

Sequence manipulation and scanning

Benjamin Jean-Marie Tremblay*

17 October 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

1	Introduction	1
2	Basic sequence handling	2
2.1	Creating random sequences	2
2.2	Calculating sequence background	3
2.3	Clustering sequences by k-let composition	4
3	Shuffling	5
3.1	Shuffling sequences	5
3.2	Local shuffling	7
4	Sequence scanning and enrichment	8
4.1	Choosing a logodds threshold	8
4.2	Regular and higher order scanning	12
4.3	Visualizing motif hits across sequences	16
4.4	Enrichment analyses	20
4.5	Fixed and variable-length gapped motifs	21
4.6	Detecting low complexity regions and sequence masking	22
5	Motif discovery with MEME	24
6	Miscellaneous string utilities	26
	Session info	27
	References	29

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

*benjamin.tremblay@uwaterloo.ca

functions, see the motif manipulation vignette. For a discussion on motif comparisons and P-values, see the motif comparisons and P-values vignette.

2 Basic sequence handling

2.1 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 AATTATTGGTAAGAAAGTCAGGCTGAGGAGAAT...TGTGTAGGCAGAAAGAAATTAAGAACGGAAAC
#> [2] 100 GAATCAATAGAACCTGTATTTCTGTTTAAAGCT...TAGTGTCGATTAATACAAGACTAAATACCTCG
#> [3] 100 CCAGCGTTCCGGTTGGAACGTCAGGTGTTGTC...CATAGAGGTTATATCATGTAACCTCACTTAAA
#> [4] 100 AAGAGGCTGGAACATTTCTTTTGTGTGTCCGA...TACATGCTATCTAGTAGCTATTCGAACATCAA
#> [5] 100 TACTGTTACTGGAATACATTACGCTTATGAACA...GTTACTTAAGTTGGGATGTTTGTCCAAGCTTA
#> ...
#> [496] 100 AGTATATCGACATCCAAAGGTATAAGCATGTCC...TTTAAACCCTGATTAAGCGAATTAAAAATCTTT
#> [497] 100 ATGATGAATGAACTTTAAACAAAACATGTTAGT...ACGATCAATAGCTGATCTAAACAGTGACTAAT
#> [498] 100 CACTAACATTCGACGTCAACAGGTCCACTGGGT...TCGATGACAGTCAGTTAAGATAGTCTAGCGCT
#> [499] 100 TATGCTTGCTGCGAGATGTAACCTACTGTAATC...CCTTTCTGTTGTAGACTTGAACGGAAGTTGCG
#> [500] 100 TATAATAGTGATTGCGGTAGGAACGGATTGGAC...CGTTAGGTTTCGAGACAAGCATGAGTTTATAG

## Amino acid:

create_sequences(alphabet = "AA")
#> AAStringSet object of length 100:
#>      width seq
#> [1] 100 NIVVQLKAWVPFQCPAPSTALLGLDHEGCVAAM...TGVFEMSEPSNYMVSHWRLQTLQEMVKLAQD
#> [2] 100 SIEAERKIHRCQYCRELDTHSKQREIEATCSG...LDGNEKYGEFDPQELIMGTAVPKIPGEVMLDG
#> [3] 100 AFCAWMHTECRLQSFQTGGLLYIKYELEGLDI...ESIRSGCDGMFHPFGDPCQVLHDNKLFPFNQ
#> [4] 100 YCTKKSEFIAEVNCREIMSLRQIFPCTYWDNM...SWLCYKTAVHVFIDGALSCPAKFSEWPFDDVK
#> [5] 100 WWYWIDRFYHWLLIHSKEPHCQCQEANTGLAWN...WINVQTWKQINNVAKSWNMKFKHLVHDDTFIY
#> ...
#> [96] 100 RDPFFTLTACYMKTGKDPVCVKNSIMKCGQY...GNLFWMCDKCKINYDRPGHLYWSAGDKPVAET
#> [97] 100 HNALQHTAVNYPWNPGCWNFTCWSSRLVCKTQW...CRIYTFKVDDSPYPWVMETAVQVWEGPMGWEM
#> [98] 100 TQSTVICGPINRRERWFEATDTARIYICMSYCW...IIEPHCHLWEWIYAFYMIKHLKGRHTCQNRST
```

```
#> [99] 100 DVEIPWHFWKMCICHFIGEGTYPANAVPETAT...HKHWKFAMTDERRCRNQNGASWPAHKSMPDY
#> [100] 100 ARQRLLCRPHCTYTVFQGESDDTIQSCSLYMRV...LMTLNKNFLNIQYFHMKCIKVKAMGINIRWGH

## Any set of characters can be used

create_sequences(alphabet = paste0(letters, collapse = ""))
#> BStringSet object of length 100:
#>      width seq
#> [1] 100 hrkyxmsjbfyypvugoqaqoxebgazuteukz...yfqmpxuznsjbcroeavdoaanngiurduv
#> [2] 100 veikqukkrdokekyhjknzdubyoaleqtdfz...rcmccifjxaurfmqjacjvpakeowpfbhdr
#> [3] 100 ddekahdobuuewumdlewkcrcvncuzjmylc...vftokmfrfnmhzyinxiixjtlmytqiqbfr
#> [4] 100 yapuxqayznkfthcpecutedqiunfqxyirb...egfqjwtsbwgubxpfwwomhelrtuwtgtmg
#> [5] 100 qaqqwdlnhigexlscxbxtodyutckojhybn...eztttlysaqrwifywepkumdutevzjiscq
#> ...      ...
#> [96] 100 ggfwuqzfulneduzekdfttpclrnfeauenx...eatxppbmeqreytqsnfurbjofaalwdgum
#> [97] 100 wduncwjdmccbsmeahgvjppgizfamanxt...xzenffckrzqlgljwdfjuenjjlcggqus
#> [98] 100 mcsntywoeuzrnhrdwgtgcuncxtslhquly...ivysobeeompdsfztngyxbwfbotlzpatsx
#> [99] 100 eitltspjpjavhdmogzsnlzppgpufdkjoc...qsjuotajpzjdowidqzhqurmpxlhtqwdi
#> [100] 100 rduxwutchljrwhgquwhhknepustavpso...nlbipfeiiesyynyiwnpsfcmnobyux
```

2.2 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      2500    0.2500000
#> 2          C      2503    0.2503000
#> 3          G      2489    0.2489000
#> 4          T      2508    0.2508000
#> 5         AA       584    0.0589899
#> ...      ...      ...      ...
#> 16         GT       626    0.0632323
#> 17         TA       608    0.0614141
#> 18         TC       612    0.0618182
#> 19         TG       649    0.0655556
#> 20         TT       613    0.0619192

## 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      1661      0.1661
#> 2             Q      1697      0.1697
#> 3             R      1641      0.1641
#> 4             T      1669      0.1669
#> 5             W      1694      0.1694
#> ...           ...         ...         ...
#> 38            YQ      267      0.0269697
#> 39            YR      250      0.0252525
#> 40            YT      272      0.0274747
#> 41            YW      284      0.0286869
#> 42            YY      266      0.0268687

```

2.3 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")
  suppressWarnings(as.data.frame(cbind(Group = groupName, bkg)))
}

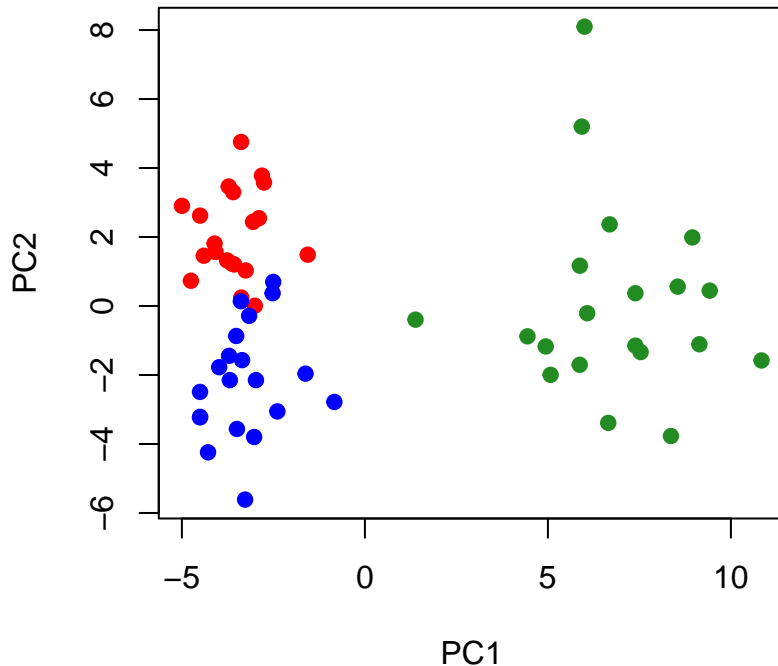
## Calculate k-let content (up to you what size k you want!):
s1 <- get_klet_matrix(s1, 4, 1)
s2 <- get_klet_matrix(s2, 4, 2)
s3 <- get_klet_matrix(s3, 4, 3)

# 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).

3 Shuffling

3.1 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 is 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  17273  17384
#> C     8081  8081   7975   8081
#> G     7583  7583   7677   7583
#> T    16952 16952  17075  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   6041   6483
#> AC     2614  2614   2683   2755
#> AG     2592  2592   2581   2555
#> AT     5276  5276   5953   5579
#> CA     3014  3014   2685   2901
#> CC     1376  1376   1381   1337
#> CG     1051  1051   1251   1143
#> CT     2621  2621   2651   2689
#> GA     2734  2734   2700   2688
#> GC     1104  1104   1194   1135
#> GG     1176  1176   1181   1219
```

```
#> GT      2561 2561 2596 2533
#> TA      4725 4725 5830 5293
#> TC      2977 2977 2709 2851
#> TG      2759 2759 2655 2663
#> TT      6477 6477 5859 6126
```

3.2 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)
))

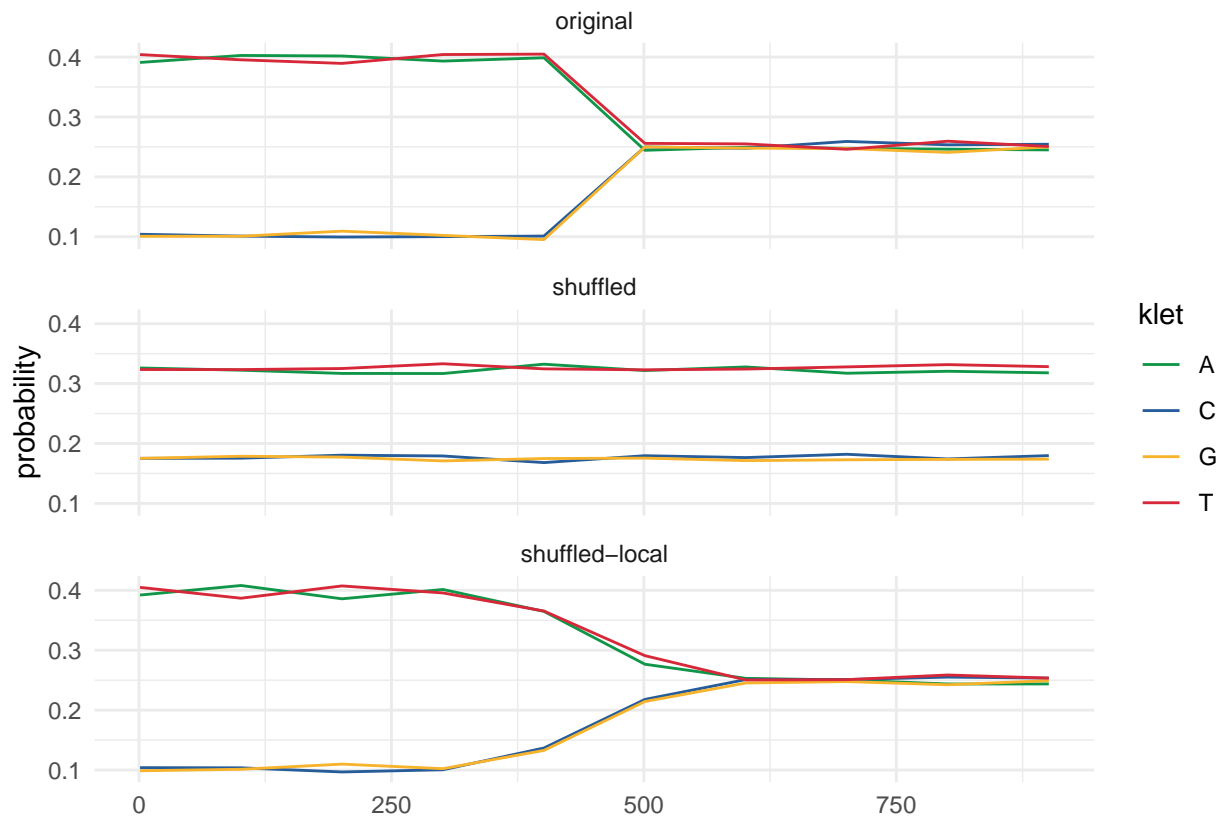
myseq_shuf <- shuffle_sequences(myseq)
myseq_shuf_local <- shuffle_sequences(myseq, window = TRUE)

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 <- suppressWarnings(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)
```



4 Sequence scanning and enrichment

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).

4.1 Choosing a logodds threshold

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.

Logodds thresholds

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.

P-values

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 usually 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] 4.858
```

Multiple testing-corrected P-values

This P-value can be further refined to correct for multiple testing (and becomes a Q-value). There are three available corrections that can be set in `scan_sequences()`: Bonferroni ("bonferroni"), Benjamini & Hochberg

(“BH”), and the false discovery rate (“fdr”) based on the empirical null distribution of motif hits in a set of sequences. They are excellently explained in Noble (2009), and these explanations will be briefly regurgitated here.

To begin to understand how these different corrections are implemented, consider the following motif, sequences, example P-value for an example motif hit, and the theoretical maximum number of motif hits:

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

(Example.Score <- score_match(ArabidopsisMotif, "TTCTCTTTTCTTTT"))
#> [1] 16.81
(Example.Pvalue <- motif_pvalue(ArabidopsisMotif, Example.Score))
#> [1] 6.612819e-07

(Max.Possible.Hits <- sum(width(ArabidopsisPromoters) - ncol(ArabidopsisMotif) + 1))
#> [1] 49300
```

The first correction method, Bonferroni, is by far the simplest. To calculate it, take the P-value of a motif hit and multiply it by the theoretical maximum number of hits:

```
(Example.bonferroni <- Example.Pvalue * Max.Possible.Hits)
#> [1] 0.0326012
```

As you can imagine, the level of punishment the P-value receives corresponds to the size of the sequences you are scanning. If you are scanning an entire genome, then you can expect this to be very punishing and only return near-perfect matches (or no matches). However for smaller sets of sequences this correction can be more appropriate.

Next, Benjamini & Hochberg. To perform this correction, the P-value is divided by the percentile rank of the P-value in the list of P-values for all theoretically possible hits sorted in ascending order (it also assumes that P-values are normally distributed under the null hypothesis). It is important to note that this means the correction cannot be calculated before the sequences have been scanned for the motif, and P-values have been calculated for all returned hits. When requesting this type of Q-value for the minimum threshold of score, `scan_sequences()` instead calculates the threshold from the input Q-value as a P-value, then filters the final results after Q-values have been calculated. Returning to our example:

```
(Scan.Results <- scan_sequences(ArabidopsisMotif, ArabidopsisPromoters,
  threshold = 0.8, threshold.type = "logodds", calc.qvals = FALSE))
#> DataFrame with 20 rows and 14 columns
#>      motif motif.i sequence sequence.i start stop
#>      <character> <integer> <character> <integer> <integer> <integer>
#> 1 YTTYTTTTTYYY 1 AT1G05670 47 68 82
#> 2 YTTYTTTTTYYY 1 AT1G19510 45 402 416
#> 3 YTTYTTTTTYYY 1 AT1G49840 27 899 913
#> 4 YTTYTTTTTYYY 1 AT2G22500 14 946 960
#> 5 YTTYTTTTTYYY 1 AT2G22500 14 948 962
#> ...
#> 16 YTTYTTTTTYYY 1 AT3G23170 34 603 617
#> 17 YTTYTTTTTYYY 1 AT4G19520 3 792 806
#> 18 YTTYTTTTTYYY 1 AT4G19520 3 793 807
#> 19 YTTYTTTTTYYY 1 AT4G27652 20 879 893
#> 20 YTTYTTTTTYYY 1 AT4G27652 20 881 895
#>      score match thresh.score min.score max.score score.pct
#>      <numeric> <character> <numeric> <numeric> <numeric> <numeric>
#> 1 15.407 GTTCTTTTTCTTT 15.0272 -125.07 18.784 82.0219
```

```

#> 2      17.405 TTTTCTTTTCTTTT      15.0272 -125.07  18.784  92.6586
#> 3      15.177 CTTTTTGTTTTTTC      15.0272 -125.07  18.784  80.7975
#> 4      15.827 TCCTCTCTTTCTCTC      15.0272 -125.07  18.784  84.2579
#> 5      15.908 CTCTCTTTCTCTCTT      15.0272 -125.07  18.784  84.6891
#> ...      ...      ...      ...      ...
#> 16     15.734 GTTCTTCTTTTTTTT      15.0272 -125.07  18.784  83.7628
#> 17     15.352 TTTTTTTTTTTTTTTT      15.0272 -125.07  18.784  81.7291
#> 18     15.352 TTTTTTTTTTTTTTTT      15.0272 -125.07  18.784  81.7291
#> 19     16.410 TTTTCTCTTTTTTTT      15.0272 -125.07  18.784  87.3616
#> 20     16.810 TTCTCTTTTTTTTTTT      15.0272 -125.07  18.784  89.4911
#>      strand      pvalue
#> <character> <numeric>
#> 1          + 3.95595e-06
#> 2          + 2.44369e-07
#> 3          + 5.01977e-06
#> 4          + 2.53853e-06
#> 5          + 2.39165e-06
#> ...      ...      ...
#> 16         + 2.83419e-06
#> 17         + 4.33848e-06
#> 18         + 4.33848e-06
#> 19         + 1.23950e-06
#> 20         + 6.61282e-07

```

First we sort and calculate the percentile ranks of our P-values, and then divide the P-values:

```

Pvalues <- Scan.Results$pvalue
Pvalues.Ranks <- (rank(Pvalues) / Max.Possible.Hits) * 100
Qvalues.BH <- Pvalues / Pvalues.Ranks
(Example.BH <- Qvalues.BH[Scan.Results$match == "TTCTCTTTT" ] [1])
#> [1] 6.52024e-05

```

Finally, calculating the false discovery rate from the empirical distribution of scores. This method requires some additional steps, as we must obtain the observed and null distributions of hits in our sequences. Then for each hit, divide the number of hits with a score equal to or greater in the null distribution with the number of hits with a score equal to or greater in the observed distribution. Along the way we must be wary of the nonmonotonicity of the final Q-values (meaning that as scores get smaller the Q-value does not always increase), and thus always select the minimum available Q-value as the score increases. To get the null distribution of hits, we can simply use the P-values associated with each score as these are analytically calculated from the null based on the background probabilities (see `?motif_pvalue`).

```

Scan.Results <- Scan.Results[order(Scan.Results$score, decreasing = TRUE), ]
Observed.Hits <- 1:nrow(Scan.Results)
Null.Hits <- Max.Possible.Hits * Scan.Results$pvalue
Qvalues.fdr <- Null.Hits / Observed.Hits
Qvalues.fdr <- rev(cummin(rev(Qvalues.fdr)))
(Example.fdr <- Qvalues.fdr[Scan.Results$match == "TTCTCTTTT" ] [1])
#> [1] 0.00652024

```

Similarly to Benjamini & Hochberg, these can only be known after scanning has occurred.

To summarize, we can compare the initial P-value with the different corrections:

```

knitr::kable(
  data.frame(
    What = c("Score", "P-value", "bonferroni", "BH", "fdr"),

```

```

Value = format(
  c(Example.Score, Example.Pvalue, Example.bonferroni, Example.BH, Example.fdr),
  scientific = FALSE
),
format = "markdown", caption = "Comparing P-value correction methods"
)

```

Table 1: Comparing P-value correction methods

What	Value
Score	16.8100000000000
P-value	0.0000006612819
bonferroni	0.0326011986749
BH	0.0000652023973
fdr	0.0065202397350

Use your best judgement as to which method is most appropriate for your specific use case.

4.2 Regular and higher order scanning

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("CWWWCC", nsites = 6)
sequences.k2 <- DNASTringSet(rep(c("AAAACC", "CTTTCC"), 3))
motif.k2 <- add_multifreq(motif.k2, sequences.k2)

```

Regular scanning:

```

scan_sequences(motif.k2, ArabidopsisPromoters, RC = TRUE,
  threshold = 0.9, threshold.type = "logodds")
#> DataFrame with 94 rows and 15 columns
#>      motif motif.i sequence sequence.i start stop score
#>   <character> <integer> <character> <integer> <integer> <integer> <numeric>
#> 1      motif         1 AT1G03850         4    203   209    9.08
#> 2      motif         1 AT1G03850         4    334   328    9.08
#> 3      motif         1 AT1G03850         4    713   707    9.08
#> 4      motif         1 AT1G05670        47    706   700    9.08
#> 5      motif         1 AT1G06160        48    498   492    9.08
#> ...      ...      ...      ...      ...      ...      ...
#> 90      motif         1 AT5G22690        46     81    87    9.08
#> 91      motif         1 AT5G22690        46    362   368    9.08
#> 92      motif         1 AT5G24660        49    146   140    9.08
#> 93      motif         1 AT5G58430        16    332   338    9.08
#> 94      motif         1 AT5G58430        16    343   349    9.08
#>      match thresh.score min.score max.score score.pct strand

```

```

#>      <character>      <numeric> <numeric> <numeric> <numeric> <character>
#> 1      CTAATCC          8.172   -19.649      9.08      100      +
#> 2      CTTTTC          8.172   -19.649      9.08      100      -
#> 3      CTTAACC          8.172   -19.649      9.08      100      -
#> 4      CTTTACC          8.172   -19.649      9.08      100      -
#> 5      CTAAACC          8.172   -19.649      9.08      100      -
#> ...      ...      ...      ...      ...      ...
#> 90     CAATACC          8.172   -19.649      9.08      100      +
#> 91     CAAATCC          8.172   -19.649      9.08      100      +
#> 92     CATTACC          8.172   -19.649      9.08      100      -
#> 93     CATAACC          8.172   -19.649      9.08      100      +
#> 94     CAAATCC          8.172   -19.649      9.08      100      +
#>      pvalue      qvalue
#>      <numeric> <numeric>
#> 1  0.000976562      1
#> 2  0.000976562      1
#> 3  0.000976562      1
#> 4  0.000976562      1
#> 5  0.000976562      1
#> ...      ...      ...
#> 90 0.000976562      1
#> 91 0.000976562      1
#> 92 0.000976562      1
#> 93 0.000976562      1
#> 94 0.000976562      1

```

Using 2-letter information to scan:

```

scan_sequences(motif.k2, ArabidopsisPromoters, use.freq = 2, RC = TRUE,
               threshold = 0.9, threshold.type = "logodds")
#> DataFrame with 8 rows and 15 columns
#>      motif motif.i sequence sequence.i start stop score
#>      <character> <integer> <character> <integer> <integer> <integer> <numeric>
#> 1      motif      1 AT1G19510      45      960      965 17.827
#> 2      motif      1 AT1G49840      27      959      964 17.827
#> 3      motif      1 AT1G77210      32      184      189 17.827
#> 4      motif      1 AT1G77210      32      954      959 17.827
#> 5      motif      1 AT2G37950      15      751      756 17.827
#> 6      motif      1 AT3G57640      33      917      922 17.827
#> 7      motif      1 AT4G12690      12      938      943 17.827
#> 8      motif      1 AT4G14365      35      977      982 17.827
#>      match thresh.score min.score max.score score.pct strand
#>      <character>      <numeric> <numeric> <numeric> <numeric> <character>
#> 1      CTTTTC      16.0443   -16.842   17.827      100      +
#> 2      CTTTTC      16.0443   -16.842   17.827      100      +
#> 3      CAAAAC      16.0443   -16.842   17.827      100      +
#> 4      CAAAAC      16.0443   -16.842   17.827      100      +
#> 5      CAAAAC      16.0443   -16.842   17.827      100      +
#> 6      CTTTTC      16.0443   -16.842   17.827      100      +
#> 7      CAAAAC      16.0443   -16.842   17.827      100      +
#> 8      CTTTTC      16.0443   -16.842   17.827      100      +
#>      pvalue      qvalue
#>      <numeric> <numeric>
#> 1 1.90735e-06 0.0236988

```

```
#> 2 1.90735e-06 0.0236988
#> 3 1.90735e-06 0.0236988
#> 4 1.90735e-06 0.0236988
#> 5 1.90735e-06 0.0236988
#> 6 1.90735e-06 0.0236988
#> 7 1.90735e-06 0.0236988
#> 8 1.90735e-06 0.0236988
```

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")
#> DataFrame with 491 rows and 15 columns
#>      motif motif.i sequence sequence.i start stop score
#>      <character> <integer> <character> <integer> <integer> <integer> <numeric>
#> 1 motif 1 AT1G03850 4 56 51 11.934
#> 2 motif 1 AT1G03850 4 57 52 11.934
#> 3 motif 1 AT1G03850 4 58 53 11.934
#> 4 motif 1 AT1G03850 4 59 54 11.934
#> 5 motif 1 AT1G03850 4 243 248 11.934
#> ...
#> 487 motif 1 AT5G64310 22 589 594 11.934
#> 488 motif 1 AT5G64310 22 590 595 11.934
#> 489 motif 1 AT5G64310 22 591 596 11.934
#> 490 motif 1 AT5G64310 22 592 597 11.934
#> 491 motif 1 AT5G64310 22 696 701 11.934
#>      match thresh.score min.score max.score score.pct strand
#>      <character> <numeric> <numeric> <numeric> <numeric> <character>
#> 1 AAAAAA 10.7406 -39.948 11.934 100 -
#> 2 AAAAAA 10.7406 -39.948 11.934 100 -
#> 3 AAAAAA 10.7406 -39.948 11.934 100 -
#> 4 AAAAAA 10.7406 -39.948 11.934 100 -
#> 5 AAAAAA 10.7406 -39.948 11.934 100 +
#> ...
#> 487 AAAAAA 10.7406 -39.948 11.934 100 +
#> 488 AAAAAA 10.7406 -39.948 11.934 100 +
#> 489 AAAAAA 10.7406 -39.948 11.934 100 +
#> 490 AAAAAA 10.7406 -39.948 11.934 100 +
#> 491 AAAAAA 10.7406 -39.948 11.934 100 +
#>      pvalue qvalue
#>      <numeric> <numeric>
#> 1 0.000244141 0.0494745
#> 2 0.000244141 0.0494745
#> 3 0.000244141 0.0494745
#> 4 0.000244141 0.0494745
#> 5 0.000244141 0.0494745
#> ...
#> 487 0.000244141 0.0494745
#> 488 0.000244141 0.0494745
#> 489 0.000244141 0.0494745
```

```

#> 490 0.000244141 0.0494745
#> 491 0.000244141 0.0494745

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

scan_sequences(motif, ArabidopsisPromoters, RC = TRUE, threshold = 0.9,
               threshold.type = "logodds", no.overlaps = TRUE)
#> DataFrame with 220 rows and 15 columns
#>      motif motif.i sequence sequence.i start stop score
#>      <character> <integer> <character> <integer> <integer> <integer> <numeric>
#> 1 motif 1 AT1G03850 4 56 51 11.934
#> 2 motif 1 AT1G03850 4 243 248 11.934
#> 3 motif 1 AT1G03850 4 735 740 11.934
#> 4 motif 1 AT1G05670 47 32 27 11.934
#> 5 motif 1 AT1G05670 47 78 73 11.934
#> ...
#> 216 motif 1 AT5G64310 22 251 246 11.934
#> 217 motif 1 AT5G64310 22 342 347 11.934
#> 218 motif 1 AT5G64310 22 586 591 11.934
#> 219 motif 1 AT5G64310 22 592 597 11.934
#> 220 motif 1 AT5G64310 22 696 701 11.934
#>      match thresh.score min.score max.score score.pct strand
#>      <character> <numeric> <numeric> <numeric> <numeric> <character>
#> 1 AAAAAA 10.7406 -39.948 11.934 100 -
#> 2 AAAAAA 10.7406 -39.948 11.934 100 +
#> 3 AAAAAA 10.7406 -39.948 11.934 100 +
#> 4 AAAAAA 10.7406 -39.948 11.934 100 -
#> 5 AAAAAA 10.7406 -39.948 11.934 100 -
#> ...
#> 216 AAAAAA 10.7406 -39.948 11.934 100 -
#> 217 AAAAAA 10.7406 -39.948 11.934 100 +
#> 218 AAAAAA 10.7406 -39.948 11.934 100 +
#> 219 AAAAAA 10.7406 -39.948 11.934 100 +
#> 220 AAAAAA 10.7406 -39.948 11.934 100 +
#>      pvalue qvalue
#>      <numeric> <numeric>
#> 1 0.000244141 0.0494745
#> 2 0.000244141 0.0494745
#> 3 0.000244141 0.0494745
#> 4 0.000244141 0.0494745
#> 5 0.000244141 0.0494745
#> ...
#> 216 0.000244141 0.0494745
#> 217 0.000244141 0.0494745
#> 218 0.000244141 0.0494745
#> 219 0.000244141 0.0494745
#> 220 0.000244141 0.0494745

```

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)
#> GRanges object with 94 ranges and 11 metadata columns:

```

```

#>      seqnames      ranges strand |      motif motif.i sequence.i      score
#>      <Rle> <IRanges> <Rle> | <character> <integer> <integer> <numeric>
#> [1] AT1G03850    203-209   + |      motif      1         4      9.08
#> [2] AT1G03850    328-334   - |      motif      1         4      9.08
#> [3] AT1G03850    707-713   - |      motif      1         4      9.08
#> [4] AT1G05670    700-706   - |      motif      1        47      9.08
#> [5] AT1G06160    956-962   + |      motif      1        48      9.08
#> ...      ...      ...      ...      ...      ...      ...
#> [90] AT5G22690    362-368   + |      motif      1        46      9.08
#> [91] AT5G22690      52-58   - |      motif      1        46      9.08
#> [92] AT5G24660    140-146   - |      motif      1        49      9.08
#> [93] AT5G58430    332-338   + |      motif      1        16      9.08
#> [94] AT5G58430    343-349   + |      motif      1        16      9.08
#>      match thresh.score min.score max.score score.pct      pvalue
#>      <character>      <numeric> <numeric> <numeric> <numeric>      <numeric>
#> [1]      CTAATCC      8.172    -19.649      9.08      100 0.000976562
#> [2]      CTTTTC      8.172    -19.649      9.08      100 0.000976562
#> [3]      CTTAACC      8.172    -19.649      9.08      100 0.000976562
#> [4]      CTTTACC      8.172    -19.649      9.08      100 0.000976562
#> [5]      CTAATCC      8.172    -19.649      9.08      100 0.000976562
#> ...      ...      ...      ...      ...      ...
#> [90]      CAAATCC      8.172    -19.649      9.08      100 0.000976562
#> [91]      CATTACC      8.172    -19.649      9.08      100 0.000976562
#> [92]      CATTACC      8.172    -19.649      9.08      100 0.000976562
#> [93]      CATAACC      8.172    -19.649      9.08      100 0.000976562
#> [94]      CAAATCC      8.172    -19.649      9.08      100 0.000976562
#>      qvalue
#>      <numeric>
#> [1]      1
#> [2]      1
#> [3]      1
#> [4]      1
#> [5]      1
#> ...      ...
#> [90]      1
#> [91]      1
#> [92]      1
#> [93]      1
#> [94]      1
#> -----
#> seqinfo: 50 sequences from an unspecified genome

```

4.3 Visualizing motif hits across sequences

A few suggestions for different ways of plotting hits across sequences are presented here.

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,
  threshold = 0.2, threshold.type = "logodds")

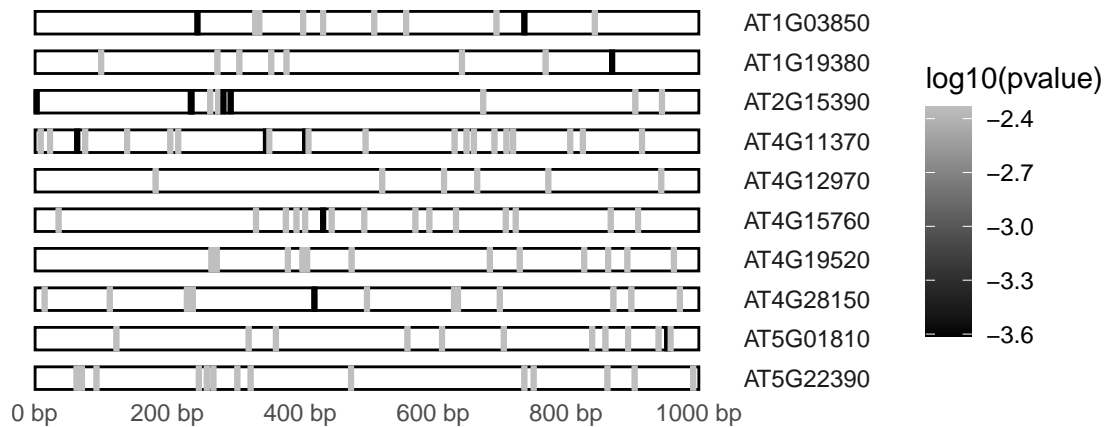
## Just plot the motif hits:
autoplot(res, layout = "karyogram", aes(fill = motif, color = motif)) +
  theme(
    strip.background = element_rect(fill = NA, colour = NA),
    panel.background = element_rect(fill = NA, colour = NA)
  )

#> 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.
```



Alternatively, just a simple heatmap with only `ggplot2`.

```
library(universalmotif)
library(ggplot2)

data(ArabidopsisMotif)
data(ArabidopsisPromoters)

res <- scan_sequences(ArabidopsisMotif, ArabidopsisPromoters,
  threshold = 0, threshold.type = "logodds.abs")
res <- suppressWarnings(as.data.frame(res))
res$x <- mapply(function(x, y) mean(c(x, y)), res$start, res$stop)

ggplot(res, aes(x, sequence, fill = score)) +
  scale_fill_viridis_c() +
  scale_x_continuous(expand = c(0, 0)) +
  xlim(0, 1000) +
  xlab(element_blank()) +
  ylab(element_blank()) +
  geom_tile(width = ncol(ArabidopsisMotif)) +
  theme_bw() +
  theme(panel.grid = element_blank(), axis.text.y = element_text(size = 6))
```



Using packages such as `ggExtra` or `ggpubr`, one could even plot marginal histogram or density plots above or below to illustrate any motif positional preference within the sequences. (Though keep in mind that the hit coordinates and sequence lengths would need to be normalized if not all sequences were of the same length, as they are here.)

Finally, the distribution of all possible motif scores could be shown as a line plot across the sequences.

```
library(universalmotif)
library(ggplot2)

data(ArabidopsisMotif)
data(ArabidopsisPromoters)

res <- scan_sequences(ArabidopsisMotif, ArabidopsisPromoters[1:5],
  threshold = -Inf, threshold.type = "logodds.abs")
res <- suppressWarnings(as.data.frame(res))
res$position <- mapply(function(x, y) mean(c(x, y)), res$start, res$stop)

ggplot(res, aes(position, score, colour = score)) +
  geom_line() +
  geom_hline(yintercept = 0, colour = "red", alpha = 0.3) +
  theme_bw() +
  scale_colour_viridis_c() +
  facet_wrap(~sequence, ncol = 1) +
  xlab(element_blank()) +
  ylab(element_blank()) +
  theme(strip.background = element_blank())
```



4.4 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 15 columns
#>      motif motif.i motif.consensus target.hits target.seq.hits
#>      <character> <integer>      <character>      <integer>      <integer>
#> 1 YTTYTTTTTTTTY 1 YTYTYTTYTTYTTY 244 50
#> target.seq.count bkg.hits bkg.seq.hits bkg.seq.count Pval Qual
#>      <integer> <integer>      <integer>      <integer> <numeric> <numeric>
#> 1 50 143 47 50 1.67436e-07 1.67436e-07
#>      Eval pct.target.seq.hits pct.bkg.seq.hits target.enrichment
#>      <numeric>      <numeric>      <numeric>      <numeric>
```

```
#> 1 3.34871e-07          100          94          1.06383
```

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.)

4.5 Fixed and variable-length 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
#>      Target sites: 1
#>      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")
#> DataFrame with 75 rows and 15 columns
#>      motif motif.i sequence sequence.i start stop score
#>      <character> <integer> <character> <integer> <integer> <integer> <numeric>
#> 1 PartA/PartB      1 AT1G03850      4 376 394 11.178
#> 2 PartA/PartB      1 AT1G03850      4 414 432 12.168
#> 3 PartA/PartB      1 AT1G06160     48 144 161 11.918
#> 4 PartA/PartB      1 AT1G12990     28 71 90 11.428
#> 5 PartA/PartB      1 AT1G19380      2 226 245 11.428
```

```

#> ...      ...      ...      ...      ...      ...      ...
#> 71 PartA/PartB      1 AT5G22690      46      638      656      11.178
#> 72 PartA/PartB      1 AT5G47230      24      91      110      12.418
#> 73 PartA/PartB      1 AT5G47230      24      449      468      11.428
#> 74 PartA/PartB      1 AT5G64310      22      869      888      11.428
#> 75 PartA/PartB      1 AT5G64310      22      909      927      11.178
#>          match thresh.score min.score max.score score.pct      strand
#>          <character>      <numeric> <numeric> <numeric> <numeric> <character>
#> 1      TATATGT.....GGTGCAA      11.1384 -93.212      27.846      40.1422      +
#> 2      TTGATAT.....TGTTAGA      11.1384 -93.212      27.846      43.6975      +
#> 3      TTTATGT.....GGTTTGT      11.1384 -93.212      27.846      42.7997      +
#> 4      GTTATGT.....TGTTAGA      11.1384 -93.212      27.846      41.0400      +
#> 5      TTTACAG.....CGTTCGT      11.1384 -93.212      27.846      41.0400      +
#> ...      ...      ...      ...      ...      ...
#> 71      TTCATTT.....GGCTTGA      11.1384 -93.212      27.846      40.1422      +
#> 72      TTTATAC.....TGTTCCA      11.1384 -93.212      27.846      44.5953      +
#> 73      TATATGT.....GGGTCAA      11.1384 -93.212      27.846      41.0400      +
#> 74      ATAATAT.....CGTTAGA      11.1384 -93.212      27.846      41.0400      +
#> 75      TTCATAT.....GTCACGA      11.1384 -93.212      27.846      40.1422      +
#>          pvalue      qvalue
#>          <numeric>      <numeric>
#> 1      1.60187e-07 0.000105403
#> 2      1.60187e-07 0.000105403
#> 3      1.60187e-07 0.000105403
#> 4      1.60187e-07 0.000105403
#> 5      1.60187e-07 0.000105403
#> ...      ...      ...
#> 71      1.60187e-07 0.000105403
#> 72      1.60187e-07 0.000105403
#> 73      1.60187e-07 0.000105403
#> 74      1.60187e-07 0.000105403
#> 75      1.60187e-07 0.000105403

```

4.6 Detecting low complexity regions and sequence masking

Highly-repetitive low complexity regions can oftentimes cause problems during *de novo* motif discovery, leading to obviously false motifs being returned. One way to get around this issue is to preemptively remove or mask these regions. The `universalmotif` package includes a few functions which can help carry out this task.

Using `mask_seqs()`, one can mask a specific pattern of letters in `XStringSet` objects. Consider the following sequences:

```

library(universalmotif)
library(Biostrings)

Ex.seq <- DNASTringSet(c(
  A = "GTTGAAAAAAAAAAAAAAAAACAGACGT",
  B = "TTAGATGGCCCATAGCTTATACGGCAA",
  C = "AATAAAATGCTTAGGAAATCGATTGCC"
))

```

We can easily mask portions that contain, say, stretches of at least 8 As:

```
mask_seqs(Ex.seq, "AAAAAAA")
#> DNASTringSet object of length 3:
#>      width seq                      names
#> [1]    27 GTTG-----CAGACGT      A
#> [2]    27 TTAGATGGCCCATAGCTTATACGGCAA  B
#> [3]    27 AATAAAATGCTTAGGAAATCGATTGCC  C
```

Alternatively, instead of masking a known stretch of letters one can find low complexity regions using `sequence_complexity()`, and then mask specific regions in the sequences using `mask_ranges()`. The `sequence_complexity()` function has several complexity metrics available: the Wootton-Federhen (Wootton and Federhen 1993) and Trifonov (Trifonov 1990) algorithms (and their approximations) are well described in Orlov and Potapov (2004), and DUST in Morgulis et al. (2006). See `?sequence_complexity` for more details.

```
(Ex.DUST <- sequence_complexity(Ex.seq, window.size = 10, method = "DUST",
  return.granges = TRUE))
#> GRanges object with 15 ranges and 1 metadata column:
#>      seqnames      ranges strand | complexity
#>      <Rle> <IRanges> <Rle> | <numeric>
#> [1]      A      1-10      * | 0.857143
#> [2]      A      6-15      * | 4.000000
#> [3]      A     11-20      * | 4.000000
#> [4]      A     16-25      * | 0.428571
#> [5]      A     21-27      * | 0.000000
#> ...      ...      ...      ...
#> [11]     C      1-10      * | 0.285714
#> [12]     C      6-15      * | 0.000000
#> [13]     C     11-20      * | 0.000000
#> [14]     C     16-25      * | 0.000000
#> [15]     C     21-27      * | 0.000000
#> -----
#> seqinfo: 3 sequences from an unspecified genome
```

Using the DUST algorithm, we can see there are a couple of regions which spike in the complexity score (for this particular algorithm, more complex sequences converge towards zero). Now it is only a matter of filtering for those regions and using `mask_ranges()`.

```
(Ex.DUST <- Ex.DUST[Ex.DUST$complexity >= 3])
#> GRanges object with 2 ranges and 1 metadata column:
#>      seqnames      ranges strand | complexity
#>      <Rle> <IRanges> <Rle> | <numeric>
#> [1]      A      6-15      * |      4
#> [2]      A     11-20      * |      4
#> -----
#> seqinfo: 3 sequences from an unspecified genome
mask_ranges(Ex.seq, Ex.DUST)
#> DNASTringSet object of length 3:
#>      width seq                      names
#> [1]    27 GTTGA-----CAGACGT      A
#> [2]    27 TTAGATGGCCCATAGCTTATACGGCAA  B
#> [3]    27 AATAAAATGCTTAGGAAATCGATTGCC  C
```

Now these sequences could be used directly with `scan_sequences()` or written to a fasta file using `Biostrings::writeXStringSet()` for use with an external *de novo* motif discovery program such as MEME.

5 Motif discovery with MEME

Note: In the time since the inception of the `run_meme()` function, Spencer Nystrom (a contributor to `universalmotif`) has created the `memes` package as a interface to much of the `MEME` suite. It is fully interoperable with the `universalmotif` package and provides a much more convenient way to run `MEME` programs from within `R`. Install it from Bioconductor with `BiocManager::install("memes")`.

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.plantmart28)
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.

6 Miscellaneous string utilities

Since biological sequences are usually contained in `XStringSet` class objects, `sequence_complexity()`, `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:

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

```
library(universalmotif)

string <- "DASDSDDASDSSA"

calc_complexity(string)
#> [1] 0.7823323

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] "DSDSDSDDASASSA"
```

A few other utilities have also been made available (based on the internal code of other `universalmotif` functions) that work on simple character vectors:

- `calc_windows()`: calculate the coordinates for sliding windows from 1 to any number `n`
- `get_klets()`: get a list of all possible `k`-lets for any sequence alphabet
- `slide_fun()`: apply a function over sliding windows across a single string
- `window_string()`: retrieve characters from sliding windows of a single string

```
library(universalmotif)

calc_windows(n = 12, window = 4, overlap = 2)
#>      start stop
#> 1         1   4
#> 2         3   6
#> 3         5   8
#> 4         7  10
#> 5         9  12

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

slide_fun("ABCDEFGH", charToRaw, raw(2), window = 2, overlap = 1)
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
#> [1,]  41  42  43  44  45  46  47
```

```
#> [2,] 42 43 44 45 46 47 48
```

```
window_string("ABCDEFGH", window = 2, overlap = 1)
```

```
#> [1] "AB" "BC" "CD" "DE" "EF" "FG" "GH"
```

Session info

```
#> R version 4.5.0 Patched (2025-04-21 r88169)
```

```
#> Platform: aarch64-apple-darwin20
```

```
#> Running under: macOS Ventura 13.7.1
```

```
#>
```

```
#> Matrix products: default
```

```
#> BLAS: /Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/lib/libRblas.0.dylib
```

```
#> LAPACK: /Library/Frameworks/R.framework/Versions/4.5-arm64/Resources/lib/libRlapack.dylib; LAPACK v
```

```
#>
```

```
#> locale:
```

```
#> [1] C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
```

```
#>
```

```
#> time zone: America/New_York
```

```
#> tzcode source: internal
```

```
#>
```

```
#> attached base packages:
```

```
#> [1] stats4      stats      graphics  grDevices  utils      datasets  methods
```

```
#> [8] base
```

```
#>
```

```
#> other attached packages:
```

```
#> [1] ggbio_1.57.0      TFBSTools_1.47.0      cowplot_1.1.3
```

```
#> [4] dplyr_1.1.4        ggtree_3.17.0         ggplot2_3.5.2
```

```
#> [7] MotifDb_1.51.0     GenomicRanges_1.61.0  Biostrings_2.77.0
```

```
#> [10] GenomeInfoDb_1.45.0 XVector_0.49.0         IRanges_2.43.0
```

```
#> [13] S4Vectors_0.47.0   BiocGenerics_0.55.0   generics_0.1.3
```

```
#> [16] universalmotif_1.27.2
```

```
#>
```

```
#> loaded via a namespace (and not attached):
```

```
#> [1] RColorBrewer_1.1-3      rstudioapi_0.17.1
```

```
#> [3] jsonlite_2.0.0          magrittr_2.0.3
```

```
#> [5] GenomicFeatures_1.61.0  farver_2.1.2
```

```
#> [7] rmarkdown_2.29          fs_1.6.6
```

```
#> [9] BiocIO_1.19.0           vctrs_0.6.5
```

```
#> [11] memoise_2.0.1           Cairo_1.6-2
```

```
#> [13] Rsamtools_2.25.0        RCurl_1.98-1.17
```

```
#> [15] base64enc_0.1-3         tinytex_0.57
```

```
#> [17] progress_1.2.3          htmltools_0.5.8.1
```

```
#> [19] S4Arrays_1.9.0          curl_6.2.2
```

```
#> [21] SparseArray_1.9.0       Formula_1.2-5
```

```
#> [23] gridGraphics_0.5-1      htmlwidgets_1.6.4
```

```
#> [25] httr2_1.1.2             plyr_1.8.9
```

```
#> [27] cachem_1.1.0            GenomicAlignments_1.45.0
```

```
#> [29] lifecycle_1.0.4         pkgconfig_2.0.3
```

```
#> [31] Matrix_1.7-3            R6_2.6.1
```

```
#> [33] fastmap_1.2.0           GenomeInfoDbData_1.2.14
```

```
#> [35] MatrixGenerics_1.21.0   digest_0.6.37
```

```
#> [37] applot_0.2.5            colorspace_2.1-1
```

```

#> [39] TFMPvalue_0.0.9
#> [41] OrganismDbi_1.51.0
#> [43] patchwork_1.3.0
#> [45] RSQLite_2.3.9
#> [47] seqLogo_1.75.0
#> [49] httr_1.4.7
#> [51] compiler_4.5.0
#> [53] withr_3.0.2
#> [55] htmlTable_2.4.3
#> [57] DBI_1.2.3
#> [59] biomaRt_2.65.0
#> [61] rappdirs_0.3.3
#> [63] rjson_0.2.23
#> [65] caTools_1.18.3
#> [67] splitstackshape_1.4.8
#> [69] ape_5.8-1
#> [71] nnet_7.3-20
#> [73] restfulr_0.0.15
#> [75] grid_4.5.0
#> [77] cluster_2.1.8.1
#> [79] ade4_1.7-23
#> [81] BSgenome_1.77.0
#> [83] tidyr_1.3.1
#> [85] data.table_1.17.0
#> [87] pillar_1.10.2
#> [89] yulab.utils_0.2.0
#> [91] treeio_1.33.0
#> [93] rtracklayer_1.69.0
#> [95] biovizBase_1.57.0
#> [97] tidyselect_1.2.1
#> [99] knitr_1.50
#> [101] grImport2_0.3-3
#> [103] ProtGenerics_1.41.0
#> [105] xfun_0.52
#> [107] matrixStats_1.5.0
#> [109] UCSC.utils_1.5.0
#> [111] ggfun_0.1.8
#> [113] evaluate_1.0.3
#> [115] tibble_3.2.1
#> [117] BiocManager_1.30.25
#> [119] cli_3.6.5
#> [121] dichromat_2.0-0.1
#> [123] dbplyr_2.5.0
#> [125] XML_3.99-0.18
#> [127] blob_1.2.4
#> [129] jpeg_0.1-11
#> [131] bitops_1.0-9
#> [133] pwalign_1.5.0
#> [135] tidytree_0.4.6
#> [137] scales_1.4.0
#> [139] purrr_1.0.4
#> [141] rlang_1.1.6
GGally_2.2.1
AnnotationDbi_1.71.0
Hmisc_5.2-3
filelock_1.0.3
labeling_0.4.3
abind_1.4-8
bit64_4.6.0-1
backports_1.5.0
BiocParallel_1.43.0
ggstats_0.9.0
MASS_7.3-65
DelayedArray_0.35.1
gtools_3.9.5
tools_4.5.0
foreign_0.8-90
ggseqlogo_0.2
glue_1.8.0
nlme_3.1-168
checkmate_2.3.2
reshape2_1.4.4
gtable_0.3.6
ensembldb_2.33.0
hms_1.1.3
xml2_1.3.8
stringr_1.5.1
BiocFileCache_2.99.0
lattice_0.22-7
bit_4.6.0
RBGL_1.85.0
DirichletMultinomial_1.51.0
gridExtra_2.3
bookdown_0.43
SummarizedExperiment_1.39.0
Biobase_2.69.0
stringi_1.8.7
lazyeval_0.2.2
yaml_2.3.10
codetools_0.2-20
graph_1.87.0
ggplotify_0.1.2
rpart_4.1.24
Rcpp_1.0.14
png_0.1-8
parallel_4.5.0
prettyunits_1.2.0
AnnotationFilter_1.33.0
txdbmaker_1.5.0
viridisLite_0.4.2
VariantAnnotation_1.55.0
motifStack_1.53.0
crayon_1.5.3
KEGGREST_1.49.0

```

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: 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: 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.
- Morgulis, A., E. M. Gertz, A. A. Schaffer, and R. Agarwala. 2006. "A Fast and Symmetric DUST Implementation to Mask Low-Complexity DNA Sequences." *Journal of Computational Biology* 13: 1028–40.
- Noble, William S. 2009. "How Does Multiple Testing Correction Work?" *Nature Biotechnology* 27: 1135–37.
- Orlov, Y. L., and V. N. Potapov. 2004. "Complexity: An Internet Resource for Analysis of DNA Sequence Complexity." *Nucleic Acids Research* 32: W628–33.
- 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.
- Trifonov, E. N. 1990. "Making Sense of the Human Genome." In *Structure & Methods*, edited by R. H. Sarma, 69–77. Albany: Adenine Press.
- Wootton, J. C., and S. Federhen. 1993. "Statistics of Local Complexity in Amino Acid Sequences and Sequence Databases." *Computers & Chemistry* 17: 149–63.