Sequence manipulation and scanning

Benjamin Jean-Marie Tremblay*

4 May 2021

Abstract

Sequences stored as XStringSet objects (from the Biostrings package) can be used by several functions in the universalmotif package. These functions are demonstrated here and fall into two categories: sequence manipulation and motif scanning. Sequences can be generated, shuffled, and background frequencies of any order calculated. Scanning can be done simply to find locations of motif hits above a certain threshold, or to find instances of enriched motifs.

Contents

*benjamin.tremblay@uwaterloo.ca

1 Introduction

This vignette goes through generating your own sequences from a specified background model, shuffling sequences whilst maintaining a certain k-let size, and the scanning of sequences and scoring of motifs. 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 a discussion on motif comparisons and P-values, see the motif comparisons and P-values vignette.

2 Creating random sequences

The `Biostrings` package offers an excellent suite of functions for dealing with biological sequences. The `universalmotif` package hopes to help extend these by providing the `create_sequences()` and `shuffle_sequences()` functions. The first of these, `create_sequences()`, generates a set of letters in random order, then passes these strings to the `Biostrings` package to generate the final `XStringSet` object. The number and length of sequences can be specified. The probabilities of individual letters can also be set.

The `freqs` option of `create_sequences()` also takes higher order backgrounds. In these cases the sequences are constructed in a Markov-style manner, where the probability of each letter is based on which letters precede it.

```
library(universalmotif)
library(Biostrings)

## Create some DNA sequences for use with an external program (default
## is DNA):

sequences.dna <- create_sequences(seqnum = 500,
                                  freqs = c(A=0.3, C=0.2, G=0.2, T=0.3))
## writeXStringSet(sequences.dna, "dna.fasta")
sequences.dna
#> DNAStringSet object of length 500:
#>       width seq
#>   [1]   100 CAGCGCATATTCTACGTAAGAACGCCGCAGTTG...TCATTAGCCCCTTTAGTTTGTGAGAAAGGCAT
#>   [2]   100 GGATGACTCTAATCAGATTTATCATCGGGTCTC...CTTGCTGCTATGGGTTGGATTATACGTTGTGA
#>   [3]   100 CATCAGGAAGCATGTTGCACCTTTGTCCTTTGA...GACGCGAAACGATCCGCTAGTTTTATGCTTGA
#>   [4]   100 ATGGTATTTTATAAACCCTAGCATGGCATACTA...TATATGGTGTATAGCACAGCAGCTTATCACTG
#>   [5]   100 ACACTATCTCCAGTTTACCCGAAAGATTGTGTT...GTCCTTGACTATGTTGAATGGTCAATTCCTTA
#>   ...   ... ...
#> [496]   100 GAACCATCAGATAATAGTTCTCATCATGCTCTT...CTCACTTTCGACGACGGATAGTACGTATAACA
#> [497]   100 GTGACATTGGATAATGCCTTAGATTTAAAACGC...TATACCTAATAAATAGCAATGTACGTAAGCTA
#> [498]   100 CATACAACCGGGAGAAACGAGCATTTGAATCCT...ACAAACCTCAACTATATCCCTAGAATATAACT
#> [499]   100 AGCACTAAGAATGTCAAATGTGATGGCTGTTCT...AAAACGGTCACACGGAAACACAGCCAAAGTGA
#> [500]   100 TTATAAATACTTGAGGATTCTATAAATTTCACC...TCCGAATGACAAAAGCTAGTCAGTATCTTTGA

## Amino acid:

create_sequences(alphabet = "AA")
#> AAStringSet object of length 100:
#>       width seq
#>   [1]   100 RALEWQFHATTILLKFFYQLFWFVPMEIHEEDE...YMRTFWCVRHFEWHVKNHHSFAALMSELCKPN
#>   [2]   100 VMNIRPAVCQQCYIYLRPNIPAGMQCAYVTFLS...HQPCNIFMNRPGQPWHHTNIHGSQDIHLFITF
#>   [3]   100 YPGSLYYPMGRFKSDYCYPVVLNQPNGEEPHRL...SITHHDWERFFCMGICPDWEKGMLIFPKKDKT
#>   [4]   100 YVNECTINHLTRHQLLQDTPCPMEDPYQMSMWT...QDRVYAQDRKDGTRTRQRLENWIRTNAAGFMC
#>   [5]   100 NQLKCNRVMAILCGFPDQLHYKMTYRTGAWEHF...FPKVMECNVYWVGPFPVHWVMHFYMCVSTLNP
```

```
#>   ...   ... ...
#>  [96]   100 VRVGPVCNLRACFTPPSEPEIMSPPNRLYEGIG...ELTVWEKVGRAQDRGSPVGFETCFFDDYCNCR
#>  [97]   100 DYNERNTPPRSKQETGAEVWCFCNEWPVVQFEC...VGMLTFDDSAYGHESRNDYLGFTDNQGYWYPY
#>  [98]   100 SEHESTDFGNCTYEWKNPFRAVRFPNGHNMRGP...VSMGYIQGTYYCKREKGNVIDCVIHDWCGAYE
#>  [99]   100 TVSEFGHPMWIEEKAAMNDVIFIKFDRPCAMGI...NQNVGNEKKAPAIVYSMSEICPEEMWNLGCDS
#> [100]   100 NDWVSWWQSYKQLRQWDWVHEWHLQVAYCLIVG...RPMRPFMISTRNTGYQPWVNGLDCVEGYWEYH

## Any set of characters can be used

create_sequences(alphabet = paste0(letters, collapse = ""))
#> BStringSet object of length 100:
#>       width seq
#>   [1]   100 afhdraajemknpupraitzridcisfuqimii...lnombgvysbokdqfpcutujdvhoaeltqvz
#>   [2]   100 opvjrphpczzdejybgpzkasfgjkdtzeznj...iefvidwygbkntdcogxzojtcuzrhjamxu
#>   [3]   100 vqrdegvtcavokaushpaqardlgihtrttfj...fxnfhkgmztmbuufrysvoakkyeagteobg
#>   [4]   100 kahbkfvuvtdsnsphgnlnfghcgdatqnduc...ypilsqfsqedfonszdfbknukeevhrsvdj
#>   [5]   100 csxievardkzyivwmlkaezzllpclabezpq...ctkbrumxzughmjmxolppuuansyeatpmd
#>   ...   ... ...
#>  [96]   100 mwioedtvtqfclvmnmiiuppncrrxfvgcwm...lejkapenpyrxaklbapchwvtluhbuqekw
#>  [97]   100 jbtqfmiyhexerdbxdblgpmbpqcqshsolt...bshlzpqalcfqqtqvnsfkquivrgfpqfxh
#>  [98]   100 mldtyxkfoqpceizcphfbyhqkftvxwotmf...bfdresfvgjeljdphdxkeqrhpawzcxpty
#>  [99]   100 ifwpdkhjoixmydrlbgtmhuwsguanuldct...oxbrkolnuvdcpzdaugwosiijrbhmgrbz
#> [100]   100 jwvmtbsnroxwmtruecygifnhwqiigulbh...dygutmvgkyztklrwhqqzynlmwegrenxc
```

3 Calculating sequence background

Sequence backgrounds can be retrieved for DNA and RNA sequences with `oligonucleotideFrequency()` from "`Biostrings`. Unfortunately, no such `Biostrings` function exists for other sequence alphabets. The `universalmotif` package proves `get_bkg()` to remedy this. Similarly, the `get_bkg()` function can calculate higher order backgrounds for any alphabet as well. It is recommended to use the original `Biostrings` for very long (e.g. billions of characters) DNA and RNA sequences whenever possible though, as it is much faster than `get_bkg()`.

```
library(universalmotif)

## Background of DNA sequences:
dna <- create_sequences()
get_bkg(dna, k = 1:2)
#> DataFrame with 20 rows and 3 columns
#>            klet     count probability
#>     <character> <numeric>   <numeric>
#> 1             A      2414   0.2414000
#> 2             C      2517   0.2517000
#> 3             G      2600   0.2600000
#> 4             T      2469   0.2469000
#> 5            AA       588   0.0593939
#> ...         ...       ...         ...
#> 16           GT       619   0.0625253
#> 17           TA       598   0.0604040
#> 18           TC       635   0.0641414
#> 19           TG       597   0.0603030
#> 20           TT       622   0.0628283
```

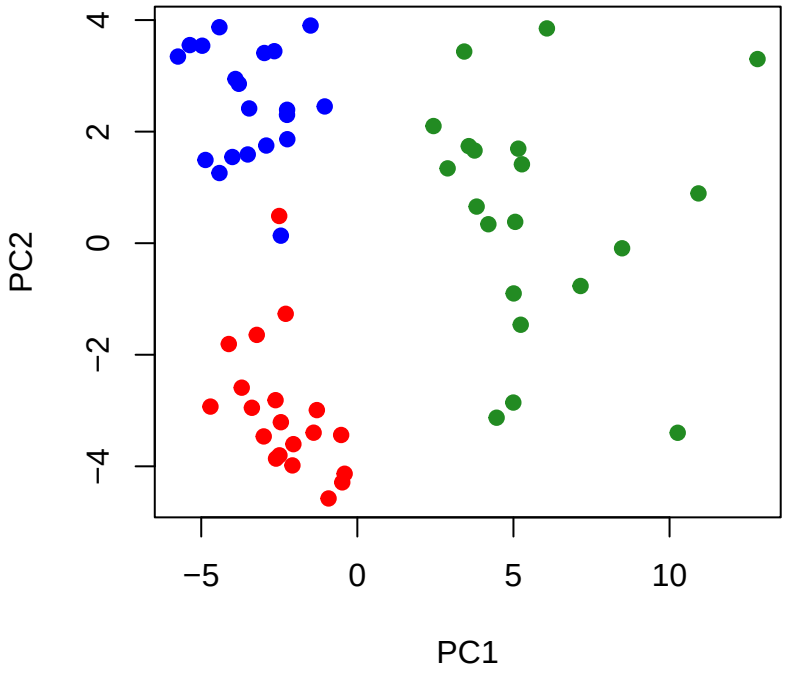
```
## Background of non DNA/RNA sequences:
qwerty <- create_sequences("QWERTY")
get_bkg(qwerty, k = 1:2)
#> DataFrame with 42 rows and 3 columns
#>            klet     count probability
#>     <character> <numeric>   <numeric>
#> 1             E      1679      0.1679
#> 2             Q      1616      0.1616
#> 3             R      1676      0.1676
#> 4             T      1680      0.1680
#> 5             W      1695      0.1695
#> ...         ...       ...         ...
#> 38           YQ       256   0.0258586
#> 39           YR       271   0.0273737
#> 40           YT       297   0.0300000
#> 41           YW       275   0.0277778
#> 42           YY       267   0.0269697
```

4 Clustering sequences by k-let composition

One way to compare sequences is by k-let composition. The following example illustrates how one could go about doing this using only the `universalmotif` package and base graphics.

```
library(universalmotif)

## Generate three random sets of sequences:
s1 <- create_sequences(seqnum = 20,
  freqs = c(A = 0.3, C = 0.2, G = 0.2, T = 0.3))
s2 <- create_sequences(seqnum = 20,
  freqs = c(A = 0.4, C = 0.4, G = 0.1, T = 0.1))
s3 <- create_sequences(seqnum = 20,
  freqs = c(A = 0.2, C = 0.3, G = 0.3, T = 0.2))

## Create a function to get properly formatted k-let counts:
get_klet_matrix <- function(seqs, k, groupName) {
  bkg <- get_bkg(seqs, k = k, merge.res = FALSE)
  bkg <- bkg[, c("sequence", "klet", "count")]
  bkg <- reshape(bkg, idvar = "sequence", timevar = "klet",
    direction = "wide")
  as.data.frame(cbind(Group = groupName, bkg))
}

## Calculate k-let content (up to you what size k you want!):
s1 <- get_klet_matrix(s1, 4, 1)
s2 <- get_klet_matrix(s2, 4, 2)
s3 <- get_klet_matrix(s3, 4, 3)

# Combine everything into a single object:
sAll <- rbind(s1, s2, s3)

## Do the PCA:
sPCA <- prcomp(sAll[, -(1:2)])
```

```
## Plot the PCA:
plot(sPCA$x, col = c("red", "forestgreen", "blue")[sAll$Group], pch = 19)
```



This example could be improved by using `tidyr::spread()` instead of `reshape()` (the former is much faster), and plotting the PCA using the `ggfortify` package to create a nicer `ggplot2` plot. Feel free to play around with different ways of plotting the data! Additionally, you could even try using t-SNE instead of PCA (such as via the `Rtsne` package).

5 Shuffling sequences

When performing *de novo* motif searches or motif enrichment analyses, it is common to do so against a set of background sequences. In order to properly identify consistent patterns or motifs in the target sequences, it is important that there be maintained a certain level of sequence composition between the target and background sequences. This reduces results which are derived purely from base differential letter frequency biases.

In order to avoid these results, typically it desirable to use a set of background sequences which preserve a certain `k`-let size (such as dinucleotide or trinucleotide frequencies in the case of DNA sequences). Though for some cases a set of similar sequences may already be available for use as background sequences, usually background sequences are obtained by shuffling the target sequences, while preserving a desired `k`-let size. For this purpose, a commonly used tool is `uShuffle` (Jiang et al. 2008). The `universalmotif` package aims to provide its own `k`-let shuffling capabilities for use within `R` via `shuffle_sequences()`.

The `universalmotif` package offers three different methods for sequence shuffling: `euler`, `markov` and `linear`. The first method, `euler`, can shuffle sequences while preserving any desired `k`-let size. Furthermore 1-letter counts will always be maintained. However due to the nature of the method, the first and last letters will remain unshuffled. This method is based on the initial random Eulerian walk algorithm proposed by Altschul and Erickson (1985) and the subsequent cycle-popping algorithm detailed by Propp and Wilson (1998) for quickly and efficiently finding Eulerian walks.

The second method, `markov` can only guarantee that the approximate `k`-let frequency will be maintained, but not that the original letter counts will be preserved. The `markov` method involves determining the original `k`-let frequencies, then creating a new set of sequences which will have approximately similar `k`-let frequency.

As a result the counts for the individual letters will likely be different. Essentially, it involves a combination of determining k-let frequencies followed by `create_sequences()`. This type of pseudo-shuffling is discussed by Fitch (1983).

The third method `linear` preserves the original 1-letter counts exactly, but uses a more crude shuffling technique. In this case the sequence is split into sub-sequences every k-let (of any size), which are then re-assembled randomly. This means that while shuffling the same sequence multiple times with `method = "linear"` will result in different sequences, they will all have started from the same set of k-length sub-sequences (just re-assembled differently).

```
library(universalmotif)
library(Biostrings)
data(ArabidopsisPromoters)

## Potentially starting off with some external sequences:
# ArabidopsisPromoters <- readDNAStringSet("ArabidopsisPromoters.fasta")

euler <- shuffle_sequences(ArabidopsisPromoters, k = 2, method = "euler")
markov <- shuffle_sequences(ArabidopsisPromoters, k = 2, method = "markov")
linear <- shuffle_sequences(ArabidopsisPromoters, k = 2, method = "linear")
k1 <- shuffle_sequences(ArabidopsisPromoters, k = 1)
```

Let us compare how the methods perform:

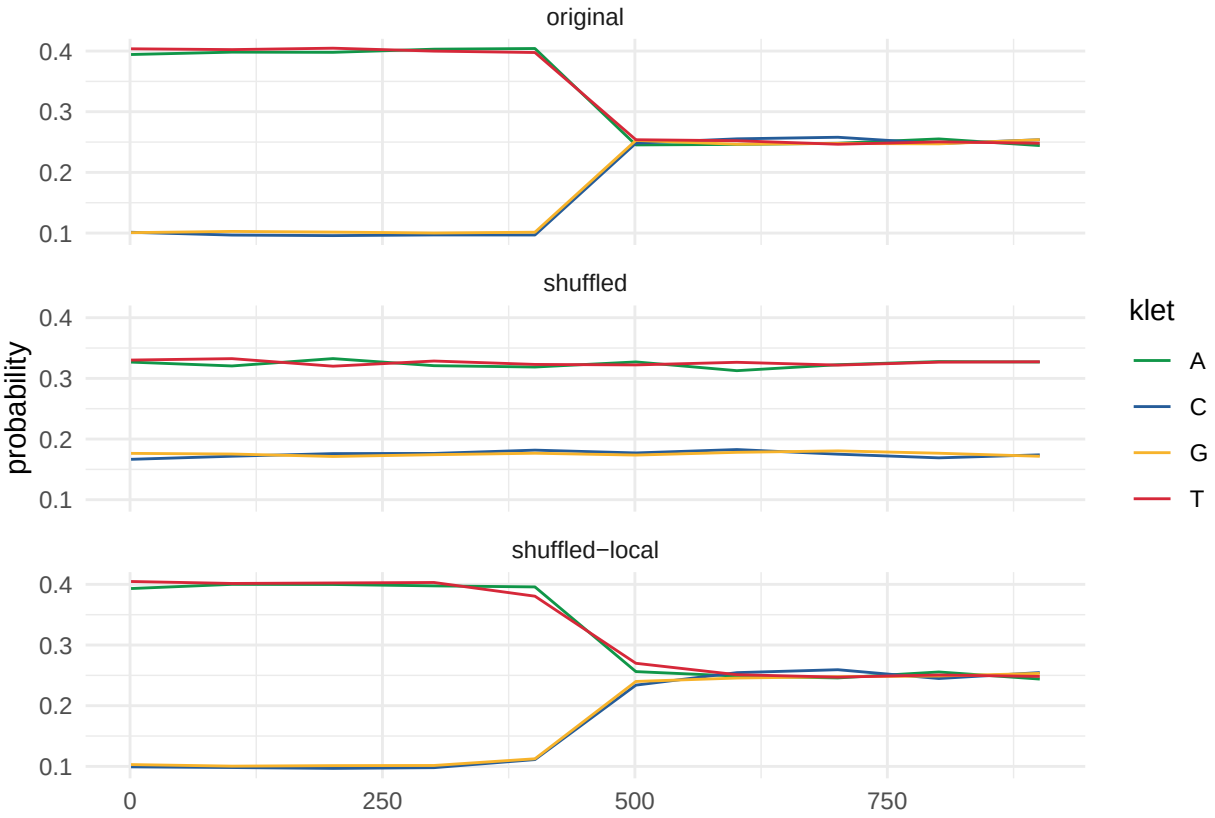
```
o.letter <- get_bkg(ArabidopsisPromoters, 1)
e.letter <- get_bkg(euler, 1)
m.letter <- get_bkg(markov, 1)
l.letter <- get_bkg(linear, 1)

data.frame(original=o.letter$count, euler=e.letter$count,
  markov=m.letter$count, linear=l.letter$count, row.names = DNA_BASES)
#>   original euler markov linear
#> A    17384 17384  17625  17384
#> C     8081  8081   8131   8081
#> G     7583  7583   7425   7583
#> T    16952 16952  16869  16952

o.counts <- get_bkg(ArabidopsisPromoters, 2)
e.counts <- get_bkg(euler, 2)
m.counts <- get_bkg(markov, 2)
l.counts <- get_bkg(linear, 2)

data.frame(original=o.counts$count, euler=e.counts$count,
  markov=m.counts$count, linear=l.counts$count,
  row.names = get_klets(DNA_BASES, 2))
#>    original euler markov linear
#> AA     6893  6893   6199   6452
#> AC     2614  2614   2808   2708
#> AG     2592  2592   2618   2592
#> AT     5276  5276   5985   5617
#> CA     3014  3014   2843   2878
#> CC     1376  1376   1356   1372
#> CG     1051  1051   1206   1146
#> CT     2621  2621   2719   2672
#> GA     2734  2734   2528   2667
#> GC     1104  1104   1240   1136
```

```
#> GG     1176  1176   1133   1206
#> GT     2561  2561   2517   2565
#> TA     4725  4725   6034   5372
#> TC     2977  2977   2720   2856
#> TG     2759  2759   2466   2631
#> TT     6477  6477   5628   6080
```

6 Local shuffling

If you have a fairly heterogeneous sequence and wish to preserve the presence of local "patches" of differential sequence composition, you can set `window = TRUE` in the `shuffle_sequences()` function. In the following example, the sequence of interest has an AT rich first half followed by a second half with an even background. The impact on this specific sequence composition is observed after regular and local shuffling, using the per-window functionality of `get_bkg()` (via `window = TRUE`). Fine-tune the window size and overlap between windows with `window.size` and `window.overlap`.

```
library(Biostrings)
library(universalmotif)
library(ggplot2)

myseq <- DNAStringSet(paste0(
  create_sequences(seqlen = 500, freqs = c(A=0.4, T=0.4, C=0.1, G=0.1)),
  create_sequences(seqlen = 500)
))

myseq_shuf <- shuffle_sequences(myseq)
myseq_shuf_local <- shuffle_sequences(myseq, window = TRUE)

myseq_bkg <- get_bkg(myseq, k = 1, window = TRUE)
myseq_shuf_bkg <- get_bkg(myseq_shuf, k = 1, window = TRUE)
myseq_shuf_local_bkg <- get_bkg(myseq_shuf_local, k = 1, window = TRUE)

myseq_bkg$group <- "original"
myseq_shuf_bkg$group <- "shuffled"
myseq_shuf_local_bkg$group <- "shuffled-local"

myseq_all <- as.data.frame(
  rbind(myseq_bkg, myseq_shuf_bkg, myseq_shuf_local_bkg)
)

ggplot(myseq_all, aes(x = start, y = probability, colour = klet)) +
  geom_line() +
  theme_minimal() +
  scale_colour_manual(values = universalmotif:::DNA_COLOURS) +
  xlab(element_blank()) +
  facet_wrap(~group, ncol = 1)
```



7 Miscellaneous string utilities

Since biological sequences are usually contained in `XStringSet` class objects, `get_bkg()` and `shuffle_sequences()` are designed to work with such objects. For cases when strings are not `XStringSet` objects, the following functions are available:

- `count_klets()`: alternative to `get_bkg()`
- `shuffle_string()`: alternative to `shuffle_sequences()`

```
library(universalmotif)

string <- "DASDSDDSASDSSA"

count_klets(string, 2)
#>   klets counts
#> 1    AA      0
#> 2    AD      0
#> 3    AS      2
#> 4    DA      1
#> 5    DD      1
#> 6    DS      3
#> 7    SA      2
#> 8    SD      3
#> 9    SS      1

shuffle_string(string, 2)
#> [1] "DSASDDSDASSDSA"
```

Finally, the `get_klets()` function can be used to get a list of all possible `k`-lets for any sequence alphabet:

```
library(universalmotif)

get_klets(c("A", "S", "D"), 2)
#> [1] "AA" "AS" "AD" "SA" "SS" "SD" "DA" "DS" "DD"
```

8 Scanning sequences for motifs

There are many motif-programs available with sequence scanning capabilities, such as HOMER and tools from the MEME suite. The `universalmotif` package does not aim to supplant these, but rather provide convenience functions for quickly scanning a few sequences without needing to leave the `R` environment. Furthermore, these functions allow for taking advantage of the higher-order (`multifreq`) motif format described here.

Two scanning-related functions are provided: `scan_sequences()` and `enrich_motifs()`. The latter simply runs `scan_sequences()` twice on a set of target and background sequences. Given a motif of length `n`, `scan_sequences()` considers every possible `n`-length subset in a sequence and scores it using the PWM format. If the match surpasses the minimum threshold, it is reported. This is case regardless of whether one is scanning with a regular motif, or using the higher-order (`multifreq`) motif format (the `multifreq` matrix is converted to a PWM).

Before scanning a set of sequences, one must first decide the minimum logodds threshold for retrieving matches. This decision is not always the same between scanning programs out in the wild, nor is it usually told to the user what the cutoff is or how it is decided. As a result, `universalmotif` aims to be as transparent as possible in this regard by allowing for complete control of the threshold. For more details on PWMs, see the introductory vignette.

One way is to set a cutoff between 0 and 1, then multiplying the highest possible PWM score to get a threshold. The `matchPWM()` function from the `Biostrings` package for example uses a default of 0.8 (shown as `"80%"`). This is quite arbitrary of course, and every motif will end up with a different threshold. For high information content motifs, there is really no right or wrong threshold, as they tend to have fewer non-specific positions. This means that incorrect letters in a match will be more punishing. To illustrate this, contrast the following PWMs:

```
library(universalmotif)
m1 <- create_motif("TATATATATA", nsites = 50, type = "PWM", pseudocount = 1)
m2 <- 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)
m2 <- create_motif(m2, alphabet = "DNA", type = "PWM")
m1["motif"]
#>           T         A         T         A         T         A         T
#> A -5.672425  1.978626 -5.672425  1.978626 -5.672425  1.978626 -5.672425
#> C -5.672425 -5.672425 -5.672425 -5.672425 -5.672425 -5.672425 -5.672425
#> G -5.672425 -5.672425 -5.672425 -5.672425 -5.672425 -5.672425 -5.672425
#> T  1.978626 -5.672425  1.978626 -5.672425  1.978626 -5.672425  1.978626
#>           A         T         A
#> A  1.978626 -5.672425  1.978626
#> C -5.672425 -5.672425 -5.672425
#> G -5.672425 -5.672425 -5.672425
#> T -5.672425  1.978626 -5.672425
m2["motif"]
#>           S          H          C          N          N          N
```

```
#> A -1.3219281  0.09667602 -0.12029423 -0.3959287  0.2141248  0.1491434
#> C  0.5260688  0.19976951  1.02856915  0.6040713 -0.1202942 -0.6582115
#> G  0.8479969 -2.33628339 -3.64385619 -0.9434165  0.1110313  0.5897160
#> T -1.4739312  0.66371661 -0.05889369  0.2630344 -0.2515388 -0.4102840
#>           R          N          N           V
#> A  1.0430687 -1.0732490  0.4436067  0.04222824
#> C -0.5418938 -0.2658941 -0.1202942  0.51171352
#> G  0.0710831  0.5897160 -1.0588937  0.29598483
#> T -2.3074285  0.2486791  0.3103401 -1.65821148
```

In the first example, sequences which do not have a matching base in every position are punished heavily. The maximum logodds score in this case is approximately 20, and for each incorrect position the score is reduced approximately by 5.7. This means that a threshold of zero would allow for at most three mismatches. At this point, it is up to you how many mismatches you would deem appropriate.

This thinking becomes impossible for the second example. In this case, mismatches are much less punishing, to the point that one could ask: what even constitutes a mismatch? The answer to this question is much more difficult in such cases. An alternative to manually deciding upon a threshold is to instead start with maximum P-value one would consider appropriate for a match. If, say, we want matches with a P-value of at most 0.001, then we can use `motif_pvalue()` to calculate the appropriate threshold (see the comparisons and P-values vignette for details on motif P-values).

```
motif_pvalue(m2, pvalue = 0.001)
#> [1] 4.8617
```

Furthermore, the `scan_sequences()` function offers the ability to scan using the `multifreq` slot, if available. This allows to take into account inter-positional dependencies, and get matches which more faithfully represent the original sequences from which the motif originated.

```
library(universalmotif)
library(Biostrings)
data(ArabidopsisPromoters)

## A 2-letter example:

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

Regular scanning:

```
scan_sequences(motif.k2, ArabidopsisPromoters, RC = TRUE,
               threshold = 0.9, threshold.type = "logodds")
#> Note: found -Inf values in motif PWM(s), adding a pseudocount. Set
#>   `allow.nonfinite = TRUE` to prevent this behaviour.
#> DataFrame with 94 rows and 12 columns
#>           motif   motif.i    sequence     start      stop     score       match
#>     <character> <integer> <character> <integer> <integer> <numeric> <character>
#> 1         motif         1   AT4G28150       621       627      9.08     CTAAACC
#> 2         motif         1   AT1G19380       139       145      9.08     CTTATCC
#> 3         motif         1   AT1G19380       204       210      9.08     CTAAACC
#> 4         motif         1   AT1G03850       203       209      9.08     CTAATCC
#> 5         motif         1   AT5G01810       821       827      9.08     CATATCC
#> ...         ...       ...         ...       ...       ...       ...         ...
#> 90        motif         1   AT5G22690        58        52      9.08     CATTACC
#> 91        motif         1   AT1G05670       706       700      9.08     CTTTACC
```

```
#> 92        motif         1   AT1G06160       498       492      9.08     CTAAACC
#> 93        motif         1   AT5G24660       146       140      9.08     CATTACC
#> 94        motif         1   AT3G19200       421       415      9.08     CTAAACC
#>     thresh.score min.score max.score score.pct      strand
#>        <numeric> <numeric> <numeric> <numeric> <character>
#> 1          8.172   -19.649      9.08       100           +
#> 2          8.172   -19.649      9.08       100           +
#> 3          8.172   -19.649      9.08       100           +
#> 4          8.172   -19.649      9.08       100           +
#> 5          8.172   -19.649      9.08       100           +
#> ...          ...       ...       ...       ...         ...
#> 90         8.172   -19.649      9.08       100           -
#> 91         8.172   -19.649      9.08       100           -
#> 92         8.172   -19.649      9.08       100           -
#> 93         8.172   -19.649      9.08       100           -
#> 94         8.172   -19.649      9.08       100           -
```

Using 2-letter information to scan:

```
scan_sequences(motif.k2, ArabidopsisPromoters, use.freq = 2, RC = TRUE,
               threshold = 0.9, threshold.type = "logodds")
#> Note: found -Inf values in motif PWM(s), adding a pseudocount. Set
#>   `allow.nonfinite = TRUE` to prevent this behaviour.
#> DataFrame with 8 rows and 12 columns
#>         motif   motif.i    sequence     start      stop     score       match
#>   <character> <integer> <character> <integer> <integer> <numeric> <character>
#> 1       motif         1   AT4G12690       938       943   17.827      CAAAAC
#> 2       motif         1   AT2G37950       751       756   17.827      CAAAAC
#> 3       motif         1   AT1G49840       959       964   17.827      CTTTTC
#> 4       motif         1   AT1G77210       184       189   17.827      CAAAAC
#> 5       motif         1   AT1G77210       954       959   17.827      CAAAAC
#> 6       motif         1   AT3G57640       917       922   17.827      CTTTTC
#> 7       motif         1   AT4G14365       977       982   17.827      CTTTTC
#> 8       motif         1   AT1G19510       960       965   17.827      CTTTTC
#>   thresh.score min.score max.score score.pct      strand
#>      <numeric> <numeric> <numeric> <numeric> <character>
#> 1      16.0443   -16.842    17.827       100           +
#> 2      16.0443   -16.842    17.827       100           +
#> 3      16.0443   -16.842    17.827       100           +
#> 4      16.0443   -16.842    17.827       100           +
#> 5      16.0443   -16.842    17.827       100           +
#> 6      16.0443   -16.842    17.827       100           +
#> 7      16.0443   -16.842    17.827       100           +
#> 8      16.0443   -16.842    17.827       100           +
```

Furthermore, sequence scanning can be further refined to avoid overlapping hits. Consider:

```
motif <- create_motif("AAAAAA")

## Leave in overlapping hits:

scan_sequences(motif, ArabidopsisPromoters, RC = TRUE, threshold = 0.9,
               threshold.type = "logodds")
#> Note: found -Inf values in motif PWM(s), adding a pseudocount. Set
#>   `allow.nonfinite = TRUE` to prevent this behaviour.
```

```
#> DataFrame with 491 rows and 12 columns
#>           motif   motif.i    sequence     start      stop     score       match
#>     <character> <integer> <character> <integer> <integer> <numeric> <character>
#> 1         motif         1   AT4G28150       419       424    11.934      AAAAAA
#> 2         motif         1   AT1G19380       867       872    11.934      AAAAAA
#> 3         motif         1   AT1G19380       868       873    11.934      AAAAAA
#> 4         motif         1   AT1G03850       243       248    11.934      AAAAAA
#> 5         motif         1   AT1G03850       735       740    11.934      AAAAAA
#> ...         ...       ...         ...       ...       ...       ...         ...
#> 487       motif         1   AT3G19200       246       241    11.934      AAAAAA
#> 488       motif         1   AT3G19200       247       242    11.934      AAAAAA
#> 489       motif         1   AT3G19200       248       243    11.934      AAAAAA
#> 490       motif         1   AT3G19200       668       663    11.934      AAAAAA
#> 491       motif         1   AT3G19200       669       664    11.934      AAAAAA
#>     thresh.score min.score max.score score.pct      strand
#>        <numeric> <numeric> <numeric> <numeric> <character>
#> 1        10.7406   -39.948    11.934       100           +
#> 2        10.7406   -39.948    11.934       100           +
#> 3        10.7406   -39.948    11.934       100           +
#> 4        10.7406   -39.948    11.934       100           +
#> 5        10.7406   -39.948    11.934       100           +
#> ...          ...       ...       ...       ...         ...
#> 487      10.7406   -39.948    11.934       100           -
#> 488      10.7406   -39.948    11.934       100           -
#> 489      10.7406   -39.948    11.934       100           -
#> 490      10.7406   -39.948    11.934       100           -
#> 491      10.7406   -39.948    11.934       100           -

## Only keep the highest scoring hit amongst overlapping hits:

scan_sequences(motif, ArabidopsisPromoters, RC = TRUE, threshold = 0.9,
               threshold.type = "logodds", no.overlaps = TRUE)
#> Note: found -Inf values in motif PWM(s), adding a pseudocount. Set
#>   `allow.nonfinite = TRUE` to prevent this behaviour.
#> DataFrame with 229 rows and 12 columns
#>           motif   motif.i    sequence     start      stop     score       match
#>     <character> <integer> <character> <integer> <integer> <numeric> <character>
#> 1         motif         1   AT4G28150       419       424    11.934      AAAAAA
#> 2         motif         1   AT1G19380       867       872    11.934      AAAAAA
#> 3         motif         1   AT1G03850       243       248    11.934      AAAAAA
#> 4         motif         1   AT1G03850       735       740    11.934      AAAAAA
#> 5         motif         1   AT5G01810       950       955    11.934      AAAAAA
#> ...         ...       ...         ...       ...       ...       ...         ...
#> 225       motif         1   AT1G05670        78        73    11.934      AAAAAA
#> 226       motif         1   AT1G05670        85        80    11.934      AAAAAA
#> 227       motif         1   AT1G06160       404       399    11.934      AAAAAA
#> 228       motif         1   AT3G19200       246       241    11.934      AAAAAA
#> 229       motif         1   AT3G19200       668       663    11.934      AAAAAA
#>     thresh.score min.score max.score score.pct      strand
#>        <numeric> <numeric> <numeric> <numeric> <character>
#> 1        10.7406   -39.948    11.934       100           +
#> 2        10.7406   -39.948    11.934       100           +
#> 3        10.7406   -39.948    11.934       100           +
```

```
#> 4       10.7406   -39.948    11.934       100          +
#> 5       10.7406   -39.948    11.934       100          +
#> ...         ...       ...       ...       ...        ...
#> 225     10.7406   -39.948    11.934       100          -
#> 226     10.7406   -39.948    11.934       100          -
#> 227     10.7406   -39.948    11.934       100          -
#> 228     10.7406   -39.948    11.934       100          -
#> 229     10.7406   -39.948    11.934       100          -
```

Finally, the results can be returned as a `GRanges` object for further manipulation:

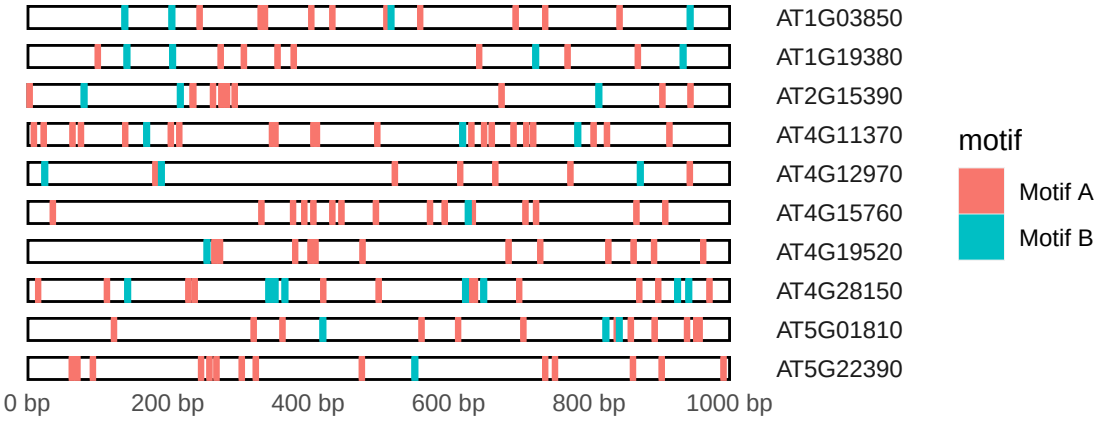
```
scan_sequences(motif.k2, ArabidopsisPromoters, RC = TRUE,
               threshold = 0.9, threshold.type = "logodds",
               return.granges = TRUE)
#> Note: found -Inf values in motif PWM(s), adding a pseudocount. Set
#>   `allow.nonfinite = TRUE` to prevent this behaviour.
#> GRanges object with 94 ranges and 8 metadata columns:
#>         seqnames    ranges strand |       motif   motif.i     score       match
#>            <Rle> <IRanges>  <Rle> | <character> <integer> <numeric> <character>
#>    [1] AT4G28150   621-627      + |       motif         1      9.08     CTAAACC
#>    [2] AT1G19380   139-145      + |       motif         1      9.08     CTTATCC
#>    [3] AT1G19380   204-210      + |       motif         1      9.08     CTAAACC
#>    [4] AT1G03850   203-209      + |       motif         1      9.08     CTAATCC
#>    [5] AT5G01810   821-827      + |       motif         1      9.08     CATATCC
#>    ...       ...       ...    ... .         ...       ...       ...         ...
#>   [90] AT5G22690     52-58      - |       motif         1      9.08     CATTACC
#>   [91] AT1G05670   700-706      - |       motif         1      9.08     CTTTACC
#>   [92] AT1G06160   492-498      - |       motif         1      9.08     CTAAACC
#>   [93] AT5G24660   140-146      - |       motif         1      9.08     CATTACC
#>   [94] AT3G19200   415-421      - |       motif         1      9.08     CTAAACC
#>        thresh.score min.score max.score score.pct
#>           <numeric> <numeric> <numeric> <numeric>
#>    [1]       8.172   -19.649      9.08       100
#>    [2]       8.172   -19.649      9.08       100
#>    [3]       8.172   -19.649      9.08       100
#>    [4]       8.172   -19.649      9.08       100
#>    [5]       8.172   -19.649      9.08       100
#>    ...         ...       ...       ...       ...
#>   [90]       8.172   -19.649      9.08       100
#>   [91]       8.172   -19.649      9.08       100
#>   [92]       8.172   -19.649      9.08       100
#>   [93]       8.172   -19.649      9.08       100
#>   [94]       8.172   -19.649      9.08       100
#>   -------
#>   seqinfo: 50 sequences from an unspecified genome
```

9 Visualizing motif hits across sequences

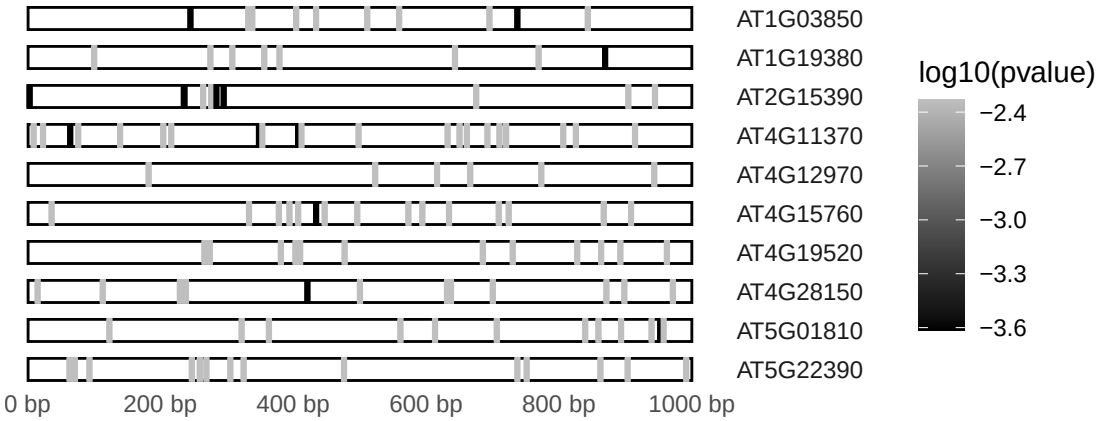
Using the `ggbio` package, it is rather trivial to generate nice visualizations of the output of `scan_sequences()`. This requires having the `GenomicRanges` and `ggbio` packages installed, and outputting the `scan_sequences()` result as a `GRanges` object (via `return.granges = TRUE`).

```
library(universalmotif)
library(GenomicRanges)
```

```
library(ggbio)

data(ArabidopsisPromoters)

motif1 <- create_motif("AAAAAA", name = "Motif A")
motif2 <- create_motif("CWWWWCC", name = "Motif B")

res <- scan_sequences(c(motif1, motif2), ArabidopsisPromoters[1:10],
  return.granges = TRUE, calc.pvals = TRUE, no.overlaps = TRUE)
#> Note: found -Inf values in motif PWM(s), adding a pseudocount. Set
#>   `allow.nonfinite = TRUE` to prevent this behaviour.

## Just plot the motif hits:
autoplot(res, layout = "karyogram", aes(fill = motif, color = motif)) +
  theme(
    strip.background = element_rect(fill = NA, colour = NA),
    panel.background = element_rect(fill = NA, colour = NA)
  )
#> Scale for 'x' is already present. Adding another scale for 'x', which will
#> replace the existing scale.
#> Scale for 'x' is already present. Adding another scale for 'x', which will
#> replace the existing scale.
```




```
## Plot Motif A hits by P-value:
autoplot(res[res$motif.i == 1, ], layout = "karyogram",
  aes(fill = log10(pvalue), colour = log10(pvalue))) +
  scale_fill_gradient(low = "black", high = "grey75") +
  scale_colour_gradient(low = "black", high = "grey75") +
  theme(
    strip.background = element_rect(fill = NA, colour = NA),
    panel.background = element_rect(fill = NA, colour = NA)
  )
#> Scale for 'x' is already present. Adding another scale for 'x', which will
#> replace the existing scale.
#> Scale for 'x' is already present. Adding another scale for 'x', which will
#> replace the existing scale.
```



10 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, RC = TRUE)
#> DataFrame with 1 row and 11 columns
#>             motif   motif.i target.hits target.seq.hits target.seq.count
#>       <character> <integer>   <integer>       <integer>        <integer>
#> 1 YTTTYTTTTTYTTTY         1         652              50               50
#>    bkg.hits bkg.seq.hits bkg.seq.count        Pval        Qval        Eval
#>   <integer>    <integer>     <integer>   <numeric>   <numeric>   <numeric>
#> 1       248           46            50 4.75618e-43 4.75618e-43 9.51237e-43
```

Here we can see that the motif is significantly enriched in the target sequences. The `Pval` was calculated by calling `stats::fisher.test()`.

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.)

11 Gapped motifs

`universalmotif` class motifs can be gapped, which can be used by `scan_sequences()` and `enrich_motifs()`. Note that gapped motif support is currently limited to these two functions. All other functions will ignore the gap information, and even discard them in functions such as `merge_motifs()`.

First, obtain the component motifs:

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

```
m1 <- create_motif("TTTATAT", name = "PartA")
m2 <- create_motif("GGTTCGA", name = "PartB")
```

Then, combine them and add the desired gap. In this case, a gap will be added between the two motifs which can range in size from 4-6 bases.

```
m <- cbind(m1, m2)
m <- add_gap(m, gaploc = ncol(m1), mingap = 4, maxgap = 6)
m
#>
#>        Motif name:   PartA/PartB
#>          Alphabet:   DNA
#>              Type:   PCM
#>           Strands:   +-
#>          Total IC:   28
#>       Pseudocount:   0
#>         Consensus:   TTTATAT..GGTTCGA
#>     Gap locations:   7-8
#>         Gap sizes:   4-6
#>
#>   T T T A T A T    G G T T C G A
#> A 0 0 0 1 0 1 0 .. 0 0 0 0 0 0 1
#> C 0 0 0 0 0 0 0 .. 0 0 0 0 1 0 0
#> G 0 0 0 0 0 0 0 .. 1 1 0 0 0 1 0
#> T 1 1 1 0 1 0 1 .. 0 0 1 1 0 0 0
```

Now, it can be used directly in `scan_sequences()` or `enrich_motifs()`:

```
scan_sequences(m, ArabidopsisPromoters, threshold = 0.4, threshold.type = "logodds")
#> Note: found -Inf values in motif PWM(s), adding a pseudocount. Set
#>   `allow.nonfinite = TRUE` to prevent this behaviour.
#> DataFrame with 75 rows and 12 columns
#>           motif   motif.i    sequence     start      stop     score
#>     <character> <integer> <character> <integer> <integer> <numeric>
#> 1   PartA/PartB         1   AT4G19520       484       501    11.918
#> 2   PartA/PartB         1   AT5G20200       731       748    11.918
#> 3   PartA/PartB         1   AT2G04025       168       185    11.918
#> 4   PartA/PartB         1   AT1G06160       144       161    11.918
#> 5   PartA/PartB         1   AT1G03850       376       394    11.178
#> ...         ...       ...         ...       ...       ...       ...
#> 71  PartA/PartB         1   AT4G33970       272       291    11.428
#> 72  PartA/PartB         1   AT3G15610       402       421    11.428
#> 73  PartA/PartB         1   AT2G17450       233       252    11.428
#> 74  PartA/PartB         1   AT2G17450       891       910    11.428
#> 75  PartA/PartB         1   AT2G24240       355       374    11.428
#>                    match thresh.score min.score max.score score.pct      strand
#>              <character>    <numeric> <numeric> <numeric> <numeric> <character>
#> 1     GTTATAT....GATTCTA      11.1384   -93.212    27.846   42.7997           +
#> 2     TTCATTT....GGTTAGA      11.1384   -93.212    27.846   42.7997           +
#> 3     TGTTTAT....GGTTCGG      11.1384   -93.212    27.846   42.7997           +
#> 4     TTTATGT....GGTTTGT      11.1384   -93.212    27.846   42.7997           +
#> 5    TATATGT.....GGTGCAA      11.1384   -93.212    27.846   40.1422           +
#> ...                  ...          ...       ...       ...       ...         ...
#> 71  TTTACCA......AGTTCGA      11.1384   -93.212    27.846     41.04           +
#> 72  TTTAAGT......AGTTCTA      11.1384   -93.212    27.846     41.04           +
```

```
#> 73  TTTTTAT......TGATAGA      11.1384   -93.212    27.846     41.04          +
#> 74  TTATTAT......GATTTGA      11.1384   -93.212    27.846     41.04          +
#> 75  CTTATAT......GGATTGT      11.1384   -93.212    27.846     41.04          +
```

12 Testing for motif positional preferences in sequences

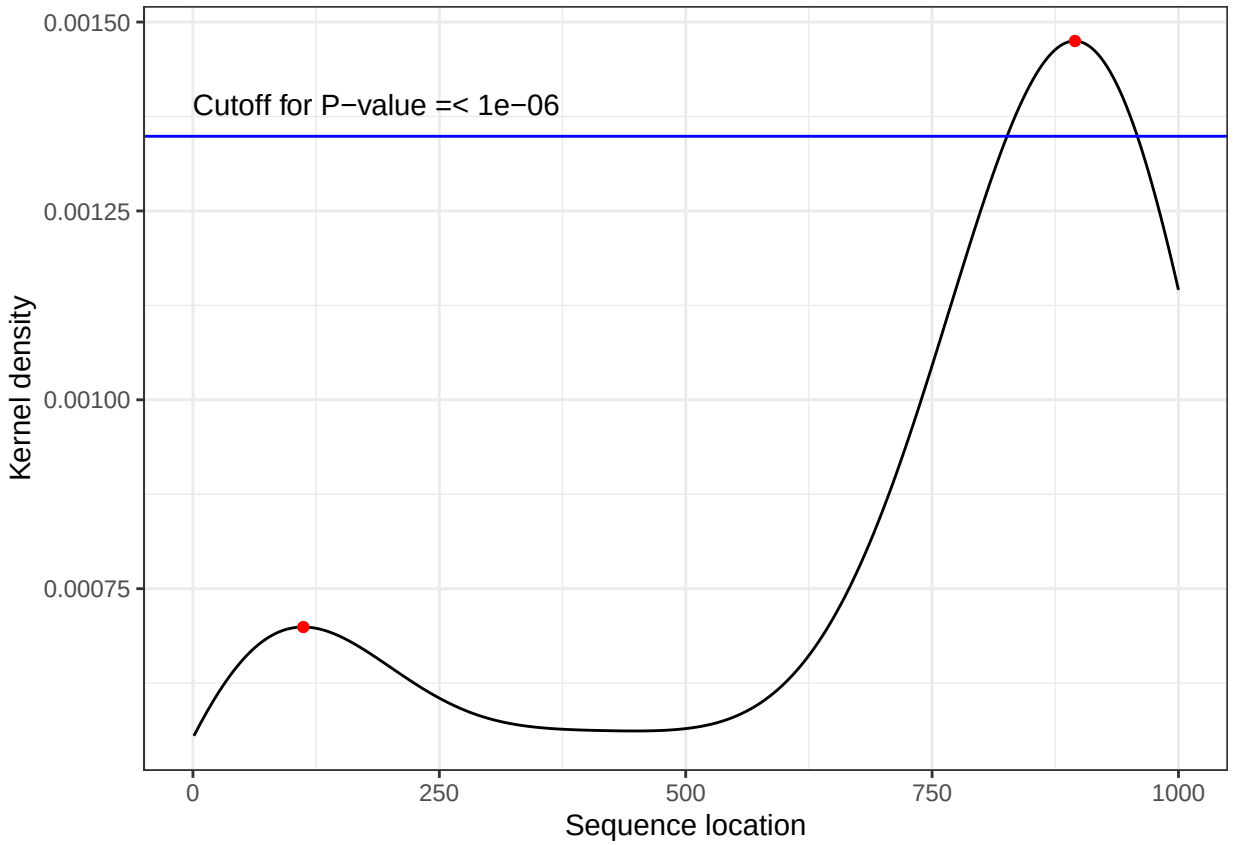
The `universalmotif` package provides the `motif_peaks()` function, which can test for positionally preferential motif sites in a set of sequences. This can be useful, for example, when trying to determine whether a certain transcription factor binding site is more often than not located at a certain distance from the transcription start site (TSS). The `motif_peaks()` function finds density peaks in the input data, then creates a null distribution from randomly generated peaks to calculate peak P-values.

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

hits <- scan_sequences(ArabidopsisMotif, ArabidopsisPromoters, RC = FALSE)

res <- motif_peaks(hits$start,
                   seq.length = unique(width(ArabidopsisPromoters)),
                   seq.count = length(ArabidopsisPromoters))

## Significant peaks:
res$Peaks
#> DataFrame with 1 row and 2 columns
#>        Peak        Pval
#>   <numeric>   <numeric>
#> 1       895 1.31625e-11
```

Using the datasets provided in this package, a significant motif peak was found about 100 bases away from the TSS. If you'd like to simply know the locations of any peaks, this can be done by setting `max.p = 1`.

The function can also output a plot:

```
res$Plot
```



In this plot, red dots are used to indicate density peaks and the blue line shows the P-value cutoff.

13 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.

```r
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 = 1000, downstream = 0)

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 sequences) 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). Doing the latter gives you the option of generating higher order representations for the imported MEME motifs as shown here:

```
motifs <- run_meme(sequences)
motifs.k23 <- mapply(add_multifreq, motifs$motifs, motifs$sites)
```

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 4.1.0 (2021-05-18)
#> Platform: x86_64-pc-linux-gnu (64-bit)
#> Running under: Ubuntu 20.04.2 LTS
#>
#> Matrix products: default
#> BLAS:   /home/biocbuild/bbs-3.13-bioc/R/lib/libRblas.so
#> LAPACK: /home/biocbuild/bbs-3.13-bioc/R/lib/libRlapack.so
#>
#> 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
#>
#> attached base packages:
#> [1] stats4    parallel  stats     graphics  grDevices utils     datasets
#> [8] methods   base
#>
#> other attached packages:
#>  [1] ggbio_1.40.0          TFBSTools_1.30.0      cowplot_1.1.1
#>  [4] dplyr_1.0.7           ggtree_3.0.2          ggplot2_3.3.5
#>  [7] MotifDb_1.34.0        GenomicRanges_1.44.0  Biostrings_2.60.2
#> [10] GenomeInfoDb_1.28.1   XVector_0.32.0        IRanges_2.26.0
```

```
#> [13] S4Vectors_0.30.0      BiocGenerics_0.38.0   universalmotif_1.10.2
#>
#> loaded via a namespace (and not attached):
#>   [1] backports_1.2.1             Hmisc_4.5-0
#>   [3] BiocFileCache_2.0.0         plyr_1.8.6
#>   [5] lazyeval_0.2.2              splines_4.1.0
#>   [7] BiocParallel_1.26.1         digest_0.6.27
#>   [9] ensembldb_2.16.4            htmltools_0.5.1.1
#>  [11] GO.db_3.13.0                fansi_0.5.0
#>  [13] magrittr_2.0.1              checkmate_2.0.0
#>  [15] memoise_2.0.0               BSgenome_1.60.0
#>  [17] grImport2_0.2-0             cluster_2.1.2
#>  [19] tzdb_0.1.2                  readr_2.0.0
#>  [21] annotate_1.70.0             matrixStats_0.60.0
#>  [23] R.utils_2.10.1              prettyunits_1.1.1
#>  [25] jpeg_0.1-9                  colorspace_2.0-2
#>  [27] rappdirs_0.3.3              blob_1.2.2
#>  [29] xfun_0.24                   crayon_1.4.1
#>  [31] RCurl_1.98-1.3              jsonlite_1.7.2
#>  [33] graph_1.70.0                TFMPvalue_0.0.8
#>  [35] VariantAnnotation_1.38.0    survival_3.2-11
#>  [37] ape_5.5                     glue_1.4.2
#>  [39] gtable_0.3.0                zlibbioc_1.38.0
#>  [41] DelayedArray_0.18.0         scales_1.1.1
#>  [43] DBI_1.1.1                   GGally_2.1.2
#>  [45] Rcpp_1.0.7                  progress_1.2.2
#>  [47] xtable_1.8-4                htmlTable_2.2.1
#>  [49] tidytree_0.3.4              foreign_0.8-81
#>  [51] bit_4.0.4                   OrganismDbi_1.34.0
#>  [53] Formula_1.2-4               htmlwidgets_1.5.3
#>  [55] httr_1.4.2                  RColorBrewer_1.1-2
#>  [57] ellipsis_0.3.2              pkgconfig_2.0.3
#>  [59] reshape_0.8.8               XML_3.99-0.6
#>  [61] R.methodsS3_1.8.1           farver_2.1.0
#>  [63] dbplyr_2.1.1                nnet_7.3-16
#>  [65] ggseqlogo_0.1               utf8_1.2.2
#>  [67] tidyselect_1.1.1            labeling_0.4.2
#>  [69] rlang_0.4.11                reshape2_1.4.4
#>  [71] AnnotationDbi_1.54.1        munsell_0.5.0
#>  [73] tools_4.1.0                 cachem_1.0.5
#>  [75] DirichletMultinomial_1.34.0 generics_0.1.0
#>  [77] RSQLite_2.2.7               ade4_1.7-17
#>  [79] evaluate_0.14               stringr_1.4.0
#>  [81] fastmap_1.1.0               yaml_2.2.1
#>  [83] knitr_1.33                  bit64_4.0.5
#>  [85] caTools_1.18.2              purrr_0.3.4
#>  [87] AnnotationFilter_1.16.0     KEGGREST_1.32.0
#>  [89] splitstackshape_1.4.8       RBGL_1.68.0
#>  [91] nlme_3.1-152                R.oo_1.24.0
#>  [93] poweRlaw_0.70.6             aplot_0.0.6
#>  [95] xml2_1.3.2                  pracma_2.3.3
#>  [97] biomaRt_2.48.2              rstudioapi_0.13
#>  [99] compiler_4.1.0              filelock_1.0.2
#> [101] curl_4.3.2                  png_0.1-7
```

```
#> [103] treeio_1.16.1               tibble_3.1.3
#> [105] stringi_1.7.3               highr_0.9
#> [107] GenomicFeatures_1.44.0      lattice_0.20-44
#> [109] ProtGenerics_1.24.0         CNEr_1.28.0
#> [111] Matrix_1.3-4                vctrs_0.3.8
#> [113] pillar_1.6.2                lifecycle_1.0.0
#> [115] BiocManager_1.30.16         data.table_1.14.0
#> [117] bitops_1.0-7                patchwork_1.1.1
#> [119] rtracklayer_1.52.0          R6_2.5.0
#> [121] BiocIO_1.2.0                latticeExtra_0.6-29
#> [123] bookdown_0.22               gridExtra_2.3
#> [125] motifStack_1.36.0           dichromat_2.0-0
#> [127] MASS_7.3-54                 gtools_3.9.2
#> [129] assertthat_0.2.1            seqLogo_1.58.0
#> [131] SummarizedExperiment_1.22.0 rjson_0.2.20
#> [133] withr_2.4.2                 GenomicAlignments_1.28.0
#> [135] Rsamtools_2.8.0             GenomeInfoDbData_1.2.6
#> [137] hms_1.1.0                   grid_4.1.0
#> [139] rpart_4.1-15                tidyr_1.1.3
#> [141] rmarkdown_2.9               rvcheck_0.1.8
#> [143] MatrixGenerics_1.4.1        biovizBase_1.40.0
#> [145] Biobase_2.52.0              base64enc_0.1-3
#> [147] tinytex_0.32                restfulr_0.0.13
```

References

Altschul, Stephen F., and Bruce W. Erickson. 1985. "Significance of Nucleotide Sequence Alignments: A Method for Random Sequence Permutation That Preserves Dinucleotide and Codon Usage." *Molecular Biology and Evolution* 2 (6): 526–38.

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.

Fitch, Walter M. 1983. "Random Sequences." *Journal of Molecular Biology* 163 (2): 171–76.

Jiang, M., J. Anderson, J. Gillespie, and M. Mayne. 2008. "uShuffle: A Useful Tool for Shuffling Biological Sequences While Preserving K-Let Counts." *BMC Bioinformatics* 9 (192).

Propp, J.G., and D.W. Wilson. 1998. "How to Get a Perfectly Random Sample from a Generic Markov Chain and Generate a Random Spanning Tree of a Directed Graph." *Journal of Algorithms* 27: 170–217.