *101*, *109*
enrich_motifs_lite, *12*, 30, *51*, *54*, *74*, *103*, *135*
enrich_motifs_lite(), *27*, *29*, *39, 40*, *47*, *56, 57*, *59*, *78, 79*
examplemotif, 32
examplemotif2, 32

filter_motifs, 33
filter_motifs(), *15*, *33*
fontDFroboto, 34, *36*

GenomeInfoDb::seqlevelsStyle(), *65*
GenomicRanges::flank(), *63*
geom_logo *(*geom_motif*)*, 35
geom_logo(), *74*
geom_motif, 35
geom_motif(), *76*, *79*, *81*
GeomMotif *(*geom_motif*)*, 35
get_bkg, 37
get_bkg(), *37, 38*, *105*, *127*
get_consensus *(*utils-motif*)*, 119
get_consensus(), *121*
get_consensusAA *(*utils-motif*)*, 119

get_consensusAA(), *121*
get_klets *(*utils-sequence*)*, 125
get_klets(), *127*
get_matches *(*utils-motif*)*, 119
get_matches(), *68*, *121*
get_scores *(*utils-motif*)*, 119
get_scores(), *68*
ggplot2::aes(), *35*
ggplot2::annotation_custom(), *36*
ggplot2::ggplot(), *72*
ggplot2::ggsave(), *76*, *81*
ggplot2::layer(), *35*
ggtree::ggtree(), *70–74*

icm_to_ppm *(*utils-motif*)*, 119
icm_to_ppm(), *122*
implant_motifs, 39
implant_motifs(), *39*, *98*
initialize,universalmotif-method
*(*universalmotif-class*)*, 114

JASPAR2018_CORE_DBSCORES, 41

log_string_pval *(*utils-motif*)*, 119
log_string_pval(), *91*, *121*

make.unique(), *79*
make_DBscores, 41
make_DBscores(), *9, 10*, *41–43*
mask_complexity, 43
mask_ranges *(*utils-sequence*)*, 125
mask_ranges(), *44*, *105*, *127*
mask_seqs *(*utils-sequence*)*, 125
mask_seqs(), *44*, *101*, *105*, *127*
MaskedXString, *101*
match_bkg, 45
match_bkg(), *29*, *32*, *45*, *47*, *82*
meme_alph *(*utils-sequence*)*, 125
meme_alph(), *96*, *127*
merge_motifs, 48
merge_motifs(), *49–52*
merge_motifs_lite, *12*, *32*, 50, *54*, *74*, *103*, *135*
merge_motifs_lite(), *48, 49*, *53, 54*, *134, 135*
merge_similar, 51
merge_similar(), *49*, *51*, *53, 54*
merge_similar_lite, *12*, *32*, *51*, 53, *74*, *103*, *135*
merge_similar_lite(), *51, 52*, *74*, *135*
motif_coocc, 55
motif_coocc(), *55*, *77, 78*
motif_finder, 57
motif_finder(), *39, 40*, *47*
motif_peaks, 60
motif_peaks(), *39, 40*, *57*, *62*, *64, 65*, *83*
motif_proximity, 62
motif_proximity(), *84, 85*
motif_pvalue, 66
motif_pvalue(), *28, 29*, *32*, *66, 67*, *85, 86*, *99–101*, *103*, *121*
motif_range *(*utils-motif*)*, 119
motif_range(), *68*, *86*, *121*
motif_rc, 69
motif_rc(), *69*
motif_score *(*utils-motif*)*, 119
motif_score(), *68*, *121*
motif_tree, 70
motif_tree(), *8*, *10*, *70*, *73, 74*
motif_tree_lite, *12*, *32*, *51*, *54*, 73, *103*, *135*
motif_tree_lite(), *37*, *70*, *72*, *75, 76*
motifStack::motifStack(), *72*

ncol, *95*
ncol *(*reexports*)*, 95
ncol,universalmotif-method
*(*universalmotif-class*)*, 114
normalize, *95*
normalize *(*reexports*)*, 95
normalize,universalmotif-method
*(*universalmotif-class*)*, 114
nrow, *95*
nrow *(*reexports*)*, 95
nrow,universalmotif-method
*(*universalmotif-class*)*, 114

pcm_to_ppm *(*utils-motif*)*, 119
pcm_to_ppm(), *122*
plot_comparison, 75
plot_coocc, 77
plot_coocc(), *79*
plot_enrichment, 78
plot_enrichment(), *78*
plot_hits, 80
plot_match_bkg, 82
plot_match_bkg(), *47*
plot_motif_peaks, 83
plot_motif_peaks(), *61*
plot_motif_proximity, 84
plot_motif_proximity(), *64, 65*
plot_score_distribution, 85
position_icscore *(*utils-motif*)*, 119
position_icscore(), *121*
ppm_to_icm *(*utils-motif*)*, 119
ppm_to_icm(), *122*
ppm_to_pcm *(*utils-motif*)*, 119

ppm_to_pcm(), *122*
ppm_to_pwm *(*utils-motif*)*, 119
ppm_to_pwm(), *122*
prob_match *(*utils-motif*)*, 119
prob_match(), *68*, *122*
prob_match_bkg *(*utils-motif*)*, 119
prob_match_bkg(), *68*
processx::run(), *97*
pwm_to_ppm *(*utils-motif*)*, 119
pwm_to_ppm(), *122*

read_cisbp, 86, *88–94*
read_homer, *87*, 87, *89–94*
read_homer(), *136*, *137*
read_jaspar, *87*, *88*, 88, *90–94*
read_jaspar(), *137*
read_matrix, *87–89*, 89, *91–94*
read_matrix(), *139*
read_meme, *87–90*, 90, *92–94*
read_meme(), *18*, *97*, *140*
read_meme(CWM = TRUE), *140*
read_motifs, *87–91*, 92, *93*, *94*
read_motifs(), *92*
read_transfac, *87–92*, 93, *94*
read_transfac(), *142*
read_uniprobe, *87–93*, 94
readRDS(), *92*, *141*
reexports, 95
requires_update *(*tidy-motifs*)*, 110
requires_update(), *111*
RNAStringSet, *22*, *101*
round_motif *(*utils-motif*)*, 119
round_motif(), *122*
rowMeans, *95*
rowMeans *(*reexports*)*, 95
rowMeans,universalmotif-method
*(*universalmotif-class*)*, 114
rownames, *95*
rownames *(*reexports*)*, 95
rownames,universalmotif-method
*(*universalmotif-class*)*, 114
rowSums, *95*
rowSums *(*reexports*)*, 95
rowSums,universalmotif-method
*(*universalmotif-class*)*, 114
run_meme, 95
run_meme(), *95*, *97*

S4Vectors::DataFrame, *34*
sample(), *25*, *67*, *108*
sample_sites, 98
sample_sites(), *40*
saveRDS(), *14*, *92*, *141*
scan_sequences, 98
scan_sequences(), *4*, *5*, *26–29*, *38*, *43*, *44*, *56*, *60*, *63*, *80*, *81*, *85–87*, *99*, *101–103*, *109*, *136*
scan_sequences_lite, *12*, *32*, *51*, *54*, *74*, 102, *135*
scan_sequences_lite(), *12*, *26*, *27*, *29–32*, *39*, *40*, *55–57*, *59–61*, *63*, *65*, *80*, *81*, *85*, *86*, *98*, *101*
score_match *(*utils-motif*)*, 119
score_match(), *68*, *122*
sequence_complexity, 104
sequence_complexity(), *44*, *104*, *126*, *127*
set.seed(), *63*, *64*
show,universalmotif-method
*(*universalmotif-class*)*, 114
shuffle_motifs, 107
shuffle_motifs(), *22*, *42*, *107*, *109*
shuffle_sequences, 108
shuffle_sequences(), *25*, *27–30*, *32*, *38*, *47*, *58*, *59*, *97*, *107*, *108*, *127*
shuffle_string *(*utils-sequence*)*, 125
shuffle_string(), *127*
slide_fun *(*utils-sequence*)*, 125
slide_fun(), *127*
StatLogo *(*geom_motif*)*, 35
StatMotif *(*geom_motif*)*, 35
stats::cutree(), *76*
stats::fisher.test(), *29*, *31*
stats::hclust(), *75*, *76*
stats::p.adjust(), *28*
subset, *95*
subset *(*reexports*)*, 95
subset,universalmotif-method
*(*universalmotif-class*)*, 114
summarise_motifs *(*utils-motif*)*, 119
summarise_motifs(), *122*
switch_alph, 110

tidy-motifs, 110
to_df *(*tidy-motifs*)*, 110
to_df(), *15*, *54*, *59*, *111*
to_list *(*tidy-motifs*)*, 110
to_list(), *15*, *59*, *111*
trim_cwm, 112
trim_cwm(), *112*
trim_motifs, 113
trim_motifs(), *49–51*, *112*, *113*

ungap *(*utils-motif*)*, 119
universalmotif, *4–6*, *15–17*, *21*, *22*, *32*, *33*, *70*, *71*, *87–89*, *91–94*, *115*, *116*, *120*, *122*

universalmotif *(*universalmotif-class*)*, 114
universalmotif-class, 114
universalmotif-package *(*universalmotif-pkg*)*, 118
universalmotif-pkg, 118
update_motifs *(*tidy-motifs*)*, 110
update_motifs(), *111*
utilities, 118
utils-motif, *118*, 119
utils-sequence, *118*, 125

view_logo, 129
view_logo(), *35–37*, *132, 133*, *135*
view_motifs, 130
view_motifs(), *10*, *34–37*, *129–132*, *134, 135*
view_motifs_lite, *12*, *32*, *51*, *54*, *74*, *103*, 133
view_motifs_lite(), *10*, *12*, *36, 37*, *74*, *130*, *133*

window_string *(*utils-sequence*)*, 125
window_string(), *127*
write_homer, 136, *137*, *139–142*
write_jaspar, *137*, 137, *139–142*
write_matrix, *137*, 138, *140–142*
write_meme, *137*, *139*, 139, *141, 142*
write_motifs, *137*, *139, 140*, 141, *142*
write_motifs(), *5*, *92*, *141*
write_transfac, *137*, *139–141*, 142

XStringSet, *4*, *21*, *24, 25*, *27, 28*, *37*, *95*, *98, 99*, *101*, *104*, *108, 109*