Motif comparisons and P-values

Benjamin Jean-Marie Tremblay*

17 October 2021

Abstract

Two important but not often discussed topics with regards to motifs are motif comparisons and P-values. These are explored here, including implementation details and example use cases.

Contents

1 Introduction **1**

2 Motif comparisons **1**
2.1 An overview of available comparison metrics 2
2.2 Comparison parameters 3
2.3 Comparison P-values 6

3 Motif trees with ggtree **6**
3.1 Using `motif_tree()` 6
3.2 Using `compare_motifs()` and `ggtree()` 7
3.3 Plotting motifs alongside trees 8

4 Motif P-values **10**
4.1 The dynamic programming algorithm for calculating P-values and scores 11
4.2 The branch-and-bound algorithm for calculating P-values from scores 14
4.3 The random subsetting algorithm for calculating scores from P-values 15

Session info **17**

References **18**

1 Introduction

This vignette covers motif comparisons (including metrics, parameters and clustering) and P-values. For an introduction to sequence motifs, see the introductory vignette. For a basic overview of available motif-related functions, see the motif manipulation vignette. For sequence-related utilities, see the sequences vignette.

2 Motif comparisons

There a couple of functions available in other Bioconductor packages which allow for motif comparison, such as `PWMSimlarity()` (`TFBSTools`) and `motifSimilarity()` (`PWMEnrich`). Unfortunately these functions are not designed for comparing large numbers of motifs. Furthermore they are restrictive in their option range. The `universalmotif` package aims to fix this by providing the `compare_motifs()` function. Several

*benjamin.tremblay@uwaterloo.ca

other functions also make use of the core `compare_motifs()` functionality, including `merge_motifs()` and `view_motifs()`.

2.1 An overview of available comparison metrics

This function has been written to allow comparisons using any of the following metrics:

- Euclidean distance (`EUCL`)
- 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 average log-likelihood ratio (`ALLR_LL`; minimum column score of -2) (Mahony, Auron, and Benos 2007)
- Bhattacharyya coefficient (`BHAT`) (Bhattacharyya 1943)

For clarity, here are the `R` implementations of these metrics:

```
EUCL <- function(c1, c2) {
  sqrt( sum( (c1 - c2)^2 ) )
}

WEUCL <- function(c1, c2, bkg1, bkg2) {
  sqrt( sum( (bkg1 + bkg2) * (c1 - c2)^2 ) )
}

KL <- function(c1, c2) {
  ( sum(c1 * log(c1 / c2)) + sum(c2 * log(c2 / c1)) ) / 2
}

HELL <- function(c1, c2) {
  sqrt( sum( ( sqrt(c1) - sqrt(c2) )^2 ) ) / sqrt(2)
}

SEUCL <- function(c1, c2) {
  sum( (c1 - c2)^2 )
}

MAN <- function(c1, c2) {
  sum ( abs(c1 - c2) )
}

PCC <- function(c1, c2) {
  n <- length(c1)
  top <- n * sum(c1 * c2) - sum(c1) * sum(c2)
  bot <- sqrt( ( n * sum(c1^2) - sum(c1)^2 ) * ( n * sum(c2^2) - sum(c2)^2 ) )
  top / bot
}

WPCC <- function(c1, c2, bkg1, bkg2) {
```

```
  weights <- bkg1 + bkg2
  mean1 <- sum(weights * c1)
  mean2 <- sum(weights * c2)
  var1 <- sum(weights * (c1 - mean1)^2)
  var2 <- sum(weights * (c2 - mean2)^2)
  cov <- sum(weights * (c1 - mean1) * (c2 - mean2))
  cov / sqrt(var1 * var2)
}

SW <- function(c1, c2) {
  2 - sum( (c1 - c2)^2 )
}

ALLR <- function(c1, c2, bkg1, bkg2, nsites1, nsites2) {
  left <- sum( c2 * nsites2 * log(c1 / bkg1) )
  right <- sum( c1 * nsites1 * log(c2 / bkg2) )
  ( left + right ) / ( nsites1 + nsites2 )
}

BHAT <- function(c1, c2) {
  sum( sqrt(c1 * c2) )
}
```

Motif comparison involves comparing a single column from each motif individually, and adding up the scores from all column comparisons. Since this causes the score to be highly dependent on motif length, the scores can instead be averaged using the arithmetic mean, geometric mean, median, or Fisher Z-transform.

If you're curious as to how the comparison metrics perform, two columns can be compared individually using `compare_columns()`:

```
c1 <- c(0.7, 0.1, 0.1, 0.1)
c2 <- c(0.5, 0.0, 0.2, 0.3)

compare_columns(c1, c2, "PCC")
#> [1] 0.8006408
compare_columns(c1, c2, "EUCL")
#> [1] 0.3162278
```

Note that some metrics do not work with zero values, and small pseudocounts are automatically added to motifs for the following:

- `KL`
- `ALLR`
- `ALLR_LL`

As seen in figure 1, the distributions for random individual column comparisons tend to be very skewed. This is usually remedied when comparing the entire motif, though some metrics still perform poorly in this regard.

```
#> `summarise()` has grouped output by 'key'. You can override using the `.groups`
#> argument.
```

2.2 Comparison parameters

There are several key parameters to keep in mind when comparing motifs. Some of these are:

- `method`: one of the metrics listed previously
- `tryRC`: choose whether to try comparing the reverse complements of each motif as well

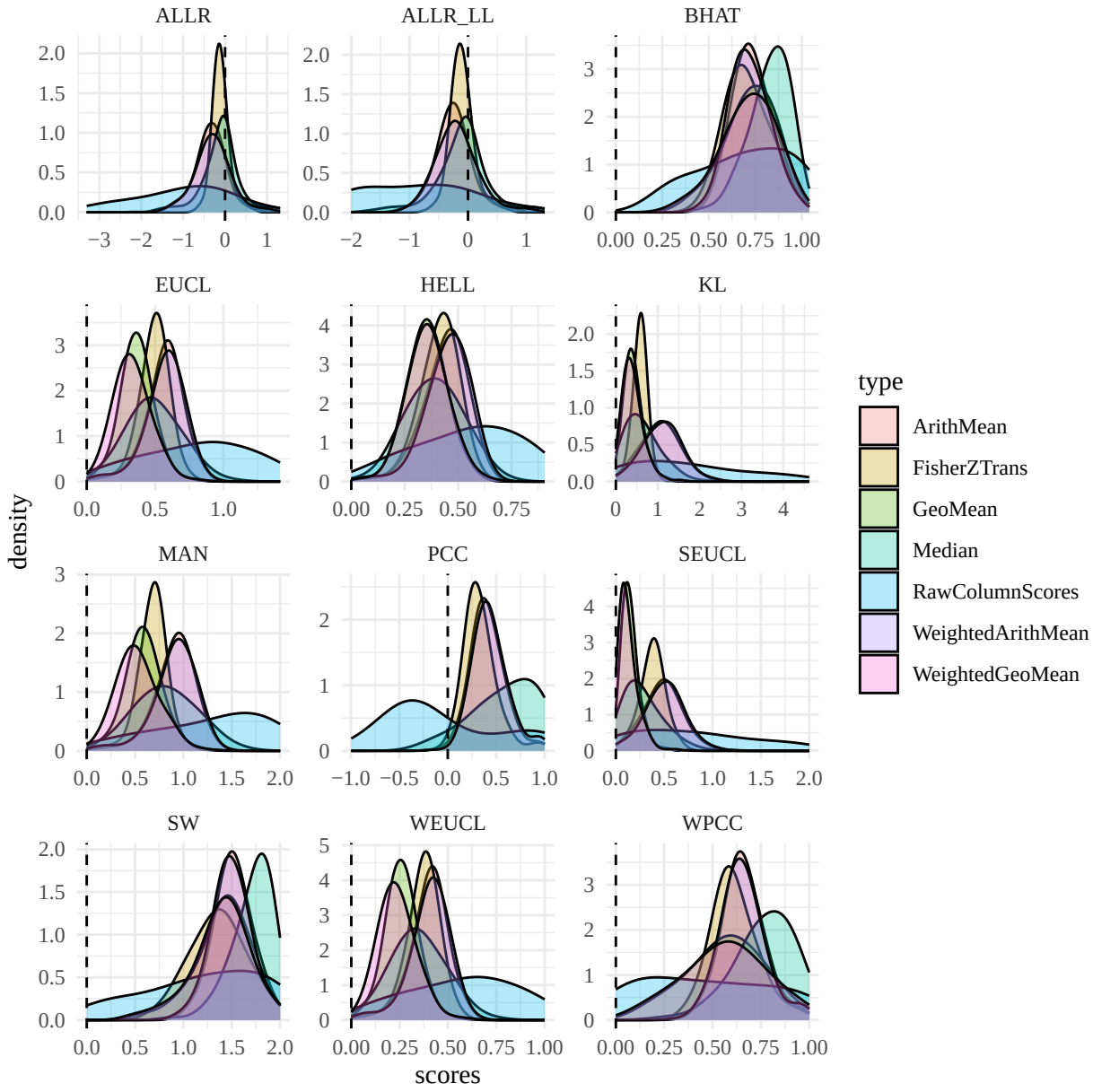


Figure 1: Distributions of scores from approximately 500 random motif and individual column comparisons

- `min.overlap`: limit the amount of allowed overhang between the two motifs
- `min.mean.ic`, `min.position.ic`: don't allow low IC alignments or positions to contribute to the final score
- `score.strat`: how to combine individual column scores in an alignment

See the following example for an idea as to how some of these settings impact scores:

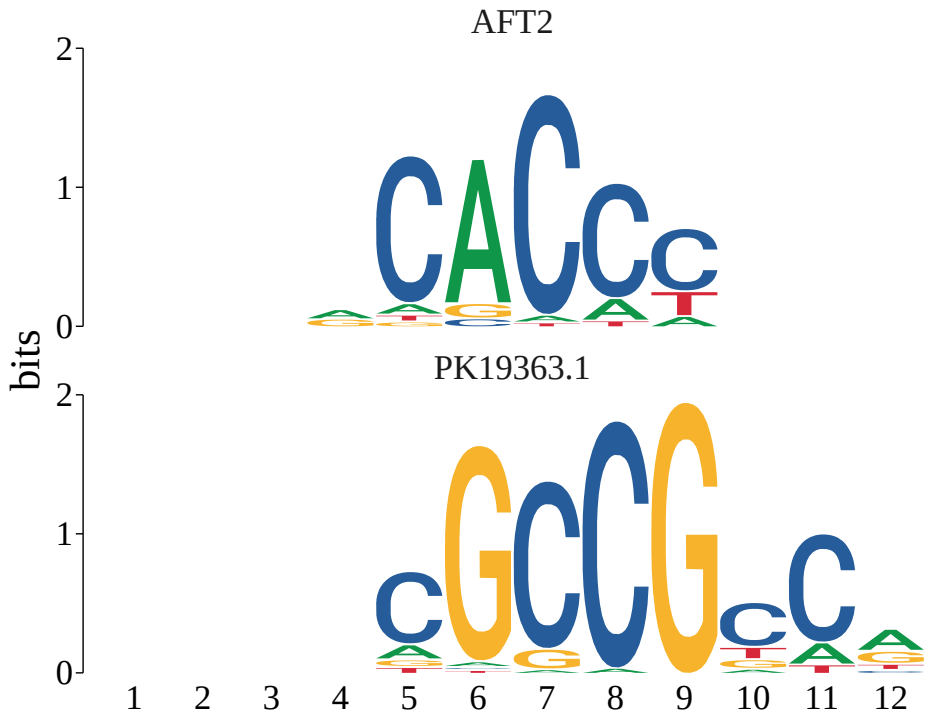


Figure 2: Example scores from comparing two motifs

Table 1: Comparing two motifs with various settings

type	method	default	normalised	checkIC
similarity	PCC	0.5145697	0.3087418	0.9356122
similarity	WPCC	0.6603253	0.5045159	0.9350947
distance	EUCL	0.5489863	0.7401466	0.2841903
similarity	SW	1.5579529	1.2057098	1.8955966
distance	KL	0.9314823	1.2424547	0.1975716
similarity	ALLR	-0.3158358	-0.1895015	0.4577374
similarity	BHAT	0.7533046	0.6026437	0.9468133
distance	HELL	0.4154478	0.2492687	0.2123219
distance	WEUCL	0.3881919	0.5233627	0.2009529
distance	SEUCL	0.4420471	0.2652283	0.1044034
distance	MAN	0.8645563	0.5187338	0.4710645
similarity	ALLR_LL	-0.1706669	-0.1024001	0.4577374

Settings used in the previous table:

- normalised: `normalise.scores = TRUE`
- checkIC: `min.position.ic = 0.25`

2.3 Comparison P-values

By default, `compare_motifs()` will compare all motifs provided and return a matrix. The `compare.to` will cause `compare_motifs()` to return P-values.

```
library(universalmotif)
library(MotifDb)
motifs <- filter_motifs(MotifDb, organism = "Athaliana")
#> motifs converted to class 'universalmotif'

# Compare the first motif with everything and return P-values
head(compare_motifs(motifs, 1))
#> Warning in compare_motifs(motifs, 1): Some comparisons failed due to low motif
#> IC
#> DataFrame with 6 rows and 8 columns
#>       subject subject.i                target  target.i     score   logPval
#>   <character> <numeric>           <character> <integer> <numeric> <numeric>
#> 1       ORA59         1 ERF11 [duplicated #6..      1371  0.991211  -13.5452
#> 2       ORA59         1 CRF4 [duplicated #566]      1195  0.990756  -13.5247
#> 3       ORA59         1                   LOB      1297  0.987357  -13.3725
#> 4       ORA59         1                 ERF15       618  0.977213  -12.9254
#> 5       ORA59         1 ERF2 [duplicated #294]       649  0.973871  -12.7804
#> 6       ORA59         1 ERF2 [duplicated #483]      1033  0.973871  -12.7804
#>          Pval       Eval
#>     <numeric>  <numeric>
#> 1 1.31042e-06 0.00359318
#> 2 1.33754e-06 0.00366754
#> 3 1.55744e-06 0.00427049
#> 4 2.43548e-06 0.00667809
#> 5 2.81553e-06 0.00772019
#> 6 2.81553e-06 0.00772019
```

P-values are made possible by estimating distribution (usually the best fitting distribution for motif comparisons) parameters from randomized motif scores, then using the appropriate `stats::p*()` distribution function to return P-values. These estimated parameters are pre-computed with `make_DBscores()` and stored as `JASPAR2018_CORE_DBSCORES` and `JASPAR2018_CORE_DBSCORES_NORM`. Since changing any of the settings and motif sizes will affect the estimated distribution parameters, estimated parameters have been pre-computed for a variety of these. See `?make_DBscores` if you would like to generate your own set of pre-computed scores using your own parameters and motifs.

3 Motif trees with ggtree

3.1 Using `motif_tree()`

Additionally, this package introduces the `motif_tree()` function for generating basic tree-like diagrams for comparing motifs. This allows for a visual result from `compare_motifs()`. All options from `compare_motifs()` are available in `motif_tree()`. This function uses the `ggtree` package and outputs a `ggplot` object (from the `ggplot2` package), so altering the look of the trees can be done easily after `motif_tree()` has already been run.

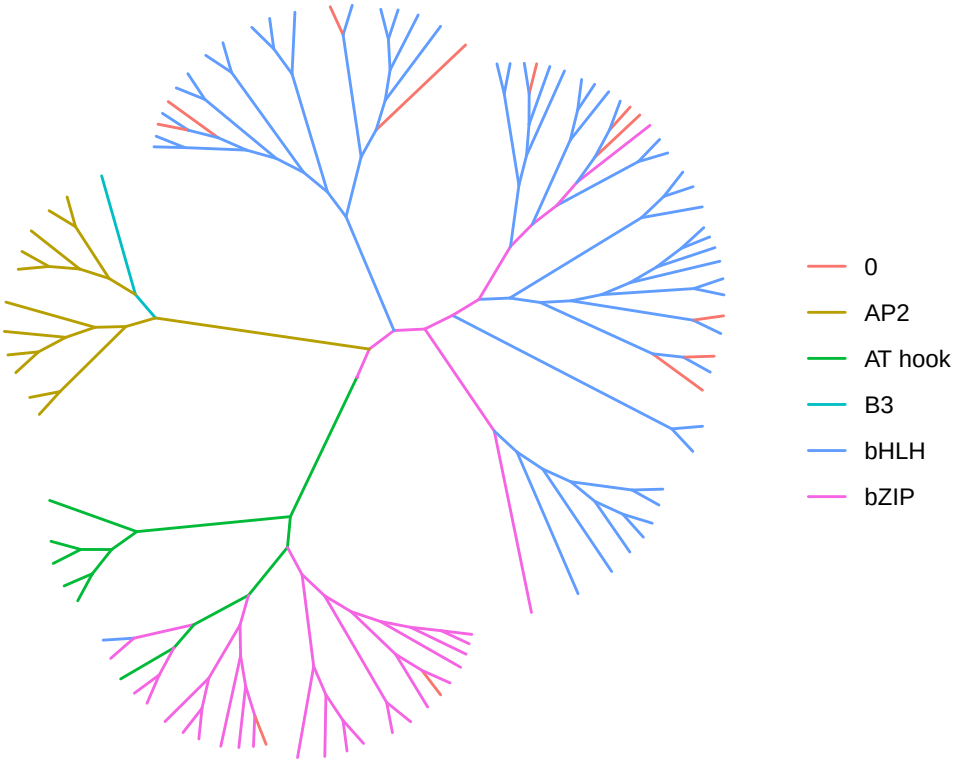
```
library(universalmotif)
library(MotifDb)

motifs <- filter_motifs(MotifDb, family = c("AP2", "B3", "bHLH", "bZIP",
                                            "AT hook"))
```

```
#> motifs converted to class 'universalmotif'
motifs <- motifs[sample(seq_along(motifs), 100)]
tree <- motif_tree(motifs, layout = "daylight", linecol = "family")

## Make some changes to the tree in regular ggplot2 fashion:
# tree <- tree + ...

tree
```



3.2 Using `compare_motifs()` and `ggtree()`

While `motif_tree()` works as a quick and convenient tree-building function, it can be inconvenient when more control is required over tree construction. For this purpose, the following code goes through how exactly `motif_tree()` generates trees.

```
library(universalmotif)
library(MotifDb)
library(ggtree)
library(ggplot2)

motifs <- convert_motifs(MotifDb)
motifs <- filter_motifs(motifs, organism = "Athaliana")
motifs <- motifs[sample(seq_along(motifs), 25)]

## Step 1: compare motifs

comparisons <- compare_motifs(motifs, method = "PCC", min.mean.ic = 0,
                              score.strat = "a.mean")

## Step 2: create a "dist" object
```

```
# The current metric, PCC, is a similarity metric
comparisons <- 1 - comparisons

comparisons <- as.dist(comparisons)

# We also want to extract names from the dist object to match annotations
labels <- attr(comparisons, "Labels")

## Step 3: get the comparisons ready for tree-building

# The R package "ape" provides the necessary "as.phylo" function
comparisons <- ape::as.phylo(hclust(comparisons))

## Step 4: incorporate annotation data to colour tree lines

family <- sapply(motifs, function(x) x["family"])
family.unique <- unique(family)

# We need to create a list with an entry for each family; within each entry
# are the names of the motifs belonging to that family
family.annotations <- list()
for (i in seq_along(family.unique)) {
  family.annotations <- c(family.annotations,
                          list(labels[family %in% family.unique[i]]))
}
names(family.annotations) <- family.unique

# Now add the annotation data:
comparisons <- ggtree::groupOTU(comparisons, family.annotations)

## Step 5: draw the tree

tree <- ggtree(comparisons, aes(colour = group), layout = "rectangular") +
          theme(legend.position = "bottom", legend.title = element_blank())

## Step 6: add additional annotations

# If we wish, we can additional annotations such as tip labelling and size

# Tip labels:
tree <- tree + geom_tiplab()

# Tip size:
tipsize <- data.frame(label = labels,
                      icscore = sapply(motifs, function(x) x["icscore"]))

tree <- tree %<+% tipsize + geom_tippoint(aes(size = icscore))
```

3.3 Plotting motifs alongside trees

Unfortunately, the `universalmotif` package does not provide any function to easily plot motifs as part of trees (as is possible via the `motifStack` package). However, it can be done (somewhat roughly) by plotting a tree and a set of motifs side by side. In the following example, the `cowplot` package is used to glue the two

plots together, though other packages which perform this function are available.

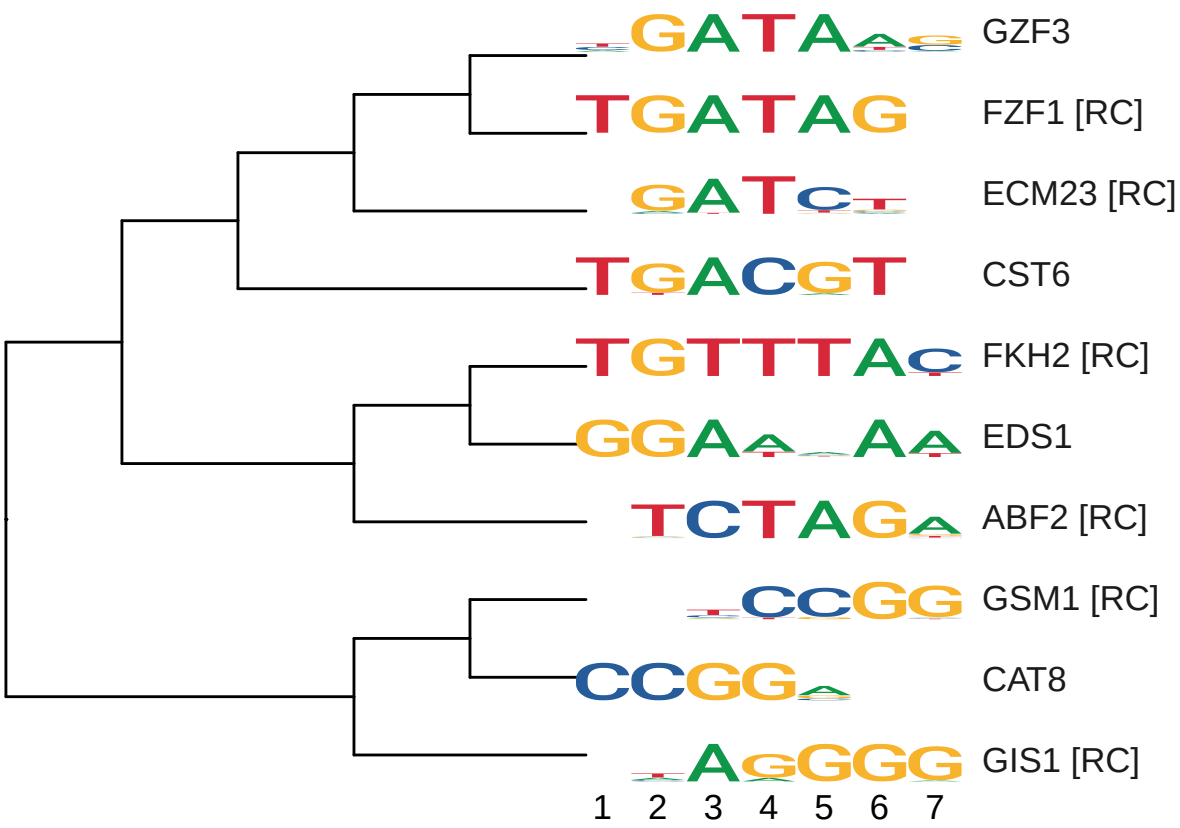
```
library(universalmotif)
library(MotifDb)
library(cowplot)

## Get our starting set of motifs:
motifs <- convert_motifs(MotifDb[1:10])

## Get the tree: make sure it's a horizontal type layout
tree <- motif_tree(motifs, layout = "rectangular", linecol = "none")

## Now, make sure we order our list of motifs to match the order of tips:
mot.names <- sapply(motifs, function(x) x["name"])
names(motifs) <- mot.names
new.order <- tree$data$label[tree$data$isTip]
new.order <- rev(new.order[order(tree$data$y[tree$data$isTip])])
motifs <- motifs[new.order]

## Plot the two together (finessing of margins and positions may be required):
plot_grid(nrow = 1, rel_widths = c(1, -0.15, 1),
  tree + xlab(""), NULL,
  view_motifs(motifs, names.pos = "right") +
    ylab(element_blank()) +
    theme(
      axis.line.y = element_blank(),
      axis.ticks.y = element_blank(),
      axis.text.y = element_blank(),
      axis.text = element_text(colour = "white")
    )
)
```



4 Motif P-values

Motif P-values are not usually discussed outside of the bioinformatics literature, but are actually quite a challenging topic. To illustrate this, consider the following example motif:

```
library(universalmotif)

m <- matrix(c(0.10,0.27,0.23,0.19,0.29,0.28,0.51,0.12,0.34,0.26,
              0.36,0.29,0.51,0.38,0.23,0.16,0.17,0.21,0.23,0.36,
              0.45,0.05,0.02,0.13,0.27,0.38,0.26,0.38,0.12,0.31,
              0.09,0.40,0.24,0.30,0.21,0.19,0.05,0.30,0.31,0.08),
            byrow = TRUE, nrow = 4)
motif <- create_motif(m, alphabet = "DNA", type = "PWM")
motif
#>
#>        Motif name:   motif
#>          Alphabet:   DNA
#>              Type:   PWM
#>           Strands:   +-
#>          Total IC:   10.03
#>       Pseudocount:   0
#>         Consensus:   SHCNNNRNNV
#>
#>       S     H     C     N     N     N     R     N     N     V
#> A -1.32  0.10 -0.12 -0.40  0.21  0.15  1.04 -1.07  0.44  0.04
#> C  0.53  0.20  1.03  0.60 -0.12 -0.66 -0.54 -0.27 -0.12  0.51
#> G  0.85 -2.34 -3.64 -0.94  0.11  0.59  0.07  0.59 -1.06  0.30
```

```
#> T -1.47  0.66 -0.06  0.26 -0.25 -0.41 -2.31  0.25  0.31 -1.66
```

Let us then use this motif with `scan_sequences()`:

```
data(ArabidopsisPromoters)

res <- scan_sequences(motif, ArabidopsisPromoters, verbose = 0,
  calc.pvals = FALSE, threshold = 0.8, threshold.type = "logodds")
head(res)
#> DataFrame with 6 rows and 13 columns
#>         motif   motif.i    sequence sequence.i     start      stop     score
#>   <character> <integer> <character>  <integer> <integer> <integer> <numeric>
#> 1       motif         1   AT1G08090         21       925       934     5.301
#> 2       motif         1   AT1G49840         27       980       989     5.292
#> 3       motif         1   AT1G76590         19       848       857     5.869
#> 4       motif         1   AT2G15390          6       337       346     5.643
#> 5       motif         1   AT3G57640         33       174       183     5.510
#> 6       motif         1   AT4G14365         35       799       808     5.637
#>         match thresh.score min.score max.score score.pct      strand
#>   <character>    <numeric> <numeric> <numeric> <numeric> <character>
#> 1  CTCCAAAGAA       5.2248     -15.4     6.531   81.1667           +
#> 2  CTCTGGATTC       5.2248     -15.4     6.531   81.0289           +
#> 3  CTCTAGAGAC       5.2248     -15.4     6.531   89.8637           +
#> 4  CCCCGGAGAC       5.2248     -15.4     6.531   86.4033           +
#> 5  GCCCAGATAG       5.2248     -15.4     6.531   84.3669           +
#> 6  CTCCAAAGTC       5.2248     -15.4     6.531   86.3114           +
```

Now let us imagine that we wish to rank these matches by P-value. First, we must calculate the match probabilities:

```
## One of the matches was CTCTAGAGAC, with a score of 5.869 (max possible = 6.531)

bkg <- get_bkg(ArabidopsisPromoters, 1)
bkg <- structure(bkg$probability, names = bkg$klet)
bkg
#>       A       C       G       T
#> 0.34768 0.16162 0.15166 0.33904
```

Now, use these to calculate the probability of getting CTCTAGAGAC.

```
hit.prob <- bkg["A"]^3 * bkg["C"]^3 * bkg["G"]^2 * bkg["T"]^2
hit.prob <- unname(hit.prob)
hit.prob
#> [1] 4.691032e-07
```

Calculating the probability of a single match was easy, but then comes the challenging part: calculating the probability of all possible matches with a score higher than 5.869, and then summing these. This final sum then represents the probability of finding a match which scores at least 5.869. One way is to list all possible sequence combinations, then filtering based on score; however this "brute force" approach is unreasonable for all but the smallest of motifs.

4.1 The dynamic programming algorithm for calculating P-values and scores

Instead of trying to find and calculate the probabilities of all matches with a score or higher than the query score, one can use a dynamic programming algorithm to generate a much smaller distribution of probabilities for the possible range of scores using set intervals. This method is implemented by the FIMO tool (Grant,

Bailey, and Noble 2011). The theory behind it is also explained in Gupta et al. (2007), though the purpose of the algorithm is for motif comparison instead of motif P-values (however it is the same algorithm). The basic concept will also be briefly explained here.

For each individual position-letter score in the PWM, the chance of getting that score from the respective background probability of that letter is added to the intervals in which getting that specific score could allow the final score to land. Once this probability distribution is generated, it can be converted to a cumulative distribution and re-used for any input P-value/score to output the equivalent score/P-value. For P-value inputs, it finds the specific score interval where the accompanying P-value in the cumulative distribution smaller or equal to it, then reports the score of the previous interval. For score inputs, the scores are rounded to the nearest interval in the cumulative distribution and the accompanying P-value retrieved. The major advantages of this method include only looking for the probabilities of the range of scores with a set interval, cutting down on needing to find the probabilities of all actual possible scores (and thus increasing performance by several orders of magnitude for larger/higher-order motifs), and being able to re-use the distribution for any number of query P-value/scores. Although this method involves rounding off scores to allow a small set interval, in practice in the `universalmotif` package it offers the same maximum possible level of accuracy as the exhaustive method (described in the next section) as motif PWMs are always internally rounded to a thousandth of a decimal place for speed. This leaves as the only downside the inability to allow non-finite values to exist in the PWM (e.g. from zero-probabilities) since then a known range with set intervals could not possibly be created.

Going back to our example, we can see this in action using the `motif_pvalue()` function:

```
res <- res[1:6, ]
pvals <- motif_pvalue(motif, res$score, bkg.probs = bkg)
res2 <- data.frame(motif=res$motif,match=res$match,pval=pvals)[order(pvals), ]
knitr::kable(res2, digits = 22, row.names = FALSE, format = "markdown")
```

motif	match	pval
motif	CTCTAGAGAC	0.001495587
motif	CCCCGGAGAC	0.001947592
motif	CTCCAAAGTC	0.001962531
motif	GCCCAGATAG	0.002257443
motif	CTCCAAAGAA	0.002825922
motif	CTCTGGATTC	0.002852671

To illustrate that we can also do the inverse of this calculation:

```
res$score
#> [1] 5.301 5.292 5.869 5.643 5.510 5.637
motif_pvalue(motif, pvalue = pvals, bkg.probs = bkg)
#> [1] 5.301 5.292 5.869 5.643 5.510 5.637
```

You may occasionally see slight errors at the last couple of digits. These are generally unavoidable to the internal rounding mechanisms of the `universalmotif` package.

Let us consider more examples, such as the following larger motif:

```
data(ArabidopsisMotif)
ArabidopsisMotif
#>
#>        Motif name:   YTTTYTTTTTYTTTY
#>          Alphabet:   DNA
#>              Type:   PPM
#>           Strands:   +-
#>          Total IC:   15.99
```

```
#>       Pseudocount:   1
#>         Consensus:   YTYTYTTYTTYTTTY
#>      Target sites:   617
#>           E-value:   2.5e-87
#>
#>      Y    T    Y    T    Y    T    T    Y    T    T    Y    T    T    T    Y
#> A 0.01 0.00 0.00 0.00 0.00 0.06 0.00 0.01 0.00 0.00 0.02 0.00 0.00 0.00 0.00
#> C 0.30 0.17 0.31 0.01 0.54 0.02 0.24 0.25 0.22 0.04 0.39 0.21 0.16 0.18 0.43
#> G 0.16 0.05 0.03 0.01 0.00 0.02 0.11 0.00 0.04 0.05 0.03 0.01 0.02 0.00 0.11
#> T 0.53 0.78 0.66 0.98 0.45 0.90 0.66 0.74 0.74 0.91 0.55 0.77 0.83 0.82 0.46
```

Using the `motif_range()` utility, we can get an idea of the possible range of scores:

```
motif_range(ArabidopsisMotif)
#>      min      max
#> -125.070   18.784
```

We can use these ranges to confirm our cumulative distribution of P-values:

```
(pvals2 <- motif_pvalue(ArabidopsisMotif, score = motif_range(ArabidopsisMotif)))
#> [1] 1.000000e+00 2.143914e-09
```

And again, going back to scores from these P-values:

```
motif_pvalue(ArabidopsisMotif, pvalue = pvals2)
#> [1] -125.070   18.784
```

As a note: if you ever provide scores which are outside the possible ranges, then you will get the following behaviour:

```
motif_pvalue(ArabidopsisMotif, score = c(-200, 100))
#> [1] 1 0
```

We can also use this function for the higher-order `multifreq` motif representation.

```
data(examplemotif2)
examplemotif2["multifreq"]["2"]
#> $`2`
#>      1   2   3   4   5 6
#> AA 0.0 0.5 0.5 0.5 0.0 0
#> AC 0.0 0.0 0.0 0.0 0.5 0
#> AG 0.0 0.0 0.0 0.0 0.0 0
#> AT 0.0 0.0 0.0 0.0 0.0 0
#> CA 0.5 0.0 0.0 0.0 0.0 0
#> CC 0.0 0.0 0.0 0.0 0.0 1
#> CG 0.0 0.0 0.0 0.0 0.0 0
#> CT 0.5 0.0 0.0 0.0 0.0 0
#> GA 0.0 0.0 0.0 0.0 0.0 0
#> GC 0.0 0.0 0.0 0.0 0.0 0
#> GG 0.0 0.0 0.0 0.0 0.0 0
#> GT 0.0 0.0 0.0 0.0 0.0 0
#> TA 0.0 0.0 0.0 0.0 0.0 0
#> TC 0.0 0.0 0.0 0.0 0.5 0
#> TG 0.0 0.0 0.0 0.0 0.0 0
#> TT 0.0 0.5 0.5 0.5 0.0 0
motif_range(examplemotif2, use.freq = 2)
#>     min     max
```

```
#> -39.948  18.921
motif_pvalue(examplemotif2, score = 15, use.freq = 2)
#> [1] 1.907349e-06
motif_pvalue(examplemotif2, pvalue = 0.00001, use.freq = 2)
#> [1] 9.276
```

Feel free to use this function with any alphabets, such as amino acid motifs or even made up ones!

```
(m <- create_motif(alphabet = "QWERTY"))
#>
#>        Motif name:   motif
#>          Alphabet:   EQRTWY
#>              Type:   PPM
#>          Total IC:   16.51
#>       Pseudocount:   0
#>
#>   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#> E 0.00 0.00 0.13 0.60 0.00 0.00 0.00 0.00 0.58  0.00
#> Q 0.80 0.97 0.13 0.28 0.05 0.00 0.00 0.24 0.36  0.00
#> R 0.00 0.03 0.66 0.00 0.00 0.00 0.00 0.00 0.02  0.00
#> T 0.00 0.00 0.01 0.00 0.00 0.00 0.35 0.01 0.02  0.44
#> W 0.12 0.00 0.08 0.02 0.91 0.99 0.00 0.00 0.02  0.32
#> Y 0.08 0.00 0.00 0.11 0.04 0.01 0.64 0.75 0.00  0.24
motif_pvalue(m, pvalue = c(1, 0.1, 0.001, 0.0001, 0.00001))
#> [1] -166.6222  -31.3840   -0.0500    8.0760   13.6220
```

4.2 The branch-and-bound algorithm for calculating P-values from scores

The alternative to the dynamic programming algorithm is to exhaustively find all actual possible hits with a score equal to or greater than the input score. Generally there is no advantage to solving this exhaustively, with the exception that it allows non-finite values to be present (i.e., zero-probability letters which were not pseudocount-adjusted during the calculation of the PWM). A few algorithms have been proposed to make solving this problem exhaustively more efficient, but the method adopted by the `universalmotif` package is that of Luehr, Hartmann, and Söding (2012). The authors propose using a branch-and-bound[1] algorithm (with a few tricks) alongside a certain approximation. Briefly: motifs are first reorganized so that the highest scoring positions and letters are considered first in the branch-and-bound algorithm. Then, motifs past a certain width (in the original paper, 10) are split in sub-motifs. All possible combinations are found in these sub-motifs using the branch-and-bound algorithm, and P-values calculated for the sub-motifs. Finally, the P-values are combined.

The `motif_pvalue()` function modifies this process slightly by allowing the size of the sub-motifs to be specified via the `k` parameter; and additionally, whereas the original implementation can only calculate P-values for motifs with a maximum of 17 positions (and motifs can only be split in at most two), the `universalmotif` implementation allows for any length of motif to be used (and motifs can be split any number of times). Changing `k` allows one to decide between speed and accuracy; smaller `k` leads to faster but worse approximations, and larger `k` leads to slower but better approximations. If `k` is equal to the width of the motif, then the calculation is *exact*. Is it important to note however that this is is still a computationally intenstive task for larger motifs unless it is broken up into several sub-motifs, though at this point significant accuracy is lost due to the high level of approximation.

Now, let us return to our original example, and this time for the branch-and-bound algorithm set `method = "exhaustive"`:

[1] https://en.wikipedia.org/wiki/Branch_and_bound

```
res <- res[1:6, ]
pvals <- motif_pvalue(motif, res$score, bkg.probs = bkg, method = "e")
res2 <- data.frame(motif=res$motif,match=res$match,pval=pvals)[order(pvals), ]
knitr::kable(res2, digits = 22, row.names = FALSE, format = "markdown")
```

motif	match	pval
motif	CTCTAGAGAC	0.001494052
motif	CCCCGGAGAC	0.001944162
motif	CTCCAAAGTC	0.001960741
motif	GCCCAGATAG	0.002255555
motif	CTCCAAAGAA	0.002823098
motif	CTCTGGATTC	0.002848363

The default `k` in `motif_pvalue()` is 8. I have found this to be a good tradeoff between speed and P-value correctness.

To demonstrate the effect that `k` has on the output P-value, consider the following (and also note that for this motif `k = 10` represents an exact calculation):

```
scores <- c(-6, -3, 0, 3, 6)
k <- c(2, 4, 6, 8, 10)
out <- data.frame(k = c(2, 4, 6, 8, 10),
                  score.minus6 = rep(0, 5),
                  score.minus3 = rep(0, 5),
                  score.0 = rep(0, 5),
                  score.3 = rep(0, 5),
                  score.6 = rep(0, 5))

for (i in seq_along(scores)) {
  for (j in seq_along(k)) {
    out[j, i + 1] <- motif_pvalue(motif, scores[i], k = k[j], bkg.probs = bkg,
      method = "e")
  }
}

knitr::kable(out, format = "markdown", digits = 10)
```

k	score.minus6	score.minus3	score.0	score.3	score.6
2	0.9692815	0.6783292	0.2241568	0.01809649	0.0000000000
4	0.8516271	0.4945960	0.1581260	0.02271134	0.0009788176
6	0.7647867	0.4298417	0.1418337	0.02291211	0.0012812392
8	0.7647867	0.4298417	0.1418337	0.02291211	0.0012812392
10	0.7649169	0.4299862	0.1419202	0.02293202	0.0012830021

For this particular motif, while the approximation worsens slightly as `k` decreases, it is still quite accurate when the number of motif subsets is limited to two. Usually, you should only have to worry about `k` for longer motifs (such as those sometimes generated by `MEME`), where the number of sub-motifs increases.

4.3 The random subsetting algorithm for calculating scores from P-values

Similarly to calculating P-values, exact scores can be calculated from small motifs, and approximate scores from big motifs using subsetting. When an exact calculation is performed, all possible scores are extracted

and a quantile function extracts the appropriate score. For approximate calculations, the overall set of scores are approximate several times by randomly adding up all possible scores from each `k` subset before a quantile function is used.

Starting from a set of P-values and setting `method = "exhaustive"`:

```
bkg <- c(A=0.25, C=0.25, G=0.25, T=0.25)
pvals <- c(0.1, 0.01, 0.001, 0.0001, 0.00001)
scores <- motif_pvalue(motif, pvalue = pvals, bkg.probs = bkg, k = 10,
  method = "e")

scores.approx6 <- motif_pvalue(motif, pvalue = pvals, bkg.probs = bkg, k = 6,
  method = "e")
scores.approx8 <- motif_pvalue(motif, pvalue = pvals, bkg.probs = bkg, k = 8,
  method = "e")

pvals.exact <- motif_pvalue(motif, score = scores, bkg.probs = bkg, k = 10,
  method = "e")

pvals.approx6 <- motif_pvalue(motif, score = scores, bkg.probs = bkg, k = 6,
  method = "e")
pvals.approx8 <- motif_pvalue(motif, score = scores, bkg.probs = bkg, k = 8,
  method = "e")

res <- data.frame(pvalue = pvals, score = scores,
                  pvalue.exact = pvals.exact,
                  pvalue.k6 = pvals.approx6,
                  pvalue.k8 = pvals.approx8,
                  score.k6 = scores.approx6,
                  score.k8 = scores.approx8)
knitr::kable(res, format = "markdown", digits = 22)
```

pvalue	score	pvalue.exact	pvalue.k6	pvalue.k8	score.k6	score.k8
1e-01	1.324	1.000299e-01	9.995747e-02	9.995747e-02	1.3221	1.3249
1e-02	3.596	1.000309e-02	9.991646e-03	9.991646e-03	3.6130	3.5963
1e-03	4.858	1.000404e-03	9.984970e-04	9.984970e-04	4.7796	4.8562
1e-04	5.647	1.001358e-04	9.727478e-05	9.727478e-05	5.5680	5.6690
1e-05	6.182	1.049042e-05	9.536743e-06	9.536743e-06	5.5691	6.1333

Starting from a set of scores:

```
bkg <- c(A=0.25, C=0.25, G=0.25, T=0.25)
scores <- -2:6
pvals <- motif_pvalue(motif, score = scores, bkg.probs = bkg, k = 10,
  method = "e")

scores.exact <- motif_pvalue(motif, pvalue = pvals, bkg.probs = bkg, k = 10,
  method = "e")

scores.approx6 <- motif_pvalue(motif, pvalue = pvals, bkg.probs = bkg, k = 6,
  method = "e")
scores.approx8 <- motif_pvalue(motif, pvalue = pvals, bkg.probs = bkg, k = 8,
  method = "e")
```

```
pvals.approx6 <- motif_pvalue(motif, score = scores, bkg.probs = bkg, k = 6,
  method = "e")
pvals.approx8 <- motif_pvalue(motif, score = scores, bkg.probs = bkg, k = 8,
  method = "e")

res <- data.frame(score = scores, pvalue = pvals,
                  pvalue.k6 = pvals.approx6,
                  pvalue.k8 = pvals.approx8,
                  score.exact = scores.exact,
                  score.k6 = scores.approx6,
                  score.k8 = scores.approx8)
knitr::kable(res, format = "markdown", digits = 22)
```

score	pvalue	pvalue.k6	pvalue.k8	score.exact	score.k6	score.k8
-2	4.627047e-01	4.625721e-01	4.625721e-01	-2.000	-2.0332	-1.9985
-1	3.354645e-01	3.353453e-01	3.353453e-01	-1.000	-0.9818	-0.9997
0	2.185555e-01	2.184534e-01	2.184534e-01	0.000	-0.0093	0.0012
1	1.244183e-01	1.243525e-01	1.243525e-01	1.000	1.0149	0.9937
2	5.911160e-02	5.907822e-02	5.907822e-02	2.000	1.9890	2.0009
3	2.163410e-02	2.160931e-02	2.160931e-02	3.000	2.9962	2.9957
4	5.360603e-03	5.347252e-03	5.347252e-03	4.000	4.0890	3.9877
5	7.162094e-04	7.152557e-04	7.152557e-04	5.000	5.0514	5.0258
6	2.193451e-05	2.193451e-05	2.193451e-05	6.057	5.4652	6.0133

As you may have noticed, results from exact calculations are not *quite* exact. This is due to the `universalmotif` package rounding off values internally for speed.

Session info

```
#> R version 4.4.1 (2024-06-14)
#> Platform: x86_64-apple-darwin20
#> Running under: macOS Monterey 12.7.1
#>
#> Matrix products: default
#> BLAS:   /Library/Frameworks/R.framework/Versions/4.4-x86_64/Resources/lib/libRblas.0.dylib
#> LAPACK: /Library/Frameworks/R.framework/Versions/4.4-x86_64/Resources/lib/libRlapack.dylib;  LAPACK v
#>
#> locale:
#> [1] C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
#>
#> time zone: America/New_York
#> tzcode source: internal
#>
#> attached base packages:
#> [1] stats4    stats     graphics  grDevices utils     datasets  methods
#> [8] base
#>
#> other attached packages:
#>  [1] cowplot_1.1.3         dplyr_1.1.4           ggtree_3.13.1
#>  [4] ggplot2_3.5.1         MotifDb_1.47.0        GenomicRanges_1.57.1
#>  [7] Biostrings_2.73.1     GenomeInfoDb_1.41.1   XVector_0.45.0
```

```
#> [10] IRanges_2.39.2        S4Vectors_0.43.2      BiocGenerics_0.51.1
#> [13] universalmotif_1.23.6
#>
#> loaded via a namespace (and not attached):
#>  [1] tidyselect_1.2.1            farver_2.1.2
#>  [3] bitops_1.0-8                fastmap_1.2.0
#>  [5] RCurl_1.98-1.16             lazyeval_0.2.2
#>  [7] GenomicAlignments_1.41.0    XML_3.99-0.17
#>  [9] digest_0.6.37               lifecycle_1.0.4
#> [11] tidytree_0.4.6              magrittr_2.0.3
#> [13] compiler_4.4.1              rlang_1.1.4
#> [15] tools_4.4.1                 utf8_1.2.4
#> [17] yaml_2.3.10                 data.table_1.16.0
#> [19] rtracklayer_1.65.0          knitr_1.48
#> [21] labeling_0.4.3              S4Arrays_1.5.7
#> [23] curl_5.2.2                  splitstackshape_1.4.8
#> [25] DelayedArray_0.31.11        aplot_0.2.3
#> [27] abind_1.4-8                 BiocParallel_1.39.0
#> [29] withr_3.0.1                 purrr_1.0.2
#> [31] grid_4.4.1                  fansi_1.0.6
#> [33] colorspace_2.1-1            scales_1.3.0
#> [35] MASS_7.3-61                 tinytex_0.52
#> [37] SummarizedExperiment_1.35.1 cli_3.6.3
#> [39] rmarkdown_2.28              crayon_1.5.3
#> [41] treeio_1.29.1               generics_0.1.3
#> [43] httr_1.4.7                  rjson_0.2.22
#> [45] ape_5.8                     zlibbioc_1.51.1
#> [47] parallel_4.4.1              ggplotify_0.1.2
#> [49] restfulr_0.0.15             matrixStats_1.4.1
#> [51] vctrs_0.6.5                 yulab.utils_0.1.7
#> [53] Matrix_1.7-0                jsonlite_1.8.8
#> [55] bookdown_0.40               gridGraphics_0.5-1
#> [57] patchwork_1.2.0             tidyr_1.3.1
#> [59] glue_1.7.0                  codetools_0.2-20
#> [61] gtable_0.3.5                BiocIO_1.15.2
#> [63] UCSC.utils_1.1.0            munsell_0.5.1
#> [65] tibble_3.2.1                pillar_1.9.0
#> [67] htmltools_0.5.8.1           GenomeInfoDbData_1.2.12
#> [69] R6_2.5.1                    evaluate_0.24.0
#> [71] Biobase_2.65.1              lattice_0.22-6
#> [73] highr_0.11                  Rsamtools_2.21.1
#> [75] ggfun_0.1.6                 Rcpp_1.0.13
#> [77] SparseArray_1.5.34          nlme_3.1-166
#> [79] xfun_0.47                   MatrixGenerics_1.17.0
#> [81] fs_1.6.4                    pkgconfig_2.0.3
```

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.

Grant, C. E., T. L. Bailey, and W. S. Noble. 2011. "FIMO: Scanning for Occurrences of a Given Motif." *Bioinformatics* 27: 1017–8.

Gupta, S., J. A. Stamatoyannopoulos, T. L. Bailey, and W. S. Noble. 2007. "Quantifying Similarity Between Motifs." *Genome Biology* 8: R24.

Hellinger, Ernst. 1909. "Neue Begründung Der Theorie Quadratischer Formen von Unendlichvielen Veränderlichen." *Journal Für Die Reine Und Angewandte Mathematik* 136: 210–71.

Kullback, S., and R. A. Leibler. 1951. "On Information and Sufficiency." *The Annals of Mathematical Statistics* 22: 79–86.

Luehr, S., H. Hartmann, and J. Söding. 2012. "The XXmotif Web Server for EXhaustive, Weight MatriX-Based Motif Discovery in Nucleotide Sequences." *Nucleic Acids Research* 40: W104–W109.

Mahony, S., P. E. Auron, and P. V. Benos. 2007. "DNA Familial Binding Profiles Made Easy: Comparison of Various Motif Alignment and Clustering Strategies." *PLoS Computational Biology* 3 (3): e61.

Roepcke, S., S. Grossmann, S. Rahmann, and M. Vingron. 2005. "T-Reg Comparator: An Analysis Tool for the Comparison of Position Weight Matrices." *Nucleic Acids Research* 33: W438–W441.

Sandelin, A., and W. W. Wasserman. 2004. "Constrained Binding Site Diversity Within Families of Transcription Factors Enhances Pattern Discovery Bioinformatics." *Journal of Molecular Biology* 338 (2): 207–15.

Wang, T., and G. D. Stormo. 2003. "Combining Phylogenetic Data with Co-Regulated Genes to Identify Motifs." *Bioinformatics* 19 (18): 2369–80.