Motif comparisons and P-values

Benjamin Jean-Marie Tremblay*

5 May 2026

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 **2**
2.1 An overview of available comparison metrics 2
2.2 Comparison parameters 5
2.3 Comparison P-values 6
2.4 A faster alternative: `compare_motifs_lite()` 6

3 Merging motifs **8**
3.1 A faster alternative: `merge_motifs_lite()` / `merge_similar_lite()` 8

4 Motif trees with ggtree **10**
4.1 Using `motif_tree()` 10
4.2 Using `compare_motifs()` and `ggtree()` 10
4.3 Plotting motifs alongside trees 12

5 Motif P-values **13**
5.1 The dynamic programming algorithm for calculating P-values and scores 14
5.2 The branch-and-bound algorithm for calculating P-values from scores 17
5.3 The random subsetting algorithm for calculating scores from P-values 19

Session info **20**

References **22**

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.

*benjamin.tremblay@uwaterloo.ca

2 Motif comparisons

There are a couple of functions available in other Bioconductor packages which allow for motif comparison, such as `PWMSimilarity()` (`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()` / `compare_motifs_lite()` function. Several other functions also make use of the core `compare_motifs()` / `compare_motifs_lite()` functionality, including `merge_motifs()` / `merge_motifs_lite()` and `view_motifs()` / `view_motifs_lite()`. (The `2` versions of these functions are newer, faster, simplified versions.) The following introduction of motif comparison is specifically concerning the original fully-featured `compare_motifs()`

2.1 An overview of available comparison metrics

The `compare_motifs()` 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.

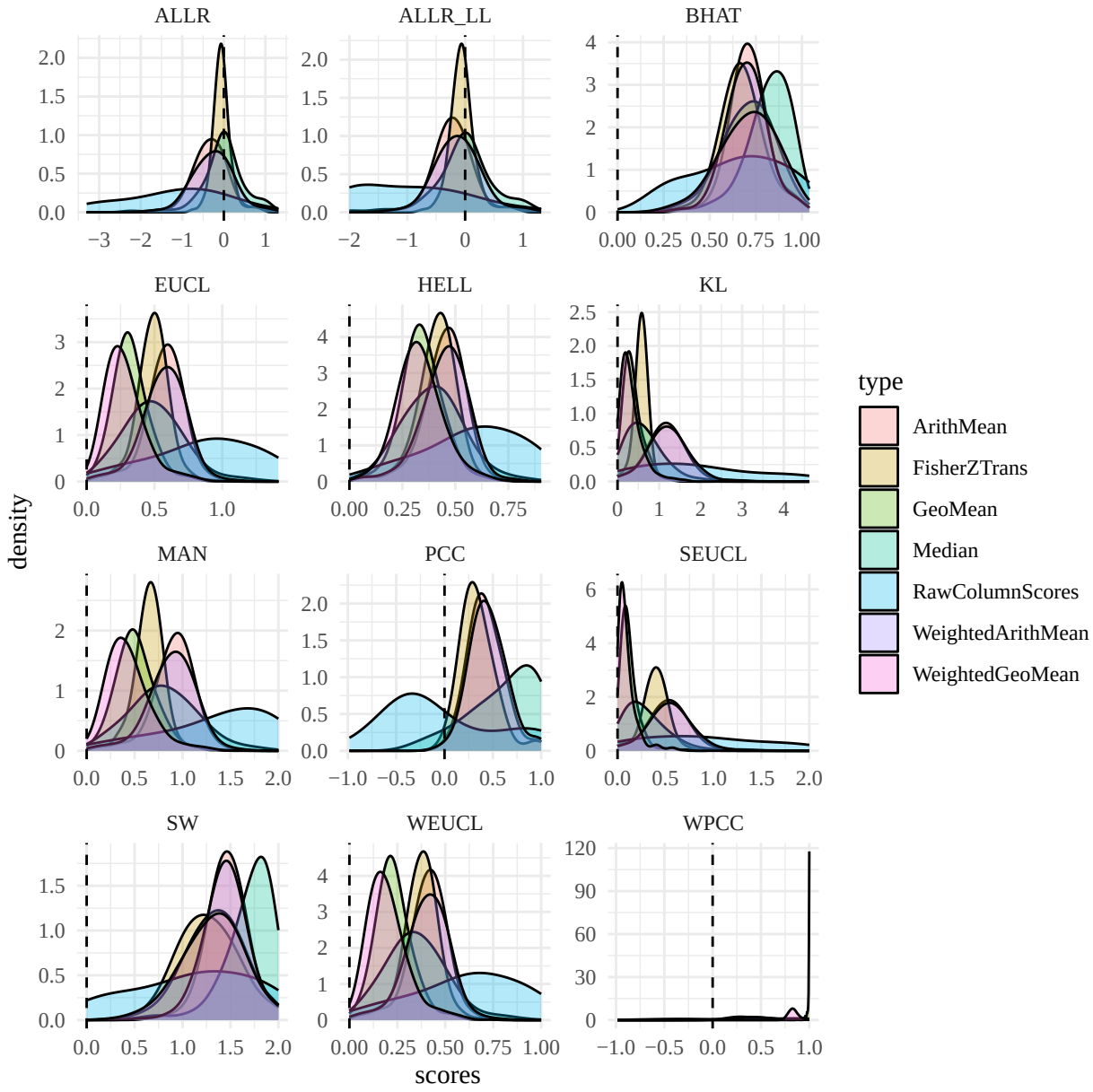


Figure 1: Distributions of scores from approximately 500 random motif and individual column comparisons

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
- `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:

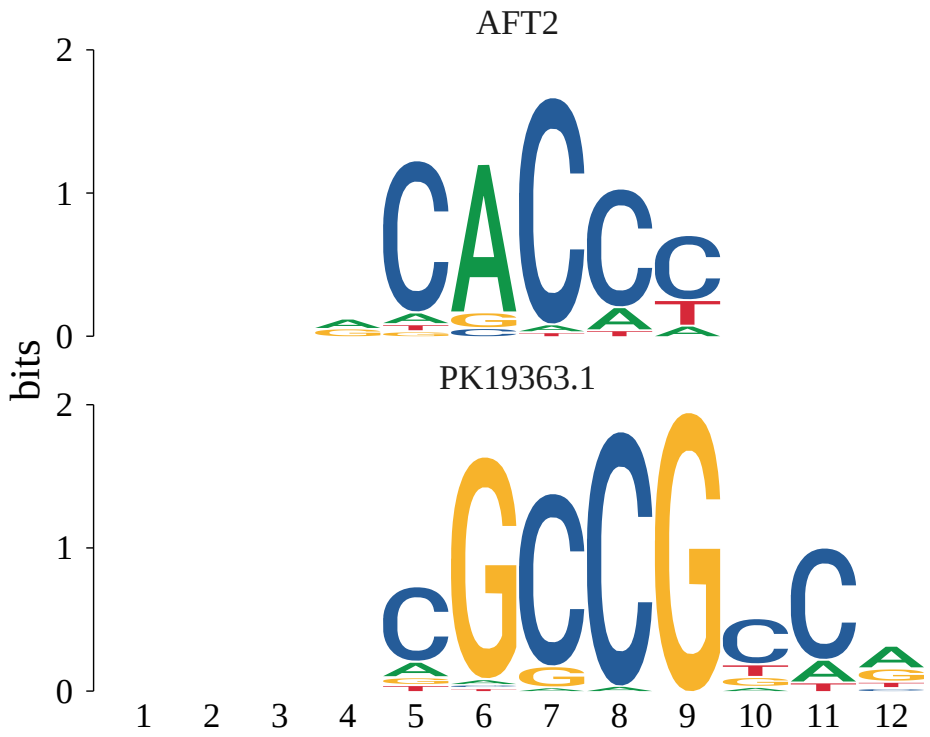


Figure 2: Example scores from comparing two motifs

Table 1: Comparing two motifs with various settings

type	method	default	normalised	checkIC
similarity	PCC	0.5145697	0.3087418	0.9356122
similarity	WPCC	0.5145697	0.3087418	0.9356122
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` argument 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 LOB [duplicated #1051]      1845  0.987357  -13.3725
#> 5       ORA59         1                  ERF15       618  0.977213  -12.9254
#> 6       ORA59         1 ERF2 [duplicated #769]      1563  0.973871  -12.7804
#>           Pval       Eval
#>      <numeric>  <numeric>
#> 1 1.31042e-06 0.00514210
#> 2 1.33754e-06 0.00524852
#> 3 1.55744e-06 0.00611138
#> 4 1.55744e-06 0.00611138
#> 5 2.43548e-06 0.00955683
#> 6 2.81553e-06 0.01104816
```

P-values are made possible by estimating distribution (usually the best fitting distribution for motif comparisons) parameters from randomised 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.

2.4 A faster alternative: `compare_motifs_lite()`

For DNA or RNA motif comparison that does not need the alternative similarity metrics (EUCL, KL, ALLR, and so on), the score-aggregation strategies, the IC-based filters, multifreq scoring, or the JASPAR-derived `db.scores` lookup-table P-value path, the lighter `compare_motifs_lite()` is generally much faster than `compare_motifs()`, and it parallelises rather better across queries. It uses the per-column Pearson correlation as its only similarity metric, computes p-values from a TOMTOM-style empirical or parametric null PMF (with both Bonferroni and Benjamini-Hochberg adjustment), and returns either a square matrix or a long-format `data.frame`.

The function has eight formals (`motifs`, `compare.to`, `qvalue`, `min.overlap`, `RC`, `null`, `bkg`, `matrix.out`,

`nthreads`); see `?compare_motifs_lite` for the details. There are two null-PMF modes:

- **`null = "empirical"`** (default): per-query column, score against every target-database column, histogram into 51 PCC bins, convolve to overlap length L. Cost per query scales with database size; 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 $\alpha = 4 \times$ `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.

```
## Score matrix (default: matrix.out = "score" returns mean PCC,
## always in [-1, 1])
m <- compare_motifs_lite(motifs)

## Significance matrix (p-values or BH q-values across the
## query-vs-database axis)
mp <- compare_motifs_lite(motifs, matrix.out = "pvalue")
mq <- compare_motifs_lite(motifs, matrix.out = "qvalue")

## Long-format: motif 1 against the rest, q-value <= 0.05
hits <- compare_motifs_lite(motifs, compare.to = 1, qvalue = 0.05)
```

When `compare_motifs()` is called with arguments that map cleanly onto `compare_motifs_lite()`'s feature set (PCC + sum strategy, the default IC filters, no multifreq, no HTML report, no `db.scores`, and a DNA/RNA alphabet), it emits a one-line hint pointing you at the faster function. (Silence it with `options(universalmotif.suggest.compare_motifs_lite = FALSE)`.)

Speed comparison vs `compare_motifs()` and `yamtk cmp`

The table below records the wall-clock time for the three functions on a single shared fixture: the first 50 (or 200) motifs of HOCOMOCOv11, compared against themselves, with RC = TRUE, min.overlap = 5, a uniform background, and the empirical null PMF. The `compare_motifs_lite()` numbers are shown for both `matrix.out = "score"` (which skips the p-value path entirely, since the best alignment is chosen by score rather than by p-value) and `matrix.out = "pvalue"` (the full significance computation, which is equivalent work to `yamtk cmp`). Each cell shows `nthreads = 1 / nthreads = 4`. The values are baked in here rather than recomputed at vignette build time. These benchmarks were run on an MBP M5 under R 4.4.1.

Workload	yamtk[1]	`compare_motifs()`	`compare_motifs_lite()` score[2]	`compare_motifs_lite()` pvalue
50 × 50 (2 500 pairs)	0.028 / 0.012	0.598 / 0.194	0.014 / 0.004	0.285 / 0.082
200 × 200 (40 000 pairs)	0.114 / 0.038	9.537 / 2.590	0.224 / 0.064	1.673 / 0.470

The `score` / `pvalue` labels indicate the value of the `matrix.out` argument.

[1] `yamtk cmp` is a dedicated C CLI tool; threading is by query motif. The `compare_motifs_lite()` algorithm is a port of yamtk's.

[2] Score-only matrix mode skips the per-pair p-value lookup, which otherwise dominates runtime on dense fixtures. This is safe because the best alignment is selected by score, not p-value; p-values are computed only as a post-hoc significance check on the already-chosen alignment.

To summarise:

- `yamtk cmp` is the fastest of the three at every scale; the `compare_motifs_lite()` port stays within ~2× at 4 threads for score-only output and within ~12× for full p-value output.

- `compare_motifs_lite()` is 5-40× faster than `compare_motifs()` at 4 threads on the same set, with the larger speedups in score-only mode. `compare_motifs()`'s p-value path uses pre-computed JASPAR score-distribution tables (`JASPAR2018_CORE_DBSCORES`) and a different null model, so the numbers are not directly comparable as "significance" but are comparable as "wall-clock to a similarity matrix".
- Threading scales 3-3.5× at 4 threads in all three universalmotif modes.
- For new code that does not need any of `compare_motifs()`'s advanced features, try `compare_motifs_lite()`.

3 Merging motifs

The `universalmotif` package offers two ways to collapse similar motifs into consensus motifs: `merge_motifs()` / `merge_similar()`, and `merge_motifs_lite()` / `merge_similar_lite()` (the latter pair are newer, built on `compare_motifs_lite()`, and use significance-based clustering).

3.1 A faster alternative: `merge_motifs_lite()` / `merge_similar_lite()`

For simple DNA/RNA motif merging tasks:

- `merge_motifs_lite(motifs)` aligns every motif against the highest-IC anchor via `compare_motifs_lite()` and column-averages them in a single shared frame.
- `merge_similar_lite(motifs, qvalue)` clusters motifs by statistical significance: two motifs are linked if their pairwise q-value meets the chosen `qvalue` cutoff. Clustering is by connected components on the resulting graph (deterministic, no linkage-method to choose). Each component is collapsed via `merge_motifs_lite()`.

```
library(universalmotif)
m1 <- create_motif("TTGACATA", name = "a")
m2 <- create_motif("CTTGACAT", name = "b")
m3 <- create_motif("TGACATAT", name = "c")
m4 <- create_motif("GGGCCCCC", name = "unrelated")

## Single merged consensus across 3 related motifs:
merge_motifs_lite(list(m1, m2, m3))
#>
#>        Motif name:   merged_a+b+c
#>          Alphabet:   DNA
#>              Type:   PPM
#>           Strands:   +-
#>          Total IC:   20
#>       Pseudocount:   0
#>         Consensus:   CTTGACATAT
#>      Target sites:   3
#>
#>   C T T G A C A T A T
#> A 0 0 0 0 1 0 1 0 1 0
#> C 1 0 0 0 0 1 0 0 0 0
#> G 0 0 0 1 0 0 0 0 0 0
#> T 0 1 1 0 0 0 0 1 0 1

## Cluster + merge: returns a shorter list, with similar motifs collapsed.
merge_similar_lite(list(m1, m2, m3, m4), qvalue = 0.05)
#> [[1]]
#>
#>        Motif name:   merged_a+b+c
#>          Alphabet:   DNA
```

```
#>              Type:   PPM
#>           Strands:   +-
#>          Total IC:   20
#>       Pseudocount:   0
#>         Consensus:   CTTGACATAT
#>      Target sites:   3
#>
#>   C T T G A C A T A T
#> A 0 0 0 0 1 0 1 0 1 0
#> C 1 0 0 0 0 1 0 0 0 0
#> G 0 0 0 1 0 0 0 0 0 0
#> T 0 1 1 0 0 0 0 1 0 1
#>
#> [[2]]
#>
#>        Motif name:   unrelated
#>          Alphabet:   DNA
#>              Type:   PPM
#>           Strands:   +-
#>          Total IC:   16
#>       Pseudocount:   0
#>         Consensus:   GGGCCCCC
#>
#>   G G G C C C C C
#> A 0 0 0 0 0 0 0 0
#> C 0 0 0 1 1 1 1 1
#> G 1 1 1 0 0 0 0 0
#> T 0 0 0 0 0 0 0 0

## Or just the cluster assignments (no merging):
merge_similar_lite(list(m1, m2, m3, m4), qvalue = 0.05, return.clusters = TRUE)
#>            motif      name consensus alphabet strand icscore type pseudocount
#> 1 *      <mot:a>         a  TTGACATA      DNA     +-      16  PPM           0
#> 2 *      <mot:b>         b  CTTGACAT      DNA     +-      16  PPM           0
#> 3 *      <mot:c>         c  TGACATAT      DNA     +-      16  PPM           0
#> 4 * <mot:unre..> unrelated  GGGCCCCC      DNA     +-      16  PPM           0
#>             bkg motif.i cluster
#> 1 0.25, 0.....       1       1
#> 2 0.25, 0.....       2       1
#> 3 0.25, 0.....       3       1
#> 4 0.25, 0.....       4       2
#>
#> [Hidden empty columns: altname, family, organism, nsites, bkgsites, pval,
#>   qval, eval.]
#> [Rows marked with * are changed. Run update_motifs() or to_list() to apply
#>   changes.]
```

When a `merge_motifs()` or `merge_similar()` call uses only features that are also available to `merge_motifs_lite()` / `merge_similar_lite()`, a one-line hint is emitted pointing you at the leaner function. (Silence it with `options(universalmotif.suggest.merge_motifs_lite = FALSE)` and `options(universalmotif.suggest.merge_similar_lite = FALSE)`.)

4 Motif trees with ggtree

4.1 Using `motif_tree()`

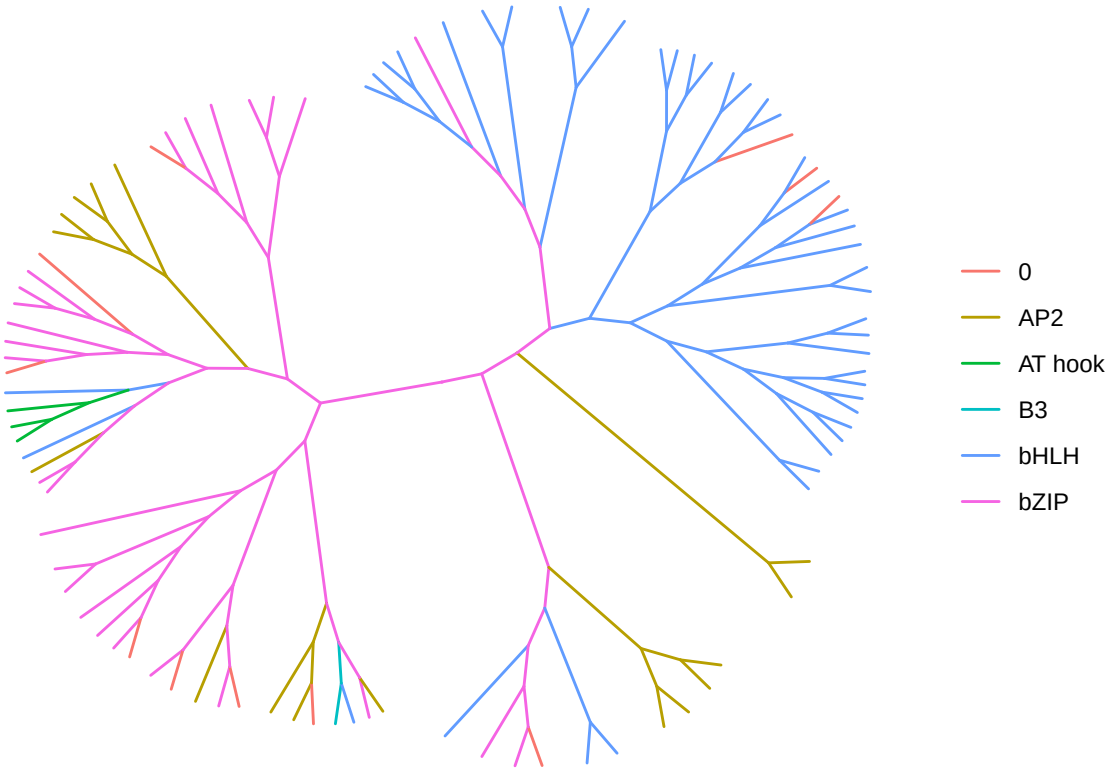
Additionally, this package introduces the `motif_tree()` function for generating basic tree-like diagrams for comparing motifs. This allows for a visual result from `compare_motifs()`. All options from `compare_motifs()` are available in `motif_tree()`. This function uses the `ggtree` package and outputs 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. There is also a new `motif_tree_lite()`, which is based on `compare_motifs_lite()`.

```
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
```



4.2 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.

```r
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
```

```r
# If we wish, we can add 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))
```

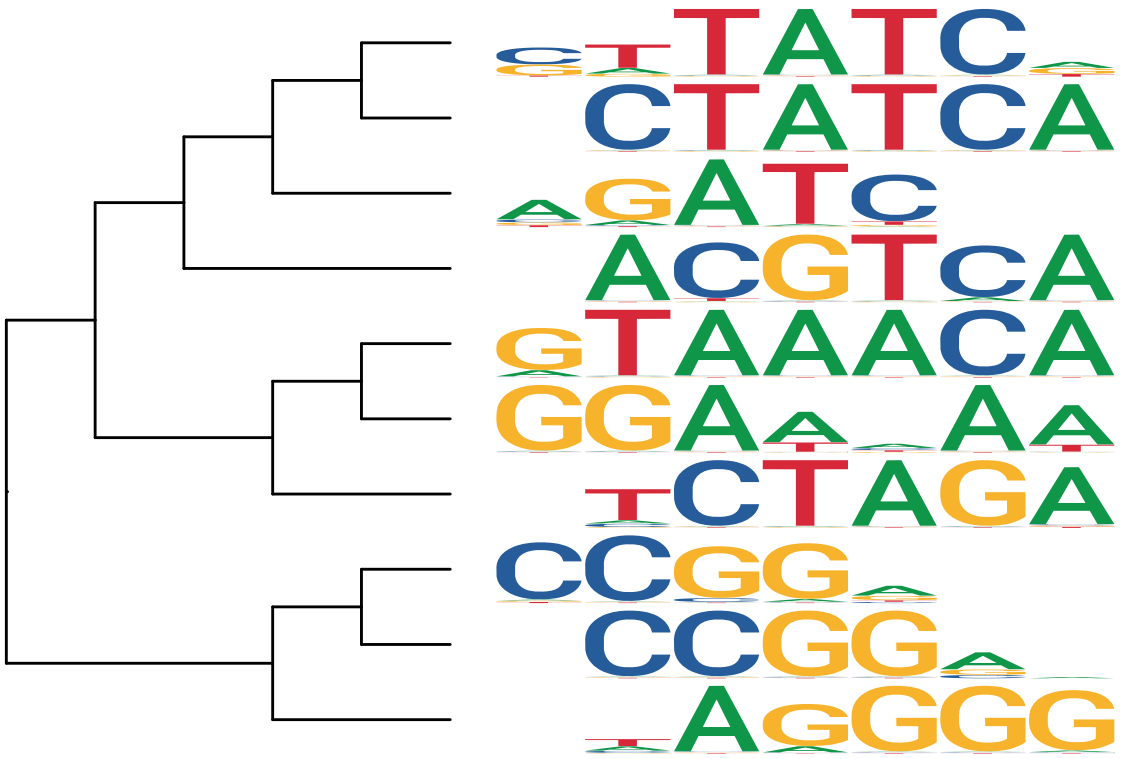
4.3 Plotting motifs alongside trees

`motif_tree_lite()` draws a sequence logo at every tip by default (`tiplogo = TRUE`), so the motifs themselves serve as the tip labels. The logos are aligned to a shared column frame (via `view_motifs_lite()`) so that homologous positions line up across tips.

```r
library(universalmotif)
library(MotifDb)

motifs <- convert_motifs(MotifDb[1:10])

## Tip logos are on by default; the default layout is now "rectangular".
motif_tree_lite(motifs, linecol = "none")
```



Use `tiplogo = FALSE` to turn the logos off, or `tiplogo.align = FALSE` to draw each tip logo independently instead of aligned to a common frame. Tip logos are only drawn on the `"rectangular"` and `"slanted"` layouts. The machinery is also exposed directly as the `geom_motif()` and `geom_logo()` `ggplot2` layers (and the `annotate_motif()` / `annotate_logo()` annotation helpers), so logos can be embedded in any `ggplot` construction.

5 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:   1.84
#>       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.

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

Dynamic programming calculates the probability of each score without listing every possible sequence. FIMO uses this approach (Grant, Bailey, and Noble 2011), and Gupta et al. (2007) describes it in the context of motif comparison.

The algorithm works through the motif one position at a time. At each position, it updates the possible total scores and their probabilities, using the background frequencies of the letters. Once this score distribution is built, it can answer many score or P-value queries without repeating the calculation.

Internally, each PWM entry is multiplied by 1000 and its decimal part is dropped (truncation towards zero). The resulting integers are then added. This puts scores on a grid with steps of 0.001. When you supply a score, it is rounded up to the next grid point, or left unchanged if already on the grid. The returned P-value is the probability of getting that score or a higher one: `Pr(score >= query)`.

When you supply a P-value cutoff instead, the function returns the lowest threshold on this grid for which the probability of a match at or above the threshold is no greater than the cutoff. The threshold need not be a score that a sequence can actually achieve. If even the highest-scoring match fails the cutoff, the function returns `Inf` and scanning reports no hits.

These calculations use the truncated PWM scores, not the original scores at full precision. Truncation can give different sequences the same score, or change the order of very similar scores. This means that small score changes can sometimes cause much larger P-value changes. This effect comes from the score approximation, rather than small numerical errors when adding probabilities.

Dynamic programming also needs finite PWM entries. If zero probabilities produce `-Inf` scores, use pseudocounts to make the scores finite, or consider the exhaustive method with `allow.nonfinite`.

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
```

Converting a score to a P-value and back may give a different score, because several thresholds can select the same matches. What matters is that the probability of a match at or above the returned threshold does not exceed the requested cutoff.

5.1.1 Which background does a P-value use?

Background frequencies affect both the PWM scores and the chance of seeing each sequence. With the dynamic method and ordinary mononucleotide motifs (`use.freq = 1`), supplying `bkg.probs` changes both: the function recalculates the PWM using the new background, then uses that background to calculate probabilities. It does not leave the PWM scores unchanged. The same sequence can therefore have a different score under a different background.

Both scanners use the background stored in the motif; they do not automatically estimate it from the sequences being scanned. To use a different background, first convert the motif to a PPM using its original background. Then set the new background and convert to a PWM:

```
probabilities <- matrix(rep(c(.5, .25, .125, .125), 4), nrow = 4,
                        dimnames = list(c("A", "C", "G", "T"), NULL))
original <- create_motif(probabilities, type = "PPM", pseudocount = 0)
rebased <- convert_type(original, "PPM")
rebased["bkg"] <- c(A = .1, C = .4, G = .4, T = .1)
rebased <- convert_type(rebased, "PWM")
#> Note: motif [motif] has an empty nsites slot, using 100.
motif_pvalue(rebased, pvalue = .01)
#> [1] 3.287
```

Changing the `bkg` slot of a PWM alone does not update its scores: the PPM-to-PWM conversion is needed too. The example above uses `use.freq = 1`. For higher-order motifs, `use.freq` selects both the higher-order matrix and its corresponding background probabilities.

The background sequences used in an enrichment test serve a different purpose: they are the control group. Matching their composition to the target sequences does not change the motif's PWM. Choose both the motif background and the control sequences to suit the question, and check how the test behaves on data with no enrichment. Composition matching alone does not guarantee accurate P-values or false discovery rates.

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   18.731
```

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] 9.277
```

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:   15.94
#>       Pseudocount:   0
#>
#>   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#> E 0.05 0.65 0.38 0.02 0.24 0.86 0.00 0.00 0.00  0.00
#> Q 0.00 0.00 0.09 0.00 0.08 0.00 0.01 0.00 0.61  0.06
#> R 0.84 0.00 0.47 0.19 0.54 0.02 0.00 0.01 0.00  0.00
#> T 0.03 0.20 0.00 0.49 0.05 0.09 0.98 0.00 0.39  0.05
#> W 0.08 0.13 0.00 0.00 0.01 0.02 0.00 0.00 0.00  0.86
#> Y 0.00 0.02 0.05 0.30 0.10 0.00 0.01 0.99 0.00  0.03
motif_pvalue(m, pvalue = c(1, 0.1, 0.001, 0.0001, 0.00001))
#> [1] -226.577  -31.292    1.347    8.704   13.672
```

5.2 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*. It is important to note however that this is still a computationally intensive 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.

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

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

```
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 trade-off 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	0.2101195	0.01352037	0.0000000000
4	0.8516271	0.4945960	0.1576815	0.02164814	0.0007409123
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.

5.3 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 larger 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 is approximated 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	1.3276	1.3242
1e-02	3.596	1.000309e-02	9.991646e-03	9.991646e-03	3.6195	3.5936
1e-03	4.858	1.000404e-03	9.984970e-04	9.984970e-04	4.7826	4.8718
1e-04	5.647	1.001358e-04	9.727478e-05	9.727478e-05	5.6090	5.6532
1e-05	6.182	1.049042e-05	9.536743e-06	9.536743e-06	5.7336	6.1944

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	-2.0055	-2.0017
-1	3.354645e-01	3.353453e-01	3.353453e-01	-1.000	-1.0105	-1.0026
0	2.185555e-01	2.184534e-01	2.184534e-01	0.000	-0.0017	0.0026
1	1.244183e-01	1.243525e-01	1.243525e-01	1.000	0.9869	0.9972
2	5.911160e-02	5.907822e-02	5.907822e-02	2.000	2.0289	1.9979
3	2.163410e-02	2.160931e-02	2.160931e-02	3.000	2.9863	3.0003
4	5.360603e-03	5.347252e-03	5.347252e-03	4.000	3.9439	4.0092
5	7.162094e-04	7.152557e-04	7.152557e-04	5.000	4.9910	4.9882
6	2.193451e-05	2.193451e-05	2.193451e-05	6.057	5.6578	5.9907

Both algorithms use truncated PWM scores rather than the original scores at full precision. The exhaustive method also uses an approximation when it splits a long motif into shorter pieces. Agreement between the methods on short motifs does not mean that either always gives exact probabilities for the original, full-precision PWM.

Session info

```
#> R version 4.6.1 (2026-06-24)
#> Platform: x86_64-pc-linux-gnu
#> Running under: Ubuntu 24.04.5 LTS
#>
#> Matrix products: default
#> BLAS:   /home/biocbuild/bbs-3.24-bioc/R/lib/libRblas.so
#> LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.12.0  LAPACK version 3.12.0
#>
#> locale:
#>  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
#>  [3] LC_TIME=en_GB              LC_COLLATE=C
#>  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
#>  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
#>  [9] LC_ADDRESS=C               LC_TELEPHONE=C
#> [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
#>
```

```
#> time zone: America/New_York
#> tzcode source: system (glibc)
#>
#> attached base packages:
#> [1] stats4    stats     graphics  grDevices utils     datasets  methods
#> [8] base
#>
#> other attached packages:
#>  [1] dplyr_1.2.1
#>  [2] ggtree_4.3.0
#>  [3] ggplot2_4.0.3
#>  [4] MotifDb_1.55.0
#>  [5] BSgenome.Athaliana.TAIR.TAIR9_1.3.1000
#>  [6] BSgenome_1.81.1
#>  [7] BiocIO_1.23.3
#>  [8] rtracklayer_1.73.0
#>  [9] GenomeInfoDb_1.49.1
#> [10] GenomicRanges_1.65.4
#> [11] Biostrings_2.81.9
#> [12] Seqinfo_1.3.2
#> [13] XVector_0.53.0
#> [14] IRanges_2.47.5
#> [15] S4Vectors_0.51.10
#> [16] BiocGenerics_0.59.12
#> [17] generics_0.1.4
#> [18] universalmotif_1.31.59
#>
#> loaded via a namespace (and not attached):
#>  [1] ggiraph_0.9.6            tidyselect_1.2.1        farver_2.1.2
#>  [4] S7_0.2.2                 bitops_1.1-0            fastmap_1.2.0
#>  [7] RCurl_1.98-1.20          lazyeval_0.2.3          fontquiver_0.2.1
#> [10] GenomicAlignments_1.49.2 XML_3.99-0.25           digest_0.6.39
#> [13] lifecycle_1.0.5          tidytree_0.4.8          magrittr_2.0.5
#> [16] compiler_4.6.1           rlang_1.3.0             tools_4.6.1
#> [19] yaml_2.3.12              data.table_1.18.6.1     knitr_1.52
#> [22] htmlwidgets_1.6.4        labeling_0.4.3          curl_8.1.0
#> [25] splitstackshape_1.4.8.1  RColorBrewer_1.1-3      aplot_0.3.2
#> [28] BiocParallel_1.47.0      withr_3.0.3             purrr_1.2.2
#> [31] grid_4.6.1               gdtools_0.5.2           scales_1.4.0
#> [34] MASS_7.3-66              dichromat_2.0-1         tinytex_0.61
#> [37] cli_3.6.6                rmarkdown_2.32          crayon_1.5.3
#> [40] treeio_1.37.1            otel_0.2.0              httr_1.4.9
#> [43] rjson_0.2.23             BiocBaseUtils_1.15.1    ape_5.8-1
#> [46] parallel_4.6.1           ggplotify_0.1.4         restfulr_0.0.17
#> [49] matrixStats_1.5.0        vctrs_0.7.3             yulab.utils_0.2.5
#> [52] fontBitstreamVera_0.1.1  jsonlite_2.0.0          bookdown_0.48
#> [55] gridGraphics_0.5-1       patchwork_1.3.2         systemfonts_1.3.2
#> [58] tidyr_1.3.2              glue_1.8.1              codetools_0.2-20
#> [61] gtable_0.3.6             UCSC.utils_1.9.0        tibble_3.3.1
#> [64] pillar_1.11.1            rappdirs_0.3.4          htmltools_0.5.9
#> [67] R6_2.6.1                 evaluate_1.0.5          lattice_0.23-1
#> [70] Rsamtools_2.29.0         cigarillo_1.3.1         fontLiberation_0.1.0
#> [73] ggfun_0.2.2              Rcpp_1.1.2              nlme_3.1-171
#> [76] xfun_0.62                MatrixGenerics_1.25.0   fs_2.1.0
```

```
#> [79] 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: 1017–18.

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: W104–9.

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 (suppl_2): W438–41.

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.