Sequence manipulation and scanning

Benjamin Jean-Marie Tremblay[*1]

[1]University of Waterloo, Waterloo, Canada

*b2tremblay@uwaterloo.ca

9 May 2019

Contents

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 advanced usage and analyses, see the advanced usage 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()`, in it's simplest form generates a set of letters in random order with `sample()`, then passes these strings to the *Biostrings* package. 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 (chosen by `sample()`) 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
#>   A DNAStringSet instance of length 500
#>       width seq
#>   [1]   100 GACAAGTCAGGTAATCAACGTCGAAACTAAT...ATTCATAAAAGGTAATGTAATATAGCGAATA
#>   [2]   100 GAGAATTGGGGATTATTCTCTCTCCTGCATA...TTGTCGCGTCTACATTCTGATATGTTCAATG
#>   [3]   100 TTGAGAAATGATGCTGATTACCTATGTAGTG...AGATGAGGCGTATTCGTGTTCGAGGATATTA
#>   [4]   100 GGCTTTCGGGACAGGCGAACAACATCTGGCG...TCTACCCGCATTATGGGCAGAAAATAGATGG
#>   [5]   100 AGTTCTAAGCCAGGAGTTACCCGCTAAGAAA...TCTCGCGACAGCATAAGAGTTGGTTGCTAAA
#>   ...   ... ...
#> [496]   100 ATATATCTCTCAACCAGAGTAGAACCTTGTA...GAATAAGTGTATTTTTACTTACATGCAAATA
#> [497]   100 CCAACTACTTCATTCTGCTCTCTCGGAATGT...CACTTAAGTAAGAGGCAAGTTTTCATTGTTT
#> [498]   100 CGTGACGTTCTTATCTATATACAGCTGTCCA...GATAAGTTACTACTTAAAATATAGGGGACTG
#> [499]   100 ATCGAGTCTAAATTCCAGAATACTTTATATC...ACGCAGTAATGCGGTGCCTGGGCTACACCGT
#> [500]   100 CACCTAGCGCTCCACACAATTTATCCAAGGG...CTTACAGACGTTTTTTCTCGTGATGTTATTG

## Amino acid:

create_sequences(alphabet = "AA")
#>   A AAStringSet instance of length 100
```

```
#>       width seq
#>   [1]   100 MHHDVSVCIGPLYQGHFSGWYECAWEHKDDD...RSQDRDAIQKDWDVNKQYAIGQKICTLCNVH
#>   [2]   100 CEPYSNLNRWARERIVLRCHITVFDICDASG...CDTMNWQCLSRRALLCHGMDYSHTIVKEKAK
#>   [3]   100 WNCANQYNYRHERTQKSSFYKLATWDVWRKH...AHVPRWCQMKNDCQYHEQDCVYKIVQMRTYG
#>   [4]   100 RFNHEKDNSHLKEGCCEVEQFWWHVETVDDC...QKQNCYITNTKSMTDSWRTEGFGNAEHQNKF
#>   [5]   100 QQHNQSRCDVNADYVLRQPTEFHYWWDRYRE...YPYRDWGETMWAILATRQNDSAHLGIGLMWS
#>   ...   ... ...
#>  [96]   100 RANWDYTFTPTDNIRWAINDWTKTMHRWDPN...MDTAKLPEDNSVSMHFMKICEDNPNLCPNND
#>  [97]   100 IAFRLEAESFQDIQYTGTRLEQEFTGNINCD...DPYAAQNNPDIMGPHRDMFMCEKMWMCQLIS
#>  [98]   100 KDVTTGATKDFHENEQWLRTHMAGGQAFFIT...AKSCFYQDIHWGSIYDKLQSWFFRTDTAWEK
#>  [99]   100 PINKNVGGCFNVIWIGWKRQIHAQTRCTGCS...CVAGFVKLILAYTWWTDNEMRTVKCQAQVWI
#> [100]   100 LYMCFGMHECFQKMTCFEELMPHNRFYTIEI...FFLPEWFSNGCGPTTSFRAWMHRIYAHAKMA

## Any set of characters can be used

create_sequences(alphabet = paste(letters, collapse = ""))
#>   A BStringSet instance of length 100
#>       width seq
#>   [1]   100 mebjcdpykbvlngdumtvrexlqlgwpybh...oedpyczvovwyzaxfjzgvkbsxyfqluvx
#>   [2]   100 nerufsrcolelyzdwbbqwisebedojysp...kceetmrmlntsbxmalqjotljvgbezazs
#>   [3]   100 gxjkdnrtyvdavlahstuvbjmsejrmyvq...qxxxfrxijnsdzwuhfvmvthzpwwdlyot
#>   [4]   100 lcnbrlbdsstrioolmqgbogfcuclvvhj...mkyjlztatuxrbysjrjruncnwxwodxka
#>   [5]   100 vrbmimjanvypjynsnjmravjckwjxtvo...kyqnxyvqfravetlrczlvcjzmvopauye
#>   ...   ... ...
#>  [96]   100 ruqykhiqlszbqkyqhormodgnzgcrelk...yyngbzhmtburarnknlqgoxypvbrapvd
#>  [97]   100 newwreuvwmzsuymovnrpseidrskhqmz...akuopojigfsdleqwmgvcihizfakkaju
#>  [98]   100 jqtevftqjgegvzzjloztfspkfaffcow...twzvldolnqnvoyozemwzgcakvnidmyt
#>  [99]   100 ucdxvwxcjywblrfzycuiqepvfkmfnly...gsiupsawqcquwuybjaughfphxargeex
#> [100]   100 msfnjfwyqgoarwywhjvzairgobgtkum...brzdwwlicudycjtzcrxmediajnqdjhm
```

3 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 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, the most commonly used tool is likely uShuffle (Jiang et al. 2008). Despite this the *universalmotif* package aims to provide its own `k`-let shuffling capabilities for use within R via `shuffle_sequences()`.

The *universalmotif* 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. Exact letter counts will be preserved. However in order for this to be possible, the first and last letters will remain unshuffled.

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.

The third method `linear` preserves the original 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 sub-sequences (just re-assembled differently).

```
library(universalmotif)
data(ArabidopsisPromoters)

## Potentially starting off with some external sequences:
# library(Biostrings)
# 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 <- colSums(oligonucleotideFrequency(ArabidopsisPromoters,
                                             1, as.prob = FALSE))
e.letter <- colSums(oligonucleotideFrequency(euler, 1, as.prob = FALSE))
m.letter <- colSums(oligonucleotideFrequency(markov, 1, as.prob = FALSE))
l.letter <- colSums(oligonucleotideFrequency(linear, 1, as.prob = FALSE))

data.frame(original=o.letter, euler=e.letter, markov=m.letter, linear=l.letter)
#>   original euler markov linear
#> A    17384 17384  17360  17384
#> C     8081  8081   8116   8081
#> G     7583  7583   7593   7583
#> T    16952 16952  16931  16952

o.counts <- colSums(oligonucleotideFrequency(ArabidopsisPromoters,
                                             2, as.prob = FALSE))
e.counts <- colSums(oligonucleotideFrequency(euler, 2, as.prob = FALSE))
m.counts <- colSums(oligonucleotideFrequency(markov, 2, as.prob = FALSE))
l.counts <- colSums(oligonucleotideFrequency(linear, 2, as.prob = FALSE))

data.frame(original=o.counts, euler=e.counts, markov=m.counts, linear=l.counts)
#>    original euler markov linear
#> AA     6893  6893   6901   6429
#> AC     2614  2614   2631   2752
#> AG     2592  2592   2569   2615
#> AT     5276  5276   5240   5568
#> CA     3014  3014   3048   2934
#> CC     1376  1376   1364   1308
```

```
#> CG     1051  1051   1054   1091
#> CT     2621  2621   2646   2741
#> GA     2734  2734   2703   2661
#> GC     1104  1104   1163   1171
#> GG     1176  1176   1178   1192
#> GT     2561  2561   2539   2549
#> TA     4725  4725   4689   5343
#> TC     2977  2977   2952   2843
#> TG     2759  2759   2783   2679
#> TT     6477  6477   6490   6074
```

4 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 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, list.out = FALSE)
#>         A         C         G         T        AA        AC        AG
#> 0.2488000 0.2564000 0.2544000 0.2404000 0.0645454 0.0629292 0.0606060
#>        AT        CA        CC        CG        CT        GA        GC
#> 0.0604040 0.0645454 0.0649494 0.0662626 0.0609090 0.0609090 0.0649494
#>        GG        GT        TA        TC        TG        TT
#> 0.0672727 0.0615151 0.0588888 0.0638383 0.0600000 0.0574747

## Background of non DNA/RNA sequences:
qwerty <- create_sequences("QWERTY")
get_bkg(qwerty, k = 1:2, list.out = FALSE)
#>         E         Q         R         T         W         Y        EE
#> 0.1684000 0.1686000 0.1677000 0.1633000 0.1640000 0.1680000 0.0282828
#>        EQ        ER        ET        EW        EY        QE        QQ
#> 0.0282828 0.0291919 0.0286868 0.0256565 0.0286868 0.0288888 0.0287878
#>        QR        QT        QW        QY        RE        RQ        RR
#> 0.0276767 0.0274747 0.0285858 0.0275757 0.0284848 0.0273737 0.0284848
#>        RT        RW        RY        TE        TQ        TR        TT
#> 0.0279798 0.0277777 0.0278787 0.0262626 0.0286868 0.0278787 0.0268686
#>        TW        TY        WE        WQ        WR        WT        WW
#> 0.0260606 0.0272727 0.0280808 0.0261616 0.0257575 0.0266666 0.0287878
#>        WY        YE        YQ        YR        YT        YW        YY
#> 0.0283838 0.0280808 0.0291919 0.0290909 0.0258585 0.0277777 0.0273737
```

5 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 in the advanced usage vignette.

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; see the advanced usage vignette. 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).

5.1 Regular scanning

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 must ask, what even constitutes a mismatch? The answer to this question is much more difficult in cases such as these. 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 advanced usage vignette for details on motif P-values).

```
motif_pvalue(m2, pvalue = 0.001, progress = FALSE)
#> [1] 6.531
```

5.2 Higher-order scanning

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. For more details on the `multifreq` slot, see the advanced usage vignette.

```
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:

```
head(scan_sequences(motif.k2, ArabidopsisPromoters, RC = TRUE, verbose = 0,
                    threshold = 0.9, threshold.type = "logodds",
                    progress = FALSE))
#>   motif  sequence start stop score max.score score.pct   match strand
```

```
#> 1 motif AT4G28150   621  627  9.08  9.081843  100.2857 CTAAACC      +
#> 2 motif AT1G19380   139  145  9.08  9.081843  100.2857 CTTATCC      +
#> 3 motif AT1G19380   204  210  9.08  9.081843  100.2857 CTAAACC      +
#> 4 motif AT1G03850   203  209  9.08  9.081843  100.2857 CTAATCC      +
#> 5 motif AT5G01810   821  827  9.08  9.081843  100.2857 CATATCC      +
#> 6 motif AT5G01810   840  846  9.08  9.081843  100.2857 CAAATCC      +
```

Using 2-letter information to scan:

```
head(scan_sequences(motif.k2, ArabidopsisPromoters, use.freq = 2, RC = TRUE,
                    threshold = 0.9, threshold.type = "logodds",
                    verbose = 0, progress = FALSE))
#>   motif  sequence start stop  score max.score score.pct   match strand
#> 1 motif AT4G12690   938  945 17.827  17.82933  102.5061 CAAAACC      +
#> 2 motif AT2G37950   751  758 17.827  17.82933  102.5061 CAAAACC      +
#> 3 motif AT1G49840   959  966 17.827  17.82933  102.5061 CTTTTCC      +
#> 4 motif AT1G77210   184  191 17.827  17.82933  102.5061 CAAAACC      +
#> 5 motif AT1G77210   954  961 17.827  17.82933  102.5061 CAAAACC      +
#> 6 motif AT3G57640   917  924 17.827  17.82933  102.5061 CTTTTCC      +
```

As an aside: the previous example involved calling `create_motif()` and `add_multifreq()` separately. In this case however this could have been simplified to just calling `create_motif()` and using the `add.multifreq` option:

```
library(universalmotif)
library(Biostrings)

sequences <- DNAStringSet(rep(c("CAAAACC", "CTTTTCC"), 3))
motif <- create_motif(sequences, add.multifreq = 2:3)
```

6 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,
                       verbose = 0, progress = FALSE, threshold = 0.8,
                       threshold.type = "logodds")

res <- motif_peaks(hits$start,
```

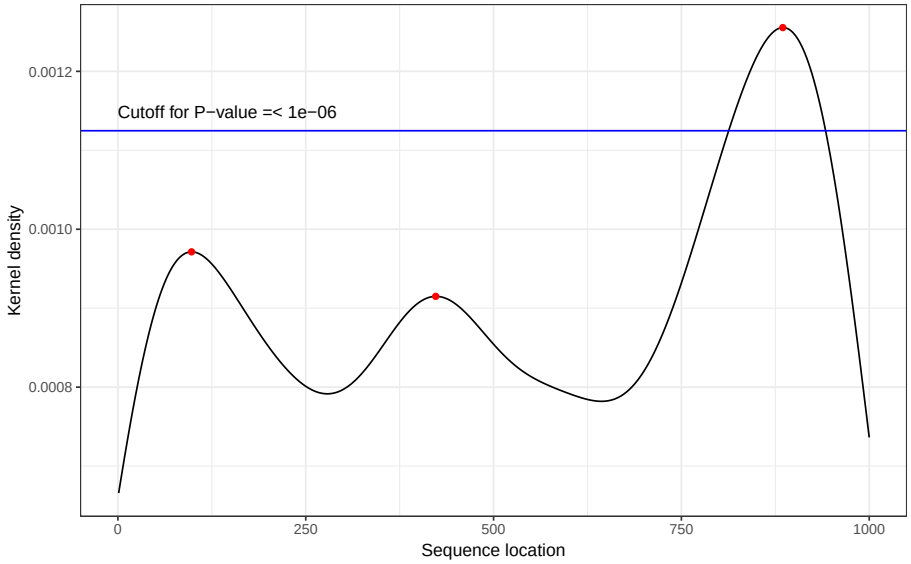
```
                      seq.length = unique(width(ArabidopsisPromoters)),
                      seq.count = length(ArabidopsisPromoters))

## Significant peaks:
res$Peaks
#>   Peak Pval
#> 1  885    0
```

Using the datasets provided in this package, a significant motif peak was found at 885 bases in the input promoters, or 115 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.

Session info

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

Sequence utilities

```
#> [4] LC_NUMERIC=C
#> [5] LC_TIME=English_United States.1252
#>
#> attached base packages:
#> [1] stats4    parallel  stats     graphics  grDevices utils     datasets
#> [8] methods   base
#>
#> other attached packages:
#>  [1] TFBSTools_1.22.0     ggplot2_3.1.1        ggtree_1.16.0
#>  [4] MotifDb_1.26.0       Biostrings_2.52.0    XVector_0.24.0
#>  [7] IRanges_2.18.0       S4Vectors_0.22.0     BiocGenerics_0.30.0
#> [10] universalmotif_1.2.1 BiocStyle_2.12.0
#>
#> loaded via a namespace (and not attached):
#>   [1] VGAM_1.1-1                  colorspace_1.4-1
#>   [3] grImport2_0.1-5             GenomicRanges_1.36.0
#>   [5] base64enc_0.1-3             rGADEM_2.32.0
#>   [7] bit64_0.9-7                 AnnotationDbi_1.46.0
#>   [9] splines_3.6.0               R.methodsS3_1.7.1
#>  [11] motifStack_1.28.0           knitr_1.22
#>  [13] ade4_1.7-13                 jsonlite_1.6
#>  [15] splitstackshape_1.4.8       Rsamtools_2.0.0
#>  [17] seqLogo_1.50.0              gridBase_0.4-7
#>  [19] annotate_1.62.0             GO.db_3.8.2
#>  [21] png_0.1-7                   R.oo_1.22.0
#>  [23] BiocManager_1.30.4          readr_1.3.1
#>  [25] compiler_3.6.0              httr_1.4.0
#>  [27] rvcheck_0.1.3               assertthat_0.2.1
#>  [29] Matrix_1.2-17               lazyeval_0.2.2
#>  [31] htmltools_0.3.6             tools_3.6.0
#>  [33] gtable_0.3.0                glue_1.3.1
#>  [35] TFMPvalue_0.0.8             GenomeInfoDbData_1.2.1
#>  [37] reshape2_1.4.3              dplyr_0.8.0.1
#>  [39] tinytex_0.12                Rcpp_1.0.1
#>  [41] Biobase_2.44.0              Logolas_1.8.0
#>  [43] ape_5.3                     nlme_3.1-139
#>  [45] rtracklayer_1.44.0          ggseqlogo_0.1
#>  [47] gbRd_0.4-11                 xfun_0.6
#>  [49] CNEr_1.20.0                 stringr_1.4.0
#>  [51] ps_1.3.0                    poweRlaw_0.70.2
#>  [53] gtools_3.8.1                XML_3.98-1.19
#>  [55] zlibbioc_1.30.0             MASS_7.3-51.4
#>  [57] scales_1.0.0                BSgenome_1.52.0
#>  [59] hms_0.4.2                   SummarizedExperiment_1.14.0
#>  [61] RColorBrewer_1.1-2          yaml_2.2.0
#>  [63] memoise_1.1.0               MotIV_1.40.0
#>  [65] SQUAREM_2017.10-1           stringi_1.4.3
#>  [67] RSQLite_2.1.1               highr_0.8
#>  [69] tidytree_0.2.4              caTools_1.17.1.2
#>  [71] BiocParallel_1.18.0         bibtex_0.4.2
#>  [73] GenomeInfoDb_1.20.0         Rdpack_0.11-0
```

```
#>  [75] rlang_0.3.4                pkgconfig_2.0.2
#>  [77] matrixStats_0.54.0         bitops_1.0-6
#>  [79] evaluate_0.13              lattice_0.20-38
#>  [81] purrr_0.3.2                htmlwidgets_1.3
#>  [83] GenomicAlignments_1.20.0   treeio_1.8.0
#>  [85] labeling_0.3               bit_1.1-14
#>  [87] processx_3.3.1             tidyselect_0.2.5
#>  [89] plyr_1.8.4                 magrittr_1.5
#>  [91] bookdown_0.9               R6_2.4.0
#>  [93] DelayedArray_0.10.0        DBI_1.0.0
#>  [95] pillar_1.3.1               withr_2.1.2
#>  [97] KEGGREST_1.24.0            RCurl_1.95-4.12
#>  [99] tibble_2.1.1               crayon_1.3.4
#> [101] rmarkdown_1.12             jpeg_0.1-8
#> [103] grid_3.6.0                 data.table_1.12.2
#> [105] blob_1.1.1                 digest_0.6.18
#> [107] xtable_1.8-4               tidyr_0.8.3
#> [109] R.utils_2.8.0              munsell_0.5.0
#> [111] DirichletMultinomial_1.26.0
```

References

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).