Sequence manipulation and scanning

Benjamin Jean-Marie Tremblay[*1]

[1]University of Waterloo, Waterloo, Canada

*b2tremblay@uwaterloo.ca

12 February 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.

When creating DNA and RNA sequences, there is additionally the option to set dinucleotide or trinucleotide probabilities. In these cases the sequences are constructed by calling `sample()` for every letter added, with the letter probabilities being dependent on which letter(s) precede that position. Unfortunately these routines consist of looping R code, which as a result makes sequence generation a bit slow in this regard.

```
library(universalmotif)
library(Biostrings)

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

sequences.dna <- create_sequences(seqnum = 500,
                                  monofreqs = c(0.3, 0.2, 0.2, 0.3))
## writeXStringSet(sequences.dna, "dna.fasta")
sequences.dna
#>   A DNAStringSet instance of length 500
#>       width seq
#>   [1]   100 TCTAACACGACATTAACAGCCTACTTTCTAT...ATTTGCGGACAAGTCAGGTAATCAACGTCGA
#>   [2]   100 AACTAATAATAAGACATGTCTCTATCCATAT...GCGAATAGAGAATTGGGGATTATTCTCTCTC
#>   [3]   100 CTGCATACTAATTTTACTCCTGTGCGATGAC...TTCAATGTTGAGAAATGATGCTGATTACCTA
#>   [4]   100 TGTAGTGAGAATATGGGGGAGTCCGAAAGTA...GATATTAGGCTTTCGGGACAGGCGAACAACA
#>   [5]   100 TCTGGCGTTACTCTTGAAATATCCAACTAAA...TAGATGGAGTTCTAAGCCAGGAGTTACCCGC
#>   ...   ... ...
#> [496]   100 TTCTGTACAAATTTGGATCCGTTATCGAATA...CAGTTTGATATATCTCTCAACCAGAGTAGAA
#> [497]   100 CCTTGTAAGTACACAGCCATTTTTGTACTTA...GCAAATACCAACTACTTCATTCTGCTCTCTC
#> [498]   100 GGAATGTACATCCGTATGTAGGATCACTTAG...ATTGTTTCGTGACGTTCTTATCTATATACAG
#> [499]   100 CTGTCCAATTCAGAGGGGATCTCTAATCTGT...GGGACTGATCGAGTCTAAATTCCAGAATACT
#> [500]   100 TTATATCTATACTGCTGGTTCTCGGCTTGTA...ACACCGTCACCTAGCGCTCCACACAATTTAT

## Amino acid:
```

```
create_sequences(alphabet = "AA")
#>   A AAStringSet instance of length 100
#>       width seq
#>   [1]   100 VWDDSTTGHNRHIQTRPPWYHWAYCLISCSF...TGGNLHTMHHDVSVCIGPLYQGHFSGWYECA
#>   [2]   100 WEHKDDDKRMKILSTKPQNTYSASMLMSEEF...CTLCNVHCEPYSNLNRWARERIVLRCHITVF
#>   [3]   100 DICDASGYERNLDTYNTRAVPETDKVDSELQ...IVKEKAKWNCANQYNYRHERTQKSSFYKLAT
#>   [4]   100 WDVWRKHGPDGSERRVTDYPLTKRDKLEQRA...VQMRTYGRFNHEKDNSHLKEGCCEVEQFWWH
#>   [5]   100 VETVDDCNLPQEIMHVNNNNQAFRFCVLHNV...AEHQNKFQQHNQSRCDVNADYVLRQPTEFHY
#>   ...   ... ...
#>  [96]   100 APGRRNKTVDVGNFNHGLAEEIWNEHHNQMM...CLKMHLDRANWDYTFTPTDNIRWAINDWTKT
#>  [97]   100 MHRWDPNWWNNKRCFSTDGMKSTKNNDHNMK...NLCPNNDIAFRLEAESFQDIQYTGTRLEQEF
#>  [98]   100 TGNINCDHYYNYNTLMRKETEYLWDRKGFPW...WMCQLISKDVTTGATKDFHENEQWLRTHMAG
#>  [99]   100 GQAFFITWFEQWFMQRGWMPMRWWIKKARDF...TDTAWEKPINKNVGGCFNVIWIGWKRQIHAQ
#> [100]   100 TRCTGCSAAFAYCINVTCWIVHVKMDAFWVL...CQAQVWILYMCFGMHECFQKMTCFEELMPHN

## Any set of characters can be used

create_sequences(alphabet = paste(letters, collapse = ""))
#>   A BStringSet instance of length 100
#>       width seq
#>   [1]   100 cryvlilvcadoutrrntrfscsjnncxkgg...xakzppamebjcdpykbvlngdumtvrexlq
#>   [2]   100 lgwpybhnuhbmirwnvqrmlybifpxuzji...yfqluvxnerufsrcolelyzdwbbqwiseb
#>   [3]   100 edojyspmcuisjdwfufvjnjhegusirqo...gbezazsgxjkdnrtyvdavlahstuvbjms
#>   [4]   100 ejrmyvqjkfcqnxiiptbtfggyguzktcz...wwdlyotlcnbrlbdsstrioolmqgbogfc
#>   [5]   100 uclvvhjlzqbvevtxmqzvabrjrtgkyrw...xwodxkavrbmimjanvypjynsnjmravjc
#>   ...   ... ...
#>  [96]   100 aanczsdoedzzaymtfbpksghjvsccvad...kbtjyngruqykhiqlszbqkyqhormodgn
#>  [97]   100 zgcrelksepjeyfpdmuxywrzzvhzregn...vbrapvdnewwreuvwmzsuymovnrpseid
#>  [98]   100 rskhqmzwyfhvbardmejoqeurztktjfe...fakkajujqtevftqjgegvzzjloztfspk
#>  [99]   100 faffcowhthvctcfoebbonuaabiisfab...vnidmytucdxvwxcjywblrfzycuiqepv
#> [100]   100 fkmfnlydhfkqmwpiehbeisbtnimipcs...xargeexmsfnjfwyqgoarwywhjvzairg
```

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: `markov`, `linear`, and `random`. The first method, `markov` (which is only available for DNA/RNA sequences with `k = c(2, 3)`) 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 *approximately* similar `k`-let frequency. As a result the counts for the individual letters will likely be different.

The second 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).

The third method `random` is an attempt to fix the sub-sequence issue from the `linear` method, though it is imperfect. In this case `k`-let sub-sequences are pulled out from the original sequence at random; meaning that they will be different every time, as opposed to the `linear` method. However, due to the fact that the sub-sequences are pulled out at random, this will leave small letter islands smaller then `k`. A few strategies are available to deal with these, see `?shuffle_sequences`.

```
library(universalmotif)
data(ArabidopsisPromoters)

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

markov <- shuffle_sequences(ArabidopsisPromoters, k = 2, method = "markov")
linear <- shuffle_sequences(ArabidopsisPromoters, k = 2, method = "linear")
random <- shuffle_sequences(ArabidopsisPromoters, k = 2, method = "random")
```

Let us compare how the methods perform:

```
o.letter <- colSums(oligonucleotideFrequency(ArabidopsisPromoters,
                                             1, as.prob = FALSE))
m.letter <- colSums(oligonucleotideFrequency(markov, 1, as.prob = FALSE))
l.letter <- colSums(oligonucleotideFrequency(linear, 1, as.prob = FALSE))
r.letter <- colSums(oligonucleotideFrequency(random, 1, as.prob = FALSE))

data.frame(original=o.letter, markov=m.letter,
           linear=l.letter, random=r.letter)
#>   original markov linear random
#> A    17384  17344  17384  17384
#> C     8081   8044   8081   8081
#> G     7583   7638   7583   7583
#> T    16952  16974  16952  16952

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

```
data.frame(original=o.counts, markov=m.counts,
           linear=l.counts, random=r.counts)
#>    original markov linear random
#> AA     6893   6850   6438   6333
#> AC     2614   2624   2708   2742
#> AG     2592   2607   2616   2663
#> AT     5276   5244   5601   5632
#> CA     3014   3000   2857   2865
#> CC     1376   1372   1311   1341
#> CG     1051   1032   1186   1160
#> CT     2621   2632   2718   2701
#> GA     2734   2739   2701   2684
#> GC     1104   1105   1188   1181
#> GG     1176   1202   1140   1143
#> GT     2561   2587   2546   2567
#> TA     4725   4738   5371   5484
#> TC     2977   2936   2868   2811
#> TG     2759   2792   2626   2608
#> TT     6477   6490   6075   6035
```

4 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 advanged 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).

4.1 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] 4.86
```

4.2 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 = 0.0001, progress = FALSE))
#>   motif  sequence start stop score max.score score.pct   match strand
#> 1 motif AT4G28150   621  627  9.08  9.081843  99.97971 CTAAACC      +
#> 2 motif AT1G19380   139  145  9.08  9.081843  99.97971 CTTATCC      +
#> 3 motif AT1G19380   204  210  9.08  9.081843  99.97971 CTAAACC      +
#> 4 motif AT1G03850   203  209  9.08  9.081843  99.97971 CTAATCC      +
#> 5 motif AT5G01810   821  827  9.08  9.081843  99.97971 CATATCC      +
#> 6 motif AT5G01810   840  846  9.08  9.081843  99.97971 CAAATCC      +
```

Using 2-letter information to scan:

```
head(scan_sequences(motif.k2, ArabidopsisPromoters, use.freq = 2, RC = TRUE,
                    verbose = 0, progress = FALSE))
#>   motif  sequence start stop score max.score score.pct   match strand
#> 1 motif AT4G28150    15   22 3.445  18.99872  18.13280 AAAAACC      +
#> 2 motif AT4G28150   621  628 3.445  18.99872  18.13280 CTAAACC      +
#> 3 motif AT4G28150   636  643 2.445  18.99872  12.86929 CAAAACT      +
#> 4 motif AT4G28150   868  875 2.445  18.99872  12.86929 CAAAACA      +
#> 5 motif AT4G28150   910  917 2.445  18.99872  12.86929 CTTTTCG      +
#> 6 motif AT1G19380   204  211 3.445  18.99872  18.13280 CTAAACC      +
```

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)
```

Session info

```
#> R version 3.5.2 (2018-12-20)
#> Platform: x86_64-pc-linux-gnu (64-bit)
#> Running under: Ubuntu 16.04.5 LTS
#>
#> Matrix products: default
#> BLAS: /home/biocbuild/bbs-3.8-bioc/R/lib/libRblas.so
#> LAPACK: /home/biocbuild/bbs-3.8-bioc/R/lib/libRlapack.so
#>
#> locale:
#>  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
#>  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
#>  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
#>  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
#>  [9] LC_ADDRESS=C               LC_TELEPHONE=C
#> [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
#>
#> attached base packages:
#> [1] stats4    parallel  stats     graphics  grDevices utils     datasets
#> [8] methods   base
#>
#> other attached packages:
#>  [1] TFBSTools_1.20.0      bindrcpp_0.2.2        MotifDb_1.24.1
#>  [4] Biostrings_2.50.2     XVector_0.22.0        IRanges_2.16.0
#>  [7] S4Vectors_0.20.1      BiocGenerics_0.28.0   universalmotif_1.0.20
#> [10] BiocStyle_2.10.0
#>
#> loaded via a namespace (and not attached):
#>   [1] VGAM_1.0-6                  colorspace_1.4-0
#>   [3] ggtree_1.14.6               GenomicRanges_1.34.0
#>   [5] rGADEM_2.30.0               bit64_0.9-7
#>   [7] AnnotationDbi_1.44.0        splines_3.5.2
#>   [9] R.methodsS3_1.7.1           motifStack_1.26.0
#>  [11] knitr_1.21                  ade4_1.7-13
#>  [13] jsonlite_1.6                splitstackshape_1.4.6
#>  [15] Rsamtools_1.34.1            seqLogo_1.48.0
#>  [17] gridBase_0.4-7              annotate_1.60.0
#>  [19] GO.db_3.7.0                 png_0.1-7
#>  [21] R.oo_1.22.0                 grImport_0.9-1.1
#>  [23] BiocManager_1.30.4          readr_1.3.1
#>  [25] compiler_3.5.2              httr_1.4.0
#>  [27] rvcheck_0.1.3               assertthat_0.2.0
#>  [29] Matrix_1.2-15               lazyeval_0.2.1
#>  [31] htmltools_0.3.6             tools_3.5.2
#>  [33] gtable_0.2.0                glue_1.3.0
#>  [35] TFMPvalue_0.0.8             GenomeInfoDbData_1.2.0
#>  [37] reshape2_1.4.3              dplyr_0.7.8
#>  [39] tinytex_0.10                Rcpp_1.0.0
#>  [41] Biobase_2.42.0              Logolas_1.6.0
```

```
#>  [43] ape_5.2                     nlme_3.1-137
#>  [45] rtracklayer_1.42.1          ggseqlogo_0.1
#>  [47] gbRd_0.4-11                 xfun_0.4
#>  [49] CNEr_1.18.1                 stringr_1.4.0
#>  [51] ps_1.3.0                    poweRlaw_0.70.2
#>  [53] gtools_3.8.1                XML_3.98-1.17
#>  [55] zlibbioc_1.28.0             MASS_7.3-51.1
#>  [57] scales_1.0.0                BSgenome_1.50.0
#>  [59] hms_0.4.2                   SummarizedExperiment_1.12.0
#>  [61] RColorBrewer_1.1-2          yaml_2.2.0
#>  [63] memoise_1.1.0               ggplot2_3.1.0
#>  [65] MotIV_1.38.0                SQUAREM_2017.10-1
#>  [67] stringi_1.2.4               RSQLite_2.1.1
#>  [69] highr_0.7                   tidytree_0.2.3
#>  [71] caTools_1.17.1.1            BiocParallel_1.16.6
#>  [73] bibtex_0.4.2                GenomeInfoDb_1.18.2
#>  [75] Rdpack_0.10-1               rlang_0.3.1
#>  [77] pkgconfig_2.0.2             matrixStats_0.54.0
#>  [79] bitops_1.0-6                evaluate_0.13
#>  [81] lattice_0.20-38             purrr_0.3.0
#>  [83] bindr_0.1.1                 htmlwidgets_1.3
#>  [85] GenomicAlignments_1.18.1    treeio_1.6.2
#>  [87] labeling_0.3                bit_1.1-14
#>  [89] processx_3.2.1              tidyselect_0.2.5
#>  [91] plyr_1.8.4                  magrittr_1.5
#>  [93] bookdown_0.9                R6_2.3.0
#>  [95] DelayedArray_0.8.0          DBI_1.0.0
#>  [97] pillar_1.3.1                KEGGREST_1.22.0
#>  [99] RCurl_1.95-4.11             tibble_2.0.1
#> [101] crayon_1.3.4                rmarkdown_1.11
#> [103] grid_3.5.2                  data.table_1.12.0
#> [105] blob_1.1.1                  digest_0.6.18
#> [107] xtable_1.8-3                tidyr_0.8.2
#> [109] R.utils_2.7.0               munsell_0.5.0
#> [111] DirichletMultinomial_1.24.1
```

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).