Motif comparisons and P-values

Benjamin Jean-Marie Tremblay*

25 May 2019

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

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 few functions available in other Bioconductor packages which allow for motif comparison. These include `PWMSimlarity()` (`TFBSTools`), `motifDistances()` (`MotIV`), and `motifSimilarity()` (`PWMEnrich`). Unfortunately these functions are not designed for comparing large numbers of motifs, and can result in long

*b2tremblay@uwaterloo.ca

run times. Furthermore they are restrictive in their option range. The `universalmotif` package aims to fix this by providing the `compare_motifs()` function.

The main workhorse in `universalmotif` is `compare_motifs()`. Several other functions also make use of these metrics, including `merge_motifs()`, `view_motifs()` and `compare_columns()`.

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, or median.

If you're interested in simply comparing two columns individually, `compare_columns()` can be used:

```
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 well 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.

2.2 Comparison parameters

There are several key parameters to keep in mind when comparing motifs. These include:

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

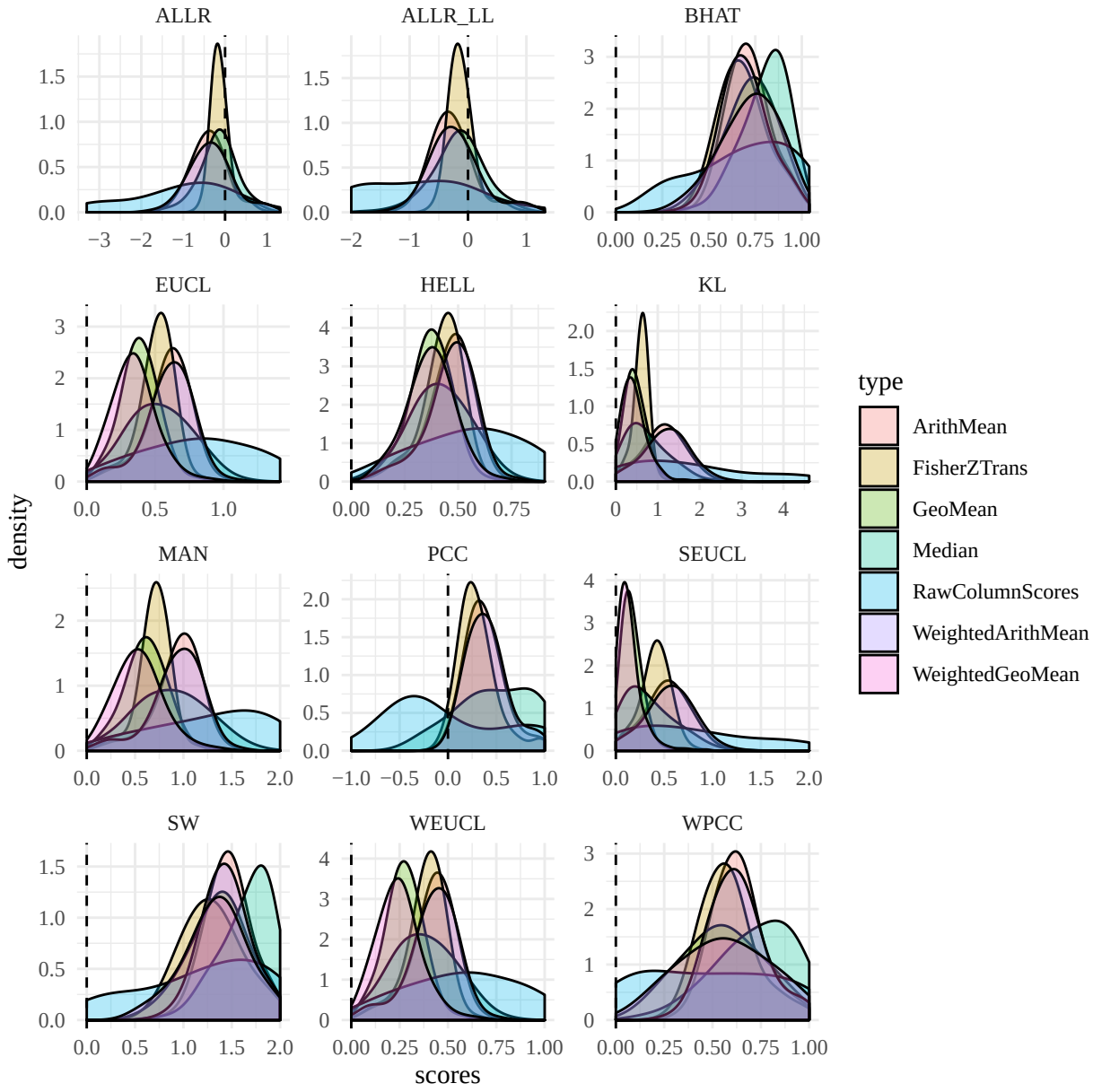


Figure 1: Distributions of scores from approximately 500 random motif and individual column comparisons

- `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:

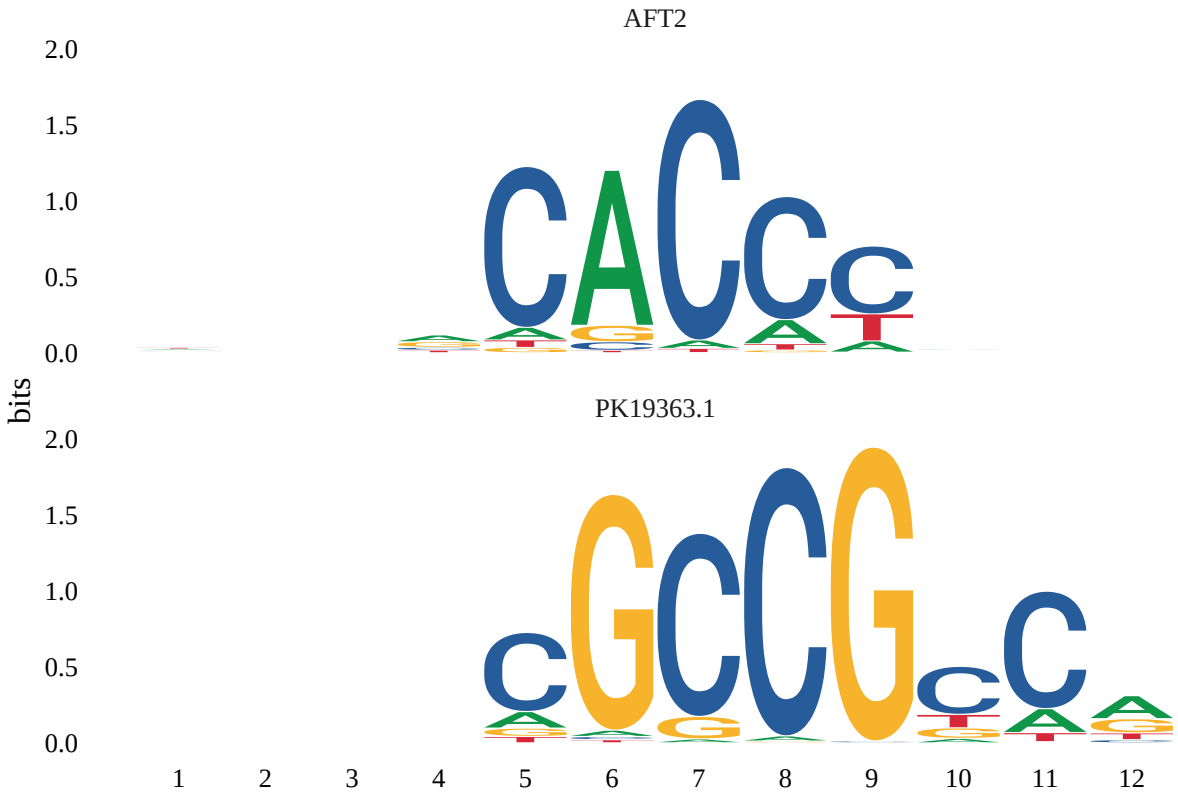


Figure 2: Example scores from comparing two motifs

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
#>   <character> <numeric>              <character> <integer>         <numeric>
#> 1       ORA59         1 ERF105 [duplicated #299]       793 0.9765521339417 39
#> 2       ORA59         1  ERF13 [duplicated #293]       641 0.9646661186831 46
#> 3       ORA59         1                   ERF15       618 0.9628523329644 96
#> 4       ORA59         1                    ESE1       651 0.9199936295414 05
#> 5       ORA59         1               AT4G18450       615 0.9662107238240 13
#> 6       ORA59         1                  ERF104       626 0.8761528744597 45
#>             logPval                 Pval                Eval
#>           <numeric>            <numeric>           <numeric>
#> 1 -10.2568572361643 3.51158755047301e-05 0.0563960960605965
#> 2 -10.0963047691211 4.12316344771845e-05 0.0662180049703583
#> 3 -10.0719280646799 4.22490764031755e-05 0.0678520167034998
#> 4 -9.50541194646771 7.44478299903147e-05  0.119563214964445
#> 5 -8.72010246270766 0.000163270460931215  0.262212360255532
#> 6 -8.32777352415559 0.000241709606823104  0.388185628557904
```

P-values are made possible by estimating logistic distribution (usually the best fitting distribution for motif comparisons) parameters from randomized motif scores, then using `plogis()` 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()`

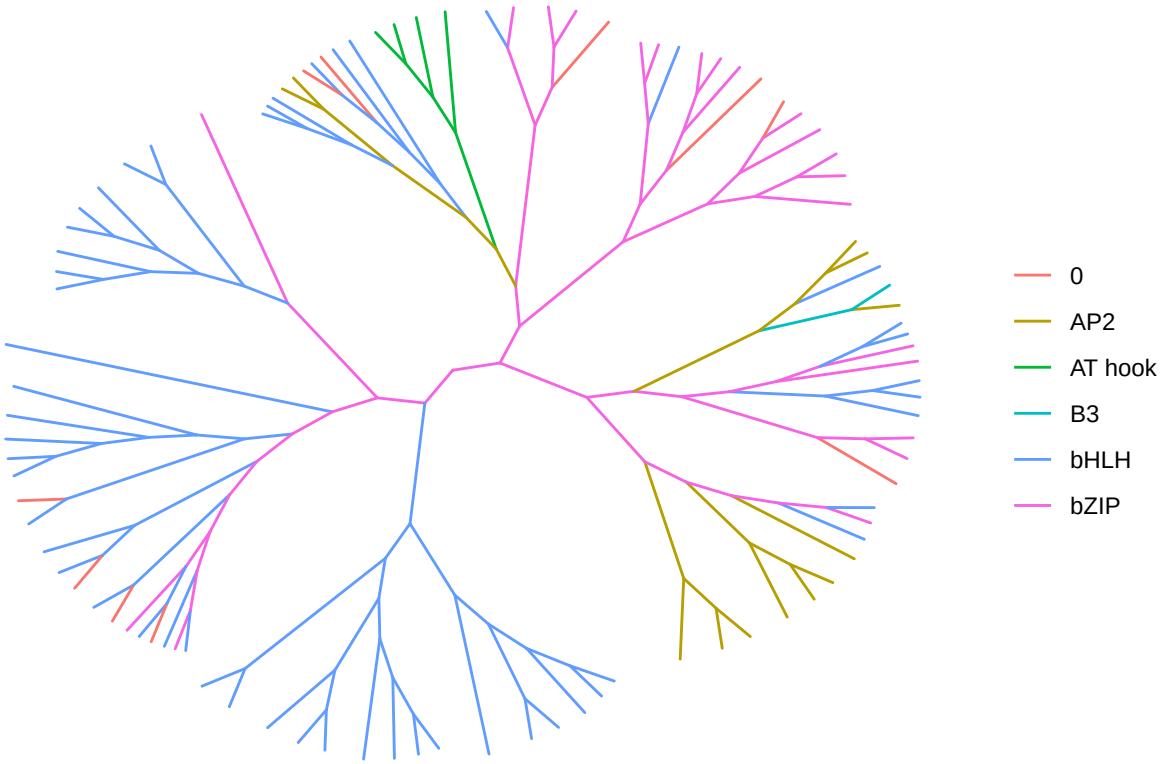
Additionally, this package introduces the `motif_tree()` function for generating basic tree-like diagrams for comparing motifs. This allows for a visual result from `compare_motifs()`. All options from `compare_motifs()` are available in `motif_tree()`. This function uses the `ggtree` package and outputs 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")
#> Average angle change [1] 0.0913800274936305
#> Average angle change [2] 0.0226652873008604

## 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
```

```r
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))
```

4 Motif P-values

4.1 Calculating P-values from scores

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)
#> DataFrame with 6 rows and 12 columns
#>         motif   motif.i    sequence     start      stop     score       match
#>   <character> <integer> <character> <integer> <integer> <numeric> <character>
#> 1       motif         1   AT2G15390       337       346     5.643  CCCCGGAGAC
#> 2       motif         1   AT1G76590       848       857     5.869  CTCTAGAGAC
#> 3       motif         1   AT1G08090       925       934     5.301  CTCCAAAGAA
#> 4       motif         1   AT5G64310       768       777     6.057  GTCCAGATTC
#> 5       motif         1   AT5G08790       734       743     5.617  GTCTAAAGTC
#> 6       motif         1   AT1G49840       980       989     5.292  CTCTGGATTC
#>   thresh.score min.score max.score        score.pct      strand
#>      <numeric> <numeric> <numeric>        <numeric> <character>
#> 1       5.2248     -15.4     6.531 86.4033073036288           +
#> 2       5.2248     -15.4     6.531 89.8637268412188           +
#> 3       5.2248     -15.4     6.531  81.166743224621           +
#> 4       5.2248     -15.4     6.531 92.7423059255857           +
#> 5       5.2248     -15.4     6.531 86.0052059408973           +
```

```
#> 6       5.2248     -15.4     6.531 81.0289389067524           +
```

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 CTCTAGAGAC, with a score of 5.869 (max possible = 6.531)

bkg <- get_bkg(ArabidopsisPromoters, 1, list.out = FALSE)
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 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.

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

motif	match	pval
motif	GTCCAGATTC	4.428792e-06
motif	CTCTAGAGAC	9.940623e-06
motif	CCCCGGAGAC	2.933222e-05
motif	GTCTAAAGTC	3.673628e-05
motif	CTCCAAAGAA	1.054900e-04
motif	CTCTGGATTC	1.119204e-04

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

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)
  }
}

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 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.

4.2 Calculating scores from P-values

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. 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:

```
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)

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

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

```
pvals.approx6 <- motif_pvalue(motif, score = scores, bkg.probs = bkg, k = 6)
pvals.approx8 <- motif_pvalue(motif, score = scores, 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	1.324	1.000299e-01	9.995747e-02	9.995747e-02	1.3677	1.3239
1e-02	3.596	1.000309e-02	9.991646e-03	9.991646e-03	3.6273	3.5991
1e-03	4.858	1.000404e-03	9.984970e-04	9.984970e-04	4.8568	4.8721
1e-04	5.647	1.001358e-04	9.727478e-05	9.727478e-05	5.6238	5.6722
1e-05	6.182	1.049042e-05	9.536743e-06	9.536743e-06	5.3307	6.0643

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)

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

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

pvals.approx6 <- motif_pvalue(motif, score = scores, bkg.probs = bkg, k = 6)
pvals.approx8 <- motif_pvalue(motif, score = scores, 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	-2.000	-2.0205	-1.9998
-1	3.354645e-01	3.353453e-01	3.353453e-01	-1.000	-1.0161	-0.9982
0	2.185555e-01	2.184534e-01	2.184534e-01	0.000	0.0075	0.0015
1	1.244183e-01	1.243525e-01	1.243525e-01	1.000	0.9984	0.9990
2	5.911160e-02	5.907822e-02	5.907822e-02	2.000	1.9936	2.0023
3	2.163410e-02	2.160931e-02	2.160931e-02	3.000	2.9727	2.9993
4	5.360603e-03	5.347252e-03	5.347252e-03	4.000	3.9626	3.9982
5	7.162094e-04	7.152557e-04	7.152557e-04	5.000	5.0445	5.0178
6	2.193451e-05	2.193451e-05	2.193451e-05	6.057	5.5074	5.9239

As you can see, results from exact calculations are not *quite* exact but close regardless.

Session info

```
#> R version 3.6.2 (2019-12-12)
#> Platform: x86_64-pc-linux-gnu (64-bit)
#> Running under: Ubuntu 18.04.3 LTS
#>
#> Matrix products: default
#> BLAS:   /home/biocbuild/bbs-3.10-bioc/R/lib/libRblas.so
#> LAPACK: /home/biocbuild/bbs-3.10-bioc/R/lib/libRlapack.so
#>
#> locale:
#>  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
#>  [3] LC_TIME=en_US.UTF-8        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
#>
#> attached base packages:
#> [1] stats4    parallel  stats     graphics  grDevices utils     datasets
#> [8] methods   base
#>
#> other attached packages:
#>  [1] dplyr_0.8.4          ggtree_2.0.1         ggplot2_3.2.1
#>  [4] MotifDb_1.28.0       Biostrings_2.54.0    XVector_0.26.0
#>  [7] IRanges_2.20.2       S4Vectors_0.24.3     BiocGenerics_0.32.0
#> [10] universalmotif_1.4.8
#>
#> loaded via a namespace (and not attached):
#>  [1] Rcpp_1.0.3                  ape_5.3
#>  [3] lattice_0.20-40             tidyr_1.0.2
#>  [5] Rsamtools_2.2.3             ps_1.3.2
#>  [7] ggseqlogo_0.1               assertthat_0.2.1
#>  [9] digest_0.6.25               R6_2.4.1
#> [11] GenomeInfoDb_1.22.0         evaluate_0.14
#> [13] highr_0.8                   pillar_1.4.3
#> [15] Rdpack_0.11-1               zlibbioc_1.32.0
#> [17] rlang_0.4.4                 lazyeval_0.2.2
#> [19] data.table_1.12.8           Matrix_1.2-18
#> [21] rmarkdown_2.1               labeling_0.3
#> [23] BiocParallel_1.20.1         stringr_1.4.0
#> [25] RCurl_1.98-1.1              munsell_0.5.0
#> [27] tinytex_0.20                DelayedArray_0.12.2
#> [29] compiler_3.6.2              rtracklayer_1.46.0
#> [31] xfun_0.12                   pkgconfig_2.0.3
#> [33] htmltools_0.4.0             SummarizedExperiment_1.16.1
#> [35] tidyselect_1.0.0            tibble_2.1.3
#> [37] GenomeInfoDbData_1.2.2      bookdown_0.17
#> [39] matrixStats_0.55.0          XML_3.99-0.3
#> [41] withr_2.1.2                 crayon_1.3.4
```

```
#> [43] GenomicAlignments_1.22.1    MASS_7.3-51.5
#> [45] bitops_1.0-6                grid_3.6.2
#> [47] nlme_3.1-144                jsonlite_1.6.1
#> [49] gtable_0.3.0                lifecycle_0.1.0
#> [51] magrittr_1.5                scales_1.1.0
#> [53] bibtex_0.4.2.2              tidytree_0.3.1
#> [55] stringi_1.4.6               farver_2.0.3
#> [57] splitstackshape_1.4.8       rvcheck_0.1.7
#> [59] vctrs_0.2.3                 tools_3.6.2
#> [61] treeio_1.10.0               Biobase_2.46.0
#> [63] glue_1.3.1                  purrr_0.3.3
#> [65] processx_3.4.2              yaml_2.2.1
#> [67] colorspace_1.4-1            BiocManager_1.30.10
#> [69] GenomicRanges_1.38.0        gbRd_0.4-11
#> [71] knitr_1.28
```

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.

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.