Advanced usage: motif and sequence analysis

Benjamin Jean-Marie Tremblay[*1]

[1]University of Waterloo, Waterloo, Canada

*b2tremblay@uwaterloo.ca

9 May 2019

Contents

1 Introduction

This vignette details the *universalmotif* implementation of higher-order motifs (`multifreq`), motif enrichment analyses, motif P-values, advanced usage related to motif comparison, drawing motif trees, and *de novo* motif searches with MEME. 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 Higher-order motifs

Though PCM, PPM, PWM, and ICM type motifs are still widely used today, a few 'next generation' motif formats have been proposed. These wish to add another layer of information to motifs: positional interdependence. To illustrate this, consider the following sequences:

Table 1: Example sequences

#	Sequence
1	CAAAACC
2	CAAAACC
3	CAAAACC
4	CTTTTCC
5	CTTTTCC
6	CTTTTCC

This becomes the following PPM:

Table 2: Position Probability Matrix

Position	1	2	3	4	5	6	7
A	0.0	0.5	0.5	0.5	0.5	0.0	0.0
C	1.0	0.0	0.0	0.0	0.0	1.0	1.0
G	0.0	0.0	0.0	0.0	0.0	0.0	0.0
T	0.0	0.5	0.5	0.5	0.5	0.0	0.0

Based on the PPM representation, all three of CAAAACC, CTTTTCC, and CTATACC are equally likely. Though looking at the starting sequences, should CTATACC really be considered so? For transcription factor binding sites, some would say the answer is no. By incorporating this type of information into the motif, this can allow for increased accuracy in motif searching. A few implementations of this include: TFFM by Mathelier and Wasserman (2013), BaMM by Siebert and Soding (2016), and KSM by Guo et al. (2018).

The *universalmotif* package implements its' own, rather simplified, version of this concept. Plainly, the standard PPM has been extended to include `k`-letter frequencies, with `k` being any number higher than 1. For example, the 2-letter version of the table 2 motif would be:

Table 3: 2-letter probability matrix

Position	1	2	3	4	5	6
AA	0.0	0.5	0.5	0.5	0.0	0.0
AC	0.0	0.0	0.0	0.0	0.5	0.0
AG	0.0	0.0	0.0	0.0	0.0	0.0
AT	0.0	0.0	0.0	0.0	0.0	0.0
CA	0.5	0.0	0.0	0.0	0.0	0.0
CC	0.0	0.0	0.0	0.0	0.0	1.0
CG	0.0	0.0	0.0	0.0	0.0	0.0
CT	0.5	0.0	0.0	0.0	0.0	0.0
GA	0.0	0.0	0.0	0.0	0.0	0.0
GC	0.0	0.0	0.0	0.0	0.0	0.0
GG	0.0	0.0	0.0	0.0	0.0	0.0
GT	0.0	0.0	0.0	0.0	0.0	0.0
TA	0.0	0.0	0.0	0.0	0.0	0.0
TC	0.0	0.0	0.0	0.0	0.5	0.0
TG	0.0	0.0	0.0	0.0	0.0	0.0
TT	0.0	0.5	0.5	0.5	0.0	0.0

This format shows the probability of each letter combined with the probability of the letter in the next position. The seventh column has been dropped, since it is not needed; the information in the sixth column is sufficient, and there is no eighth position to draw 2-letter probabilities from. Now, the probability of getting CTATACC is no longer equal to CTTTTCC and CAAAACC. This information is kept in the `multifreq` slot of `universalmotif` class motifs. To add this information, use the `add_multifreq` function.

```
library(universalmotif)

motif <- create_motif("CWWWWCC", nsites = 6)
sequences <- DNAStringSet(rep(c("CAAAACC", "CTTTTCC"), 3))
motif.k2 <- add_multifreq(motif, sequences, add.k = 2)

## Alternatively:
# motif.k2 <- create_motif(sequences, add.multifreq = 2)

motif.k2
#>
#>        Motif name:   motif
#>          Alphabet:   DNA
#>              Type:   PPM
#>           Strands:   +-
#>          Total IC:   10
#>         Consensus:   CWWWWCC
#>      Target sites:   6
#>    k-letter freqs:   2
#>
#>   C   W   W   W   W C C
#> A 0 0.5 0.5 0.5 0.5 0 0
#> C 1 0.0 0.0 0.0 0.0 1 1
#> G 0 0.0 0.0 0.0 0.0 0 0
```

```
#> T 0 0.5 0.5 0.5 0.5 0 0
```

This information is most useful with functions such as `scan_sequences()` and `enrich_motifs()`. Though other tools in the *universalmotif* can work with `multifreq` motifs (such as `motif_pvalue()`, `compare_motifs()`), keep in mind they are not as well supported as regular motifs (getting P-values from `multifreq` motifs is exponentially slower, and P-values from using `compare_motifs()` for `multifreq` motifs are not available by default). See the sequences vignette for using `scan_sequences()` with the `multifreq` slot.

3 Enrichment analyses

The *universalmotif* package offers the ability to search for enriched motif sites in a set of sequences via `enrich_motifs()`. There is little complexity to this, as it simply runs `scan_sequences()` twice; once on a set of target sequences, and once on a set of background sequences. After which the results between the two sequences are collated and run through enrichment tests. The background sequences can be given explicitly, or else `enrich_motifs()` will create background sequences on its own by using `shuffle_sequences()` on the target sequences.

Let us consider the following basic example:

```
library(universalmotif)
data(ArabidopsisMotif)
data(ArabidopsisPromoters)

enrich_motifs(ArabidopsisMotif, ArabidopsisPromoters, shuffle.k = 3,
              threshold = 0.001, verbose = 0, progress = FALSE, RC = TRUE)
#>             motif total.seq.hits num.seqs.hit num.seqs.total total.bkg.hits
#> 1 YTTTYTTTTTYTTTY           197          41            50            45
#>   num.bkg.hit num.bkg.total   Pval.hits   Qval.hits    Eval.hits
#> 1          21            50 3.41482e-24 3.41482e-24 6.829641e-24
```

Here we can see that the motif is significantly enriched in the target sequences. Looking for closely at the results, the motif was found 197 times in 41 of the 50 sequences; whereas it was found 37 times in 14 of the 50 background sequences. The `Pval.hits` was calculated by calling `fisher.test` from the *stats* package. Another type of test is available which looks for positional enrichment, i.e. whether one area in the target sequences is favoured.

```
library(universalmotif)
data(ArabidopsisMotif)
data(ArabidopsisPromoters)

enrich_motifs(ArabidopsisMotif, ArabidopsisPromoters, shuffle.k = 3,
              search.mode = "positional", verbose = 0, progress = FALSE,
              RC = TRUE)
#>  ! No enriched motifs
```

In this case however no such bias exists in the target sequences. There are a number of tests available for positional enrichment; see `?enrich_motifs`.

One final point: always keep in mind the `threshold` parameter, as this will ultimately decide the number of hits found. (A bad threshold can lead to a false negative.) See the sequences vignette for a discussion about using P-values to determine the threshold of `scan_sequences()`. Motif P-values are discussed in more details in the next section of this document.

4 Motif P-values

Motif P-values are not usually discussed outside of the bioinformatics literature, but are actually quite a challenging topic. For 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
#>         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,
                      progress = FALSE, threshold = 0.8,
                      threshold.type = "logodds")
head(res)
#>   motif  sequence start stop score max.score score.pct      match strand
#> 1 motif AT4G28150     1   10 2.195  6.402864    80.975 CATACAAGTA      +
#> 2 motif AT4G28150    87   96 3.896  6.402864    89.480 CTCTTTATTC      +
#> 3 motif AT4G28150   154  163 3.475  6.402864    87.375 CTTTTGATAA      +
#> 4 motif AT4G28150   165  174 3.396  6.402864    86.980 GTTTTAATAA      +
#> 5 motif AT4G28150   187  196 2.452  6.402864    82.260 GACTAGCGGC      +
#> 6 motif AT4G28150   235  244 2.951  6.402864    84.755 CATAAAAGTC      +
```

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

```
## The second match was CTCTTTATTC, with a score of 3.896 (max possible = 6.403)

library(Biostrings)

bkg <- colMeans(oligonucleotideFrequency(ArabidopsisPromoters, 1,
                                         as.prob = TRUE))
bkg
#>       A       C       G       T
#> 0.34768 0.16162 0.15166 0.33904
```

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

```
hit.prob <- bkg["A"]^1 * bkg["C"]^3 * bkg["G"]^0 * bkg["T"]^6
hit.prob <- unname(hit.prob)
hit.prob
#> [1] 2.229311e-06
```

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 3.896, and then summing these. This final sum then represents the probability of finding a match which scores at least 3.896. One way is to list all possible sequence combinations, then filtering based on score; however this "brute force" approach is unreasonable but for the smallest of motifs.

A few algorithms have been proposed to make this 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.

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

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*.

Now, let us return to our original example:

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

motif	match	pval
motif	CTCTTTATTC	0.003408644
motif	CTTTTGATAA	0.006929586
motif	GTTTTAATAA	0.007833046
motif	CATAAAAGTC	0.014524545

motif	match	pval
motif	GACTAGCGGC	0.026110815
motif	CATACAAGTA	0.034175189

The default `k` in `motif_pvalue()` is 6. 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], progress = FALSE,
                                  bkg.probs = bkg)
  }
}

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

k	score.minus6	score.minus3	score.0	score.3	score.6
2	0.9275584	0.6679457	0.2453828	0.007187964	0.0000e+00
4	0.8835751	0.5738532	0.1750234	0.010256733	0.0000e+00
6	0.8841629	0.5824325	0.1873000	0.013655532	5.3155e-06
8	0.8841629	0.5824325	0.1873000	0.013655532	5.3155e-06
10	0.8842381	0.5826067	0.1874028	0.013669345	5.3155e-06

For this particular motif, while the approximation worsens slightly as `k` decreases, it is still quite reasonable down to `k = 6`. Usually, you should only have to worry about `k` for longer motifs (such as those typically generated by MEME), where the number of sub-motifs increases.

The *universalmotif* also allows for calculating motifs 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. Unlike the P-value calculation however, a uniform background is assumed. This is because approximate scores are calculated by assuming the distribution of possible scores follows a normal distribution. See the following code for a comparison of exact and approximate score calculations.

Starting from a set of P-values:

```
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, progress = FALSE,
                       bkg.probs = bkg, k = 10)

scores.approx6 <- motif_pvalue(motif, pvalue = pvals, progress = FALSE,
                               bkg.probs = bkg, k = 6)
scores.approx8 <- motif_pvalue(motif, pvalue = pvals, progress = FALSE,
                               bkg.probs = bkg, k = 8)

pvals.exact <- motif_pvalue(motif, score = scores, progress = FALSE,
                            bkg.probs = bkg, k = 10)

pvals.approx6 <- motif_pvalue(motif, score = scores, progress = FALSE,
                              bkg.probs = bkg, k = 6)
pvals.approx8 <- motif_pvalue(motif, score = scores, progress = FALSE,
                              bkg.probs = bkg, k = 8)

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	1.324000	1.000299e-01	9.995747e-02	9.995747e-02	1.402687	1.415141
1e-02	3.596000	1.000309e-02	9.991646e-03	9.991646e-03	4.565841	4.588447
1e-03	4.857425	1.005173e-03	1.000404e-03	1.000404e-03	6.531000	6.531000
1e-04	5.645285	1.010895e-04	1.001358e-04	1.001358e-04	6.531000	6.531000
1e-05	6.145569	1.049042e-05	1.049042e-05	1.049042e-05	6.531000	6.531000

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, progress = FALSE,
                      bkg.probs = bkg, k = 10)

scores.exact <- motif_pvalue(motif, pvalue = pvals, progress = FALSE,
                             bkg.probs = bkg, k = 10)

scores.approx6 <- motif_pvalue(motif, pvalue = pvals, progress = FALSE,
                               bkg.probs = bkg, k = 6)
scores.approx8 <- motif_pvalue(motif, pvalue = pvals, progress = FALSE,
                               bkg.probs = bkg, k = 8)

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

```
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	-2.0005372953	-2.1938060	-2.1928963
-1	3.354645e-01	3.353453e-01	3.353453e-01	-1.0006645355	-1.1909325	-1.1868038
0	2.185555e-01	2.184534e-01	2.184534e-01	-0.0007814445	-0.1246126	-0.1170613
1	1.244183e-01	1.243525e-01	1.243525e-01	0.9991244183	1.0140385	1.0252444
2	5.911160e-02	5.907822e-02	5.907822e-02	1.9990591116	2.2525868	2.2677680
3	2.163410e-02	2.160931e-02	2.160931e-02	2.9990216341	3.6417289	3.6613688
4	5.360603e-03	5.347252e-03	5.347252e-03	3.9990053606	5.2479708	5.2727663
5	7.162094e-04	7.152557e-04	7.152557e-04	4.9970021486	6.5310000	6.5310000
6	2.193451e-05	2.193451e-05	2.193451e-05	5.9940013819	6.5310000	6.5310000

As you can see, results from exact calculations are not *quite* exact. This is due to the fact that the distributions of P-values and scores are assumed to be continuous, when in reality they are discrete. As for results from approximate calculations: they are fairly close to exact values for larger P-values, but fall off for small P-values ($P < 0.001$).

5 Advanced motif comparison

Here I will explore the effects of some of the options available in `compare_motifs()`. See the relevant section in the basic motif manipulation vignette for an introduction to using `compare_motifs()`. Let us begin by comparing the available methods:

```
library(universalmotif)
library(MotifDb)

motifs <- convert_motifs(MotifDb)
motifs <- filter_motifs(motifs, altname = c("M0003_1.02", "M0004_1.02"))
summarise_motifs(motifs)
#>        name    altname family    organism  consensus alphabet strand
#> 1      AFT2 M0003_1.02    AFT Scerevisiae NHNNCACCYN      DNA      +-
#> 2 PK19363.1 M0004_1.02    AP2     Csativa   CGCCGCCR      DNA      +-
#>    icscore
#> 1 5.809302
#> 2 8.926117
try_all <- function(motifs, ...) {
  scores <- numeric(8)
  methods <- c("MPCC", "PCC", "MEUCL", "EUCL", "MSW", "SW", "MKL", "KL")
  for (i in 1:8) {
    scores[i] <- compare_motifs(motifs, method = methods[i],
```

```
                                progress = FALSE, ...)[1, 2]
  }
  res <- data.frame(method = c("MPCC", "PCC", "MEUCL", "EUCL",
                               "MSW", "SW", "MKL", "KL"),
                    score = scores,
                    max = c(1, NA, sqrt(2), NA, 2, NA, NA, NA),
                    type = c("similarity", "similarity", "distance",
                             "distance", "similarity", "similarity",
                             "distance", "distance"))
  knitr::kable(res, format = "markdown")
}
try_all(motifs)
```

method	score	max	type
MPCC	0.5145697	1.000000	similarity
PCC	18.5245085	NA	similarity
MEUCL	0.3881919	1.414214	distance
EUCL	2.3291516	NA	distance
MSW	1.5579529	2.000000	similarity
SW	12.0570984	NA	similarity
MKL	0.9314823	NA	distance
KL	5.5888940	NA	distance

From this you can get a sense of how the different methods perform. See the function documentation at `?compare_motifs` for details on the method implementations. For now, let us explore how changing the other parameters affects the scores. First, we can control whether to allow the reverse complements to be compared as well:

```
try_all(motifs, tryRC = FALSE)
```

method	score	max	type
MPCC	0.5145697	1.000000	similarity
PCC	18.5245085	NA	similarity
MEUCL	0.3881919	1.414214	distance
EUCL	2.3291516	NA	distance
MSW	1.5579529	2.000000	similarity
SW	12.0570984	NA	similarity
MKL	0.9314823	NA	distance
KL	5.5888940	NA	distance

In this case not allowing the reverse complement of the motifs did nothing; this is because the current motif orientations had better scores regardless. Next, whether to compare the motifs as PPM or ICM:

```
try_all(motifs, use.type = "ICM")
```

method	score	max	type
MPCC	0.4577775	1.000000	similarity

method	score	max	type
PCC	23.8933593	NA	similarity
MEUCL	0.6497634	1.414214	distance
EUCL	4.2120436	NA	distance
MSW	0.7782892	2.000000	similarity
SW	6.2263136	NA	similarity
MKL	1.5366031	NA	distance
KL	10.6218155	NA	distance

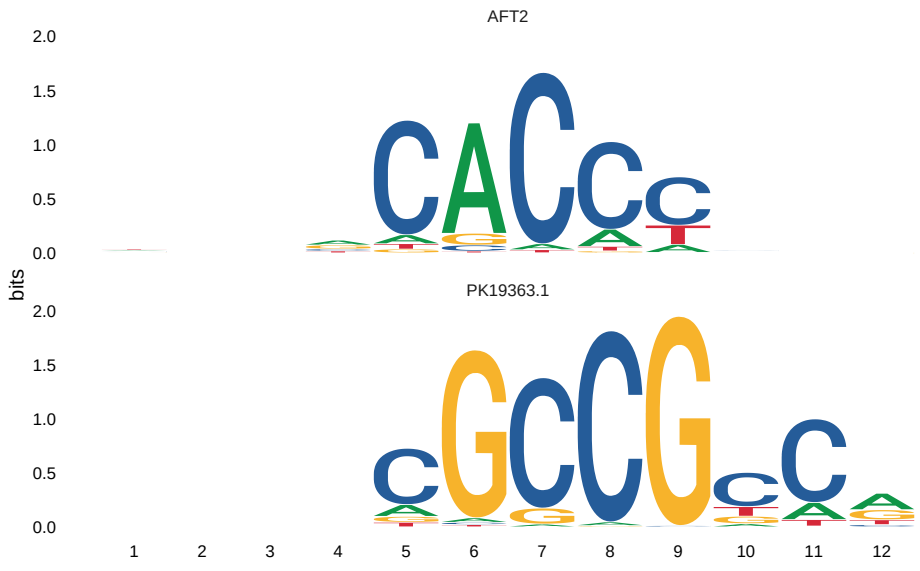
Going from comparing PPM motifs to ICM motifs changes the scores quite a bit; in this case noticeable by increasing the distance between the two. Be careful about jumping to using `use.type = "ICM"` though; keep in mind that the conversion from PPM is ICM is not linear, so the comparison is inherently quite different in nature. Next, `normalise.scores`:

```
try_all(motifs, normalise.scores = TRUE)
```

method	score	max	type
MPCC	0.3087418	1.000000	similarity
PCC	13.8528827	NA	similarity
MEUCL	0.5233627	1.414214	distance
EUCL	3.8819194	NA	distance
MSW	1.2057098	2.000000	similarity
SW	9.6456787	NA	similarity
MKL	1.2424547	NA	distance
KL	9.3148234	NA	distance

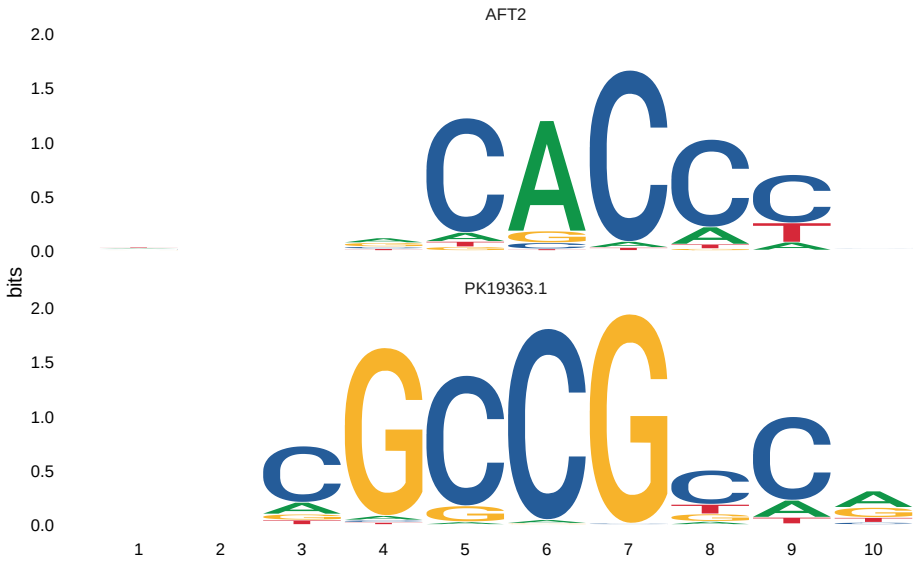
Again, using this option increases the distance between motifs. To explain this, let's visualize the motif alignment:

```
view_motifs(motifs)
```



From this image you can see that there is a bit of an overhang. What `normalise.scores` does is punish this overhang by multiplying the final score by the ratio of aligned to un-aligned positions. To illustrate this further, we can restrict the alignment so there are as few as possible overhangs:

```
view_motifs(motifs, min.overlap = 99)
```




```
try_all(motifs, min.overlap = 99)
```

method	score	max	type
MPCC	0.2705641	1.000000	similarity
PCC	17.3161033	NA	similarity
MEUCL	0.4186901	1.414214	distance
EUCL	3.3495210	NA	distance
MSW	1.5071373	2.000000	similarity
SW	12.0570984	NA	similarity
MKL	0.9939638	NA	distance
KL	7.9517102	NA	distance

```
try_all(motifs, min.overlap = 99, normalise.scores = TRUE)
```

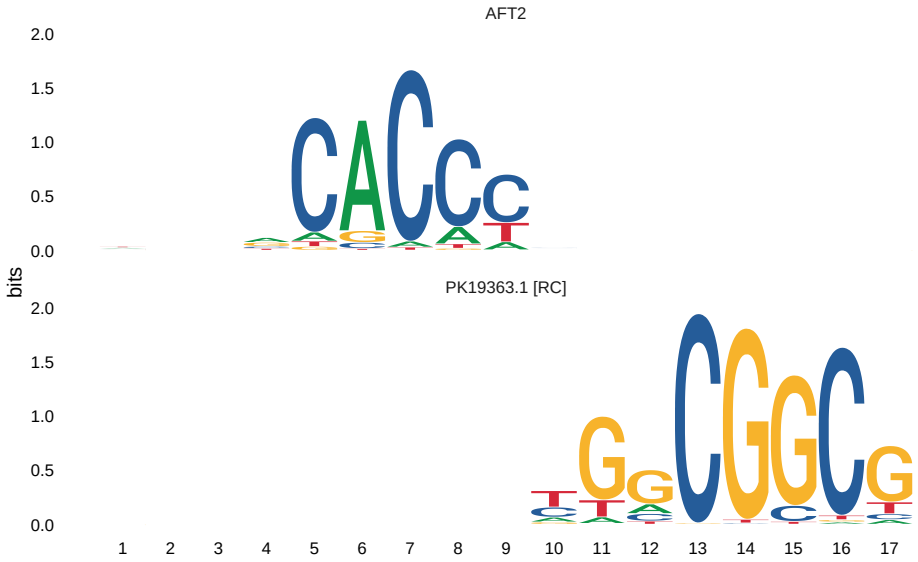
method	score	max	type
MPCC	0.2164513	1.000000	similarity
PCC	13.8528827	NA	similarity
MEUCL	0.5233627	1.414214	distance
EUCL	4.1869012	NA	distance
MSW	1.2057098	2.000000	similarity
SW	9.6456787	NA	similarity
MKL	1.2424547	NA	distance
KL	9.9396378	NA	distance

Now, using `normalise.scores = TRUE` is less punishing. These options are important; turning them off results in the following:

```
try_all(motifs, min.overlap = 1)
```

method	score	max	type
MPCC	0.8198668	1.000000	similarity
PCC	18.5245085	NA	similarity
MEUCL	0.3325102	1.414214	distance
EUCL	0.3325102	NA	distance
MSW	1.7788739	2.000000	similarity
SW	12.0570984	NA	similarity
MKL	0.3737297	NA	distance
KL	0.3737297	NA	distance

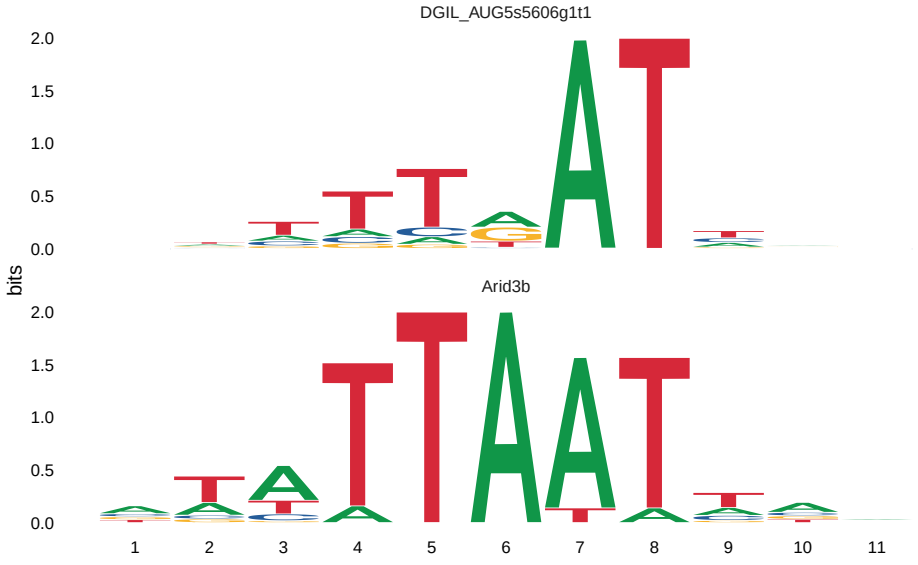
```
view_motifs(motifs, min.overlap = 1)
```



In trying to find the highest possible score, `compare_motifs()` ended up with this alignment, which most would agree is not a fair comparison of the two motifs. The final parameter I will discuss is `min.mean.ic`. When you look at the motifs being compared, they both have low information content regions. The `compare_motifs()` function allows you decide whether you want low information content positions to be scored (positions which don't pass the set threshold will not contribute to the score).

```
motifs2 <- filter_motifs(MotifDb, altname = c("M0100_1.02", "M0104_1.02"))
#> motifs converted to class 'universalmotif'
view_motifs(motifs2, min.mean.ic = 0)
```

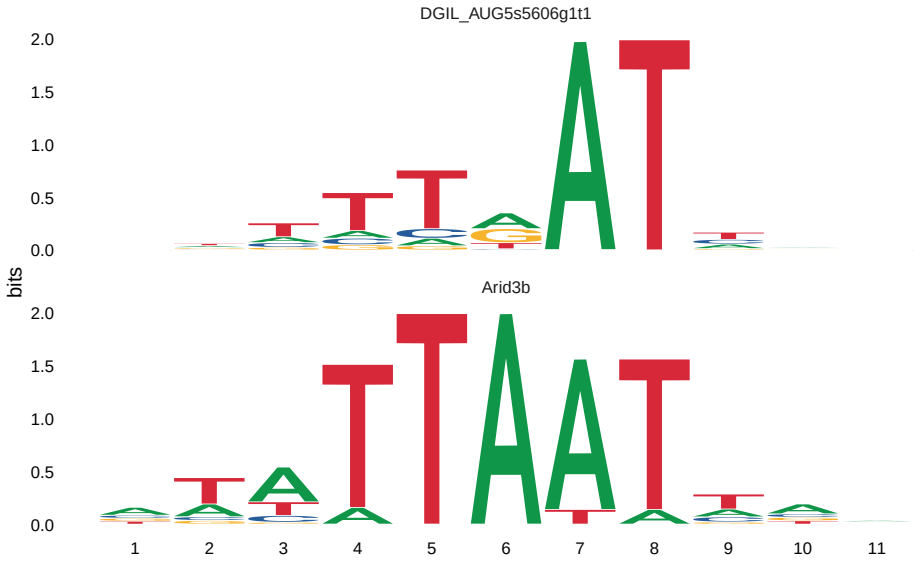
Advanced usage




```
try_all(motifs2, min.mean.ic = 0)
```

method	score	max	type
MPCC	0.9008912	1.000000	similarity
PCC	67.1572807	NA	similarity
MEUCL	0.2075896	1.414214	distance
EUCL	1.8508969	NA	distance
MSW	1.8802405	2.000000	similarity
SW	16.9221646	NA	similarity
MKL	0.2900102	NA	distance
KL	2.6100919	NA	distance

```
view_motifs(motifs2, min.mean.ic = 0.7)
```



```
try_all(motifs2, min.mean.ic = 0.7)
```

method	score	max	type
MPCC	0.9008912	1.000000	similarity
PCC	44.1436684	NA	similarity
MEUCL	0.2644138	1.414214	distance
EUCL	1.4142136	NA	distance
MSW	1.8000058	2.000000	similarity
SW	12.6000409	NA	similarity
MKL	0.3989916	NA	distance
KL	2.7929414	NA	distance

In this case, changing the `min.mean.ic` did not alter the alignment, but had a great impact on the scores. Setting this too high can be quite punishing; I have found `min.mean.ic = 0.5` to be a good middle ground.

6 Motif trees with ggtree

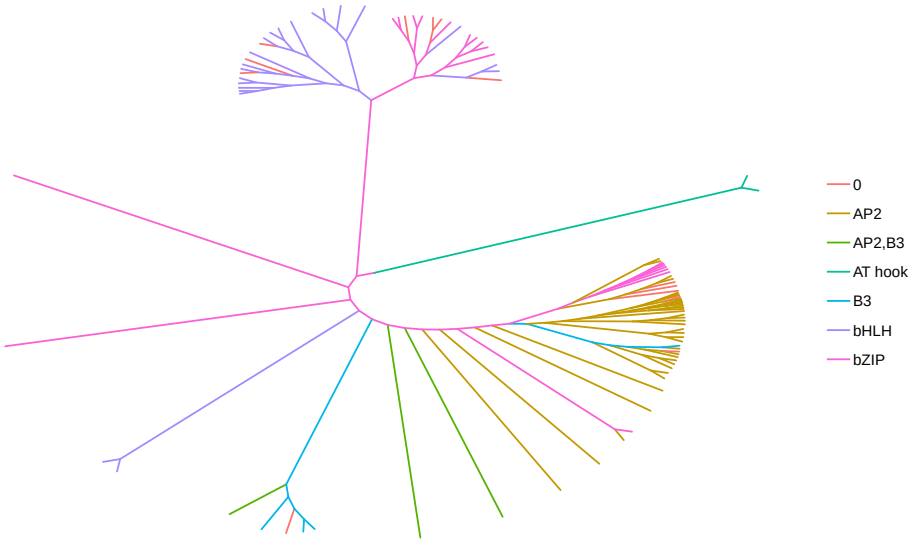
Additionally, this package introduces the `motif_tree()` function for generating basic tree-like diagrams for comparing motifs. This allows for a visual result from `compare_motifs()`. All options from `compare_motifs()` are available in `motif_tree()`. This function uses the *ggtree* package and outputs 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 <- convert_motifs(MotifDb)
motifs <- filter_motifs(motifs, organism = "Athaliana")[1:100]
tree <- motif_tree(motifs, layout = "daylight", linecol = "family",
                   progress = FALSE)
#> Average angle change [1] 0.0728586923874488
#> Average angle change [2] 0.024920471919218

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

tree
```



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, progress = FALSE)

## Step 2: create a "dist" object

# The default metric, MPCC, 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))
```

7 Motif discovery with MEME

The *universalmotif* package provides a simple wrapper to the powerful motif discovery tool MEME (Bailey and Elkan 1994). To run an analysis with MEME, all that is required is a set of `XStringSet` class sequences (defined in the *Biostrings* package), and `run_meme()` will take care of running the program and reading the output for use within R.

The first step is to check that R can find the MEME binary in your `$PATH` by running `run_meme()` without any parameters. If successful, you should see the default MEME help message in your console. If not, then you'll need to provide the complete path to the MEME binary. There are two options:

```
library(universalmotif)

## 1. Once per session: via `options`
```

```
options(meme.bin = "/path/to/meme/bin/meme")

run_meme(...)

## 2. Once per run: via `run_meme`

run_meme(..., bin = "/path/to/meme/bin/meme")
```

Now we need to get some sequences to use with `run_meme()`. At this point we can read sequences from disk or extract them from one of the Bioconductor `BSgenome` packages.

```
library(universalmotif)
data(ArabidopsisPromoters)

## 1. Read sequences from disk (in fasta format):

library(Biostrings)

# The following `read*` functions are available in Biostrings:
# DNA: readDNAStringSet
# DNA with quality scores: readQualityScaledDNAStringSet
# RNA: readRNAStringSet
# amino acid: readAAStringSet
# any: readBStringSet

sequences <- readDNAStringSet("/path/to/sequences.fasta")

run_meme(sequences, ...)

## 2. Extract from a `BSgenome` object:

library(GenomicFeatures)
library(TxDb.Athaliana.BioMart.plantsmart28)
library(BSgenome.Athaliana.TAIR.TAIR9)

# Let us retrieve the same promoter sequences from ArabidopsisPromoters:
gene.names <- names(ArabidopsisPromoters)

# First get the transcript coordinates from the relevant `TxDb` object:
transcripts <- transcriptsBy(TxDb.Athaliana.BioMart.plantsmart28,
                             by = "gene")[gene.names]

# There are multiple transcripts per gene, we only care for the first one
# in each:

transcripts <- lapply(transcripts, function(x) x[1])
transcripts <- unlist(GRangesList(transcripts))

# Then the actual sequences:

# Unfortunately this is a case where the chromosome names do not match
```

```
# between the two databases

seqlevels(TxDb.Athaliana.BioMart.plantsmart28)
#> [1] "1"  "2"  "3"  "4"  "5"  "Mt" "Pt"
seqlevels(BSgenome.Athaliana.TAIR.TAIR9)
#> [1] "Chr1" "Chr2" "Chr3" "Chr4" "Chr5" "ChrM" "ChrC"

# So we must first rename the chromosomes in `transcripts`:
seqlevels(transcripts) <- seqlevels(BSgenome.Athaliana.TAIR.TAIR9)

# Finally we can extract the sequences
promoters <- getPromoterSeq(transcripts,
                            BSgenome.Athaliana.TAIR.TAIR9,
                            upstream = 0, downstream = 1000)

run_meme(promoters, ...)
```

Once the sequences are ready, there are few important options to keep in mind. One is whether to conserve the output from MEME. The default is not to, but this can be changed by setting the relevant option.

```
run_meme(sequences, output = "/path/to/desired/output/folder")
```

The second important option is the search function (`objfun`). Some search functions such as the default `classic` do not require a set of background sequences, whilst some do (such as `de`). If you choose one of the latter, then you can either let MEME create them for you (it will shuffle the target seqeunces) or you can provide them via the `control.sequences` parameter.

Finally, choose how you'd like the data imported into R. Once the MEME program exits, `run_meme()` will import the results into R with `read_meme()`; at this point you can decide if you want just the motifs themselves (`readsites = FALSE`) or if you'd like the original sequence sites as well (`readsites = TRUE`, the default).

There are a wealth of other MEME options available, such as the number of desired motifs (`nmotifs`), the width of desired motifs (`minw`, `maxw`), the search mode (`mod`), assigning sequence weights (`weights`), using a custom alphabet (`alph`), and many others. See the output from `run_meme()` for a brief description of the options, or visit the online manual for more details.

Session info

```
#> R version 3.6.0 (2019-04-26)
#> Platform: x86_64-w64-mingw32/x64 (64-bit)
#> Running under: Windows Server 2012 R2 x64 (build 9600)
#>
#> Matrix products: default
#>
#> locale:
#> [1] LC_COLLATE=C
#> [2] LC_CTYPE=English_United States.1252
#> [3] LC_MONETARY=English_United States.1252
```

```
#> [4] LC_NUMERIC=C
#> [5] LC_TIME=English_United States.1252
#>
#> attached base packages:
#> [1] stats4    parallel  stats     graphics  grDevices utils     datasets
#> [8] methods   base
#>
#> other attached packages:
#>  [1] ggplot2_3.1.1        ggtree_1.16.0        MotifDb_1.26.0
#>  [4] Biostrings_2.52.0    XVector_0.24.0       IRanges_2.18.0
#>  [7] S4Vectors_0.22.0     BiocGenerics_0.30.0  universalmotif_1.2.1
#> [10] BiocStyle_2.12.0
#>
#> loaded via a namespace (and not attached):
#>  [1] Rcpp_1.0.1                  ape_5.3
#>  [3] lattice_0.20-38             tidyr_0.8.3
#>  [5] Rsamtools_2.0.0             ps_1.3.0
#>  [7] ggseqlogo_0.1               gtools_3.8.1
#>  [9] assertthat_0.2.1            digest_0.6.18
#> [11] R6_2.4.0                    GenomeInfoDb_1.20.0
#> [13] plyr_1.8.4                  evaluate_0.13
#> [15] highr_0.8                   pillar_1.3.1
#> [17] Rdpack_0.11-0               zlibbioc_1.30.0
#> [19] rlang_0.3.4                 lazyeval_0.2.2
#> [21] data.table_1.12.2           Matrix_1.2-17
#> [23] rmarkdown_1.12              labeling_0.3
#> [25] BiocParallel_1.18.0         stringr_1.4.0
#> [27] RCurl_1.95-4.12             munsell_0.5.0
#> [29] DelayedArray_0.10.0         compiler_3.6.0
#> [31] rtracklayer_1.44.0          xfun_0.6
#> [33] pkgconfig_2.0.2             htmltools_0.3.6
#> [35] SummarizedExperiment_1.14.0 tidyselect_0.2.5
#> [37] tibble_2.1.1                GenomeInfoDbData_1.2.1
#> [39] bookdown_0.9                matrixStats_0.54.0
#> [41] XML_3.98-1.19               withr_2.1.2
#> [43] crayon_1.3.4                dplyr_0.8.0.1
#> [45] GenomicAlignments_1.20.0    bitops_1.0-6
#> [47] grid_3.6.0                  nlme_3.1-139
#> [49] jsonlite_1.6                gtable_0.3.0
#> [51] magrittr_1.5                scales_1.0.0
#> [53] bibtex_0.4.2                tidytree_0.2.4
#> [55] stringi_1.4.3               splitstackshape_1.4.8
#> [57] rvcheck_0.1.3               tools_3.6.0
#> [59] treeio_1.8.0                Biobase_2.44.0
#> [61] glue_1.3.1                  purrr_0.3.2
#> [63] processx_3.3.1              yaml_2.2.0
#> [65] colorspace_1.4-1            BiocManager_1.30.4
#> [67] GenomicRanges_1.36.0        gbRd_0.4-11
#> [69] knitr_1.22
```

References

Bailey, T.L., and C. Elkan. 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.

Guo, Y., K. Tian, H. Zeng, X. Guo, and D.K. Gifford. 2018. "A Novel K-Mer Set Memory (KSM) Motif Representation Improves Regulatory Variant Prediction." *Genome Research* 28:891–900.

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.

Mathelier, A., and W.W. Wasserman. 2013. "The Next Generation of Transcription Factor Binding Site Prediction." *PLoS Computational Biology* 9 (9):e1003214.

Siebert, M., and J. Soding. 2016. "Bayesian Markov Models Consistently Outperform PWMs at Predicting Motifs in Nucleotide Sequences." *Nucleic Acids Research* 44 (13):6055–69.