

Sequence manipulation and scanning

Benjamin Jean-Marie Tremblay*

14 March 2020

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	Creating random sequences	2
3	Calculating sequence background	3
4	Shuffling sequences	4
5	Miscellaneous string utilities	5
6	Scanning sequences for motifs	6
7	Enrichment analyses	8
8	Testing for motif positional preferences in sequences	9
9	Motif discovery with MEME	10
	Session info	12
	References	14

1 Introduction

This vignette goes through generating your own sequences from a specified background model, shuffling sequences whilst maintaining a certain k -let size, and the scanning of sequences and scoring of motifs. For an introduction to sequence motifs, see the introductory vignette. For a basic overview of available motif-related functions, see the motif manipulation vignette. For a discussion on motif comparisons and P-values, see the motif comparisons and P-values vignette.

*b2tremblay@uwaterloo.ca

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 its simplest form generates a set of letters in random order, 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 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 CTGATTCTCTATTCTACCCGGAATAAACTTCT...GAGTGTGGTTAATAGACGGCGTCAACTAGAG
#> [2] 100 TTCCTGTCTCCGGGTATTGCACCGGAGTTTT...AAATATCCCTCAGGCACTACGAGTCCGTAAAC
#> [3] 100 ATATTTAAAAACAAATACATAGCGGTATATTCT...GATTTTAAATGACTCGGGATTATCATGTTATAA
#> [4] 100 GGTTTTATTCTGAACTATCATAGTCACAGTTTT...ACGTCTTTGCTGAATATGCGCCAGTTAGACAT
#> [5] 100 TGCATTATTTTTTGTCTCGCTAACCCCTTTCT...GCATATGCAAGGAAAGAAATAGTATATGGATCC
#> ... ..
#> [496] 100 CTGTGTTCAAAAATTGACATTAAAATATCTCGG...GACATCAAAACCCGAATTTAAAGAGTAAAGG
#> [497] 100 TTCAGTTTTGTATGACATCAGTCATACATTAAG...AAAACCTTTCTGGATTCCCCGTTAGTATCGGC
#> [498] 100 ATAAGTGCGTGCGAGCAATCAAGCTCCGATAGG...TCTATCGGCACGTATAGTTATCGCTTGTCATA
#> [499] 100 CGTGGTATCCAGAATCATAGTTTGGGCTGGAAG...ACCAGCAAAAATATATTAGTATGTAATTCTTG
#> [500] 100 TGCTCGAAAGAGATAACAGTGCTTGTGATGAGG...TGTAACTTTTTTCGCATGACGGCAACATGCTC

## Amino acid:

create_sequences(alphabet = "AA")
#> AAStringSet object of length 100:
#>      width seq
#> [1] 100 PGNWIIGFPADFRYRNYCNMQFSQPCTRSADAM...SRKYLRFSYALNCQTKRETFYFQKPMAWSCTL
#> [2] 100 GMDTRRNLFCKLYLDWEEAILMHGEPKMGCF...NKTYWLLNWAYDEIQTKIPLYLHVGCCTMDQY
#> [3] 100 VTRRC DVRVCHPFYERVGSNCRGYWFLDGKDHP...IDFYIFRIVALRFCMGDMLRYRAGWPDRHFMK
#> [4] 100 MCGPLLDYLDKTYWGTIICRYAPNHGTANEKD...ETPYTYEVCYGHSHQTRGWYQRNEDPCGIW
#> [5] 100 EIVMTVKEDDMAQWPVSLYNKFSGFKNSRFMQ...YMAYGRFYTCLVKKECMVCEWFIEFQEMVHFI
#> ... ..
#> [96] 100 NIWYRLWPKVLYYQRMNNSTNLRTMIKNNAI...NLVLTVDVNMWWSEFEQWNRPKKLWPMGYNHV
#> [97] 100 EQLYHCSEFLAQRYIHLQEHLTFLIWICERPDW...IEGLGPIQMNKMTVCNKCIYPQDWNENESPEI
#> [98] 100 TWCVRKCKVLDVLYDKSSSTECYDAYESWVQFL...EVQLSIPLLNWCWNVADGFFPWSHFRNCMQAV
#> [99] 100 KFPCSHTQLMFCFYTLIVIHVHRSPCYMQARHA...YNCKFD TGLNKPAFRKTKAKNELSVGPYHSVH
#> [100] 100 CLFKQEPVCMHGWWNAHYVNNLKGETFKDSKM...SGLKQTCAPWFCWNVMP TQNKDEMSQVCTR
```

Any set of characters can be used

```

create_sequences(alphabet = paste0(letters, collapse = ""))
#> BStringSet object of length 100:
#>      width seq
#> [1] 100 hhboqxjyvyustttxtmwybyoukavckktvqd...tcutnaqdzrcegkoqwsmtyaxcvnsaeax
#> [2] 100 ooddhvsuwqplmnmymyswdwcquaquvmqh...neombbggzjejmudgtalzmwaufrrblbibv
#> [3] 100 vvfstbtlukefhsgkoufvmfalgefflxk...hgjgpbwjzbgoseswqfelfuasinpdbmbt
#> [4] 100 cchhprltgsfwyqazxslshthepbgjpqygon...bjdzdcnmztisyohnmgxyytaplicwcqcr
#> [5] 100 jjjvgousbqapruotjhqjrsczbbblzbsber...vlysrcdpzkkxeywdjipkrranoeqocvcp
#> ... ..
#> [96] 100 woemtukcrzfyrufsigrebgaftjyhbxf...mommdgmyslmheidueifmopoyaskdqw
#> [97] 100 dugbkttanxarlodlucpgzuvpfolisusv...gqhfadwpyknrnoxtgrbsfnplbwgceuxj
#> [98] 100 kciqbrdzivujehbfhyniyiqzgntdnmy...asbzoensycpwtymjnhueylqiesuvezxh
#> [99] 100 rjjespmxdupcabyytulkwxmjgeqdnhic...uwscedyttrazibakjnrjgghninfdyf
#> [100] 100 yqltjmvwyskvquwsfrjmulhugzsnayadt...nxqmqtzyltffsqhlfdkhqdkgjvgfhyd

```

3 Calculating sequence background

Sequence backgrounds can be retrieved for DNA and RNA sequences with `oligonucleotideFrequency()` from "Biostrings". Unfortunately, no such `Biostrings` function exists for other sequence alphabets. The `universalmotif` package provides `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 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
#> 0.25060000 0.24630000 0.25470000 0.24840000 0.06414141 0.06636364 0.05969697
#>      AT      CA      CC      CG      CT      GA      GC
#> 0.06030303 0.06161616 0.05878788 0.06353535 0.06242424 0.06363636 0.06222222
#>      GG      GT      TA      TC      TG      TT
#> 0.06787879 0.06090909 0.06161616 0.05898990 0.06343434 0.06444444

## 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
#> 0.16060000 0.16460000 0.16790000 0.16450000 0.17210000 0.17030000 0.02636364
#>      EQ      ER      ET      EW      EY      QE      QQ
#> 0.02424242 0.02575758 0.02565657 0.03080808 0.02777778 0.02767677 0.02949495
#>      QR      QT      QW      QY      RE      RQ      RR
#> 0.02646465 0.02545455 0.02848485 0.02696970 0.02767677 0.02747475 0.02737374
#>      RT      RW      RY      TE      TQ      TR      TT
#> 0.02676768 0.02797980 0.03070707 0.02686869 0.02575758 0.03101010 0.02606061
#>      TW      TY      WE      WQ      WR      WT      WW
#> 0.02707071 0.02757576 0.02656566 0.02838384 0.02828283 0.02979798 0.03010101
#>      WY      YE      YQ      YR      YT      YW      YY
#> 0.02898990 0.02545455 0.02919192 0.02909091 0.03070707 0.02767677 0.02828283

```

4 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 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 in order for this to be possible, 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 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, as.prob = FALSE, list.out = FALSE)
e.letter <- get_bkg(euler, 1, as.prob = FALSE, list.out = FALSE)
m.letter <- get_bkg(markov, 1, as.prob = FALSE, list.out = FALSE)
l.letter <- get_bkg(linear, 1, as.prob = FALSE, list.out = FALSE)

data.frame(original=o.letter, euler=e.letter, markov=m.letter, linear=l.letter)
#>   original euler markov linear
#> A    17384 17384  17605 17384
#> C     8081  8081   8124  8081
```

```

#> G      7583 7583 7572 7583
#> T      16952 16952 16749 16952

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

data.frame(original=o.counts, euler=e.counts, markov=m.counts, linear=l.counts)
#>      original euler markov linear
#> AA      6893 6893 6172 6480
#> AC      2614 2614 2880 2692
#> AG      2592 2592 2634 2633
#> AT      5276 5276 5901 5565
#> CA      3014 3014 2876 2941
#> CC      1376 1376 1242 1312
#> CG      1051 1051 1272 1164
#> CT      2621 2621 2727 2655
#> GA      2734 2734 2629 2717
#> GC      1104 1104 1262 1146
#> GG      1176 1176 1199 1182
#> GT      2561 2561 2474 2533
#> TA      4725 4725 5911 5229
#> TC      2977 2977 2730 2925
#> TG      2759 2759 2461 2594
#> TT      6477 6477 5630 6182

```

5 Miscellaneous string utilities

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

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

```

library(universalmotif)

string <- "DASDSDSASDSSA"

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] "DSDDSDSSASDASA"

```

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

```
library(universalmotif)

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

6 Scanning sequences for motifs

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

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

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

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

```
library(universalmotif)
m1 <- create_motif("TATATATATA", nsites = 50, type = "PWM", pseudocount = 1)
m2 <- matrix(c(0.10,0.27,0.23,0.19,0.29,0.28,0.51,0.12,0.34,0.26,
               0.36,0.29,0.51,0.38,0.23,0.16,0.17,0.21,0.23,0.36,
               0.45,0.05,0.02,0.13,0.27,0.38,0.26,0.38,0.12,0.31,
               0.09,0.40,0.24,0.30,0.21,0.19,0.05,0.30,0.31,0.08),
             byrow = TRUE, nrow = 4)
m2 <- create_motif(m2, alphabet = "DNA", type = "PWM")
m1["motif"]
#>           T           A           T           A           T           A           T
#> A -5.672425  1.978626 -5.672425  1.978626 -5.672425  1.978626 -5.672425
#> C -5.672425 -5.672425 -5.672425 -5.672425 -5.672425 -5.672425 -5.672425
#> G -5.672425 -5.672425 -5.672425 -5.672425 -5.672425 -5.672425 -5.672425
#> T  1.978626 -5.672425  1.978626 -5.672425  1.978626 -5.672425  1.978626
#>           A           T           A
#> A  1.978626 -5.672425  1.978626
#> C -5.672425 -5.672425 -5.672425
#> G -5.672425 -5.672425 -5.672425
#> T -5.672425  1.978626 -5.672425
m2["motif"]
#>           S           H           C           N           N           N
```

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

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

This thinking becomes impossible for the second example. In this case, mismatches are much less punishing, to the point that one could ask: what even constitutes a mismatch? The answer to this question is much more difficult in 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 comparisons and P-values vignette for details on motif P-values).

```
motif_pvalue(m2, pvalue = 0.001)
#> [1] 4.8531
```

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("CAAAACC", "CTTTTCC"), 3))
motif.k2 <- add_multifreq(motif.k2, sequences.k2)
```

Regular scanning:

```
head(scan_sequences(motif.k2, ArabidopsisPromoters, RC = TRUE,
                    threshold = 0.9, threshold.type = "logodds"))
#> Note: detected non-finite positional weights, normalising
#> Note: since this motif has a pseudocount of 0, 1 will be used
#> DataFrame with 6 rows and 12 columns
#>   motif motif.i sequence start stop score match
#>   <character> <integer> <character> <integer> <integer> <numeric> <character>
#> 1 motif 1 AT4G28150 621 627 9.08 CTAAACC
#> 2 motif 1 AT1G19380 139 145 9.08 CTTATCC
#> 3 motif 1 AT1G19380 204 210 9.08 CTAAACC
#> 4 motif 1 AT1G03850 203 209 9.08 CTAATCC
#> 5 motif 1 AT5G01810 821 827 9.08 CATATCC
#> 6 motif 1 AT5G01810 840 846 9.08 CAAATCC
#>   thresh.score min.score max.score score.pct strand
#>   <numeric> <numeric> <numeric> <numeric> <character>
```

```
#> 1      8.172 -19.649  9.08  100      +
#> 2      8.172 -19.649  9.08  100      +
#> 3      8.172 -19.649  9.08  100      +
#> 4      8.172 -19.649  9.08  100      +
#> 5      8.172 -19.649  9.08  100      +
#> 6      8.172 -19.649  9.08  100      +
```

Using 2-letter information to scan:

```
head(scan_sequences(motif.k2, ArabidopsisPromoters, use.freq = 2, RC = TRUE,
                    threshold = 0.9, threshold.type = "logodds"))
#> Note: detected non-finite positional weights, normalising
#> Note: since this motif has a pseudocount of 0, 1 will be used
#> DataFrame with 6 rows and 12 columns
#>      motif motif.i sequence start stop score match
#>   <character> <integer> <character> <integer> <integer> <numeric> <character>
#> 1      motif          1 AT4G12690     938    943  17.827 CAAAAC
#> 2      motif          1 AT2G37950     751    756  17.827 CAAAAC
#> 3      motif          1 AT1G49840     959    964  17.827 CTTTTC
#> 4      motif          1 AT1G77210     184    189  17.827 CAAAAC
#> 5      motif          1 AT1G77210     954    959  17.827 CAAAAC
#> 6      motif          1 AT3G57640     917    922  17.827 CTTTTC
#>   thresh.score min.score max.score score.pct strand
#>   <numeric> <numeric> <numeric> <numeric> <character>
#> 1      16.0443  -16.842   17.827    100      +
#> 2      16.0443  -16.842   17.827    100      +
#> 3      16.0443  -16.842   17.827    100      +
#> 4      16.0443  -16.842   17.827    100      +
#> 5      16.0443  -16.842   17.827    100      +
#> 6      16.0443  -16.842   17.827    100      +
```

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)
```

7 Enrichment analyses

The `universalmotif` package offers the ability to search for enriched motif sites in a set of sequences via `enrich_motifs()`. There is little complexity to this, as it simply runs `scan_sequences()` twice: once on a set of target sequences, and once on a set of background sequences. After which the results between the two sequences are collated and run through enrichment tests. The background sequences can be given explicitly, or else `enrich_motifs()` will create background sequences on its own by using `shuffle_sequences()` on the target sequences.

Let us consider the following basic example:

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



```

enrich_motifs(ArabidopsisMotif, ArabidopsisPromoters, shuffle.k = 3,
              threshold = 0.001, RC = TRUE)
#> DataFrame with 1 row and 11 columns
#>      motif motif.i target.hits target.seq.hits target.seq.count
#>      <character> <integer> <integer> <integer> <integer>
#> 1 YTTYTTTTTTYTTY 1      648      50      50
#>      bkg.hits bkg.seq.hits bkg.seq.count Pval Qual Eval
#>      <integer> <integer> <integer> <numeric> <numeric> <numeric>
#> 1      225      47      50 1.09692e-48 1.09692e-48 2.19383e-48

```

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.)

8 Testing for motif positional preferences in sequences

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

```

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

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

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

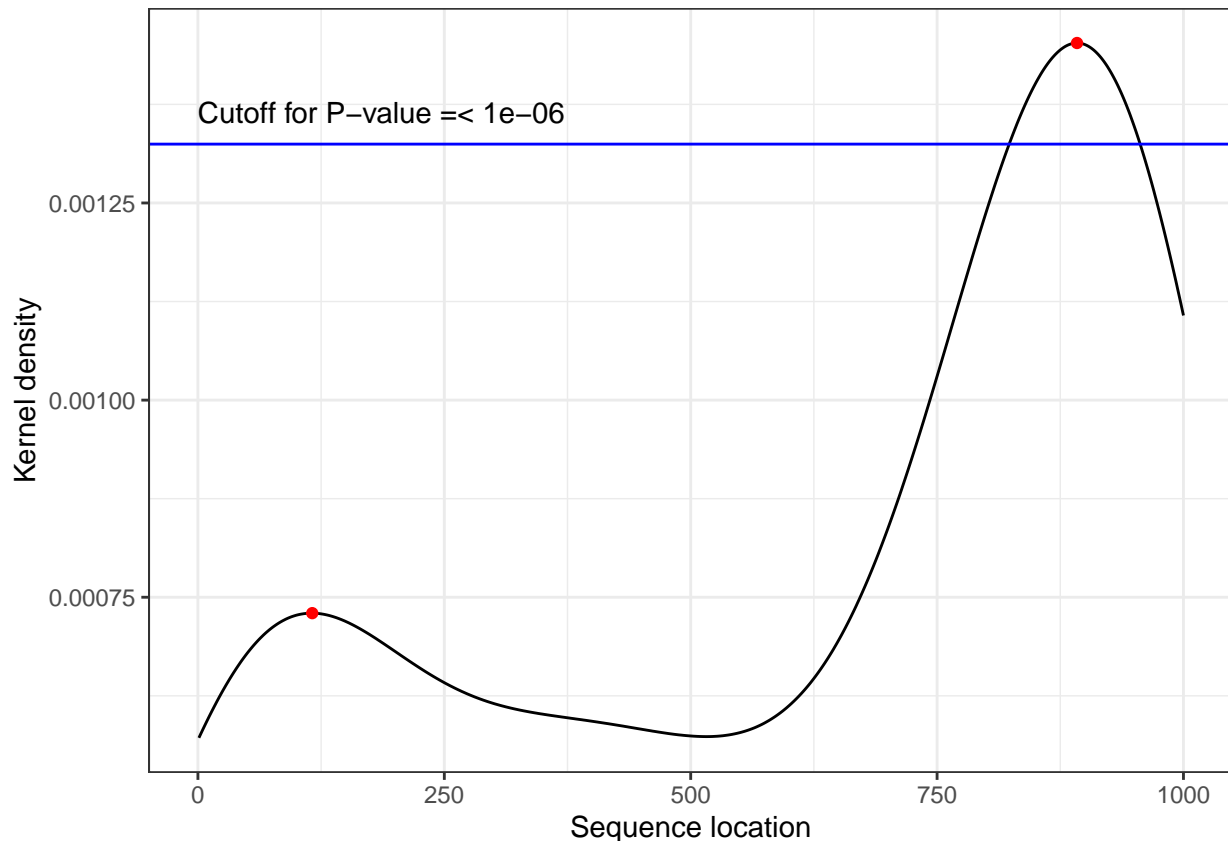
## Significant peaks:
res$Peaks
#> DataFrame with 1 row and 2 columns
#>      Peak Pval
#>      <numeric> <numeric>
#> 1      892 3.91482e-12

```

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

The function can also output a plot:

```
res$Plot
```



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

9 Motif discovery with MEME

The `universalmotif` package provides a simple wrapper to the powerful motif discovery tool **MEME** (Bailey and Elkan 1994). To run an analysis with **MEME**, all that is required is a set of `XStringSet` class sequences (defined in the `Biostrings` package), and `run_meme()` will take care of running the program and reading the output for use within R.

The first step is to check that R can find the **MEME** binary in your `$PATH` by running `run_meme()` without any parameters. If successful, you should see the default **MEME** help message in your console. If not, then you'll need to provide the complete path to the **MEME** binary. There are two options:

```
library(universalmotif)

## 1. Once per session: via `options()`
options(meme.bin = "/path/to/meme/bin/meme")

run_meme(...)

## 2. Once per run: via `run_meme()`
run_meme(..., bin = "/path/to/meme/bin/meme")
```

Now we need to get some sequences to use with `run_meme()`. At this point we can read sequences from disk or extract them from one of the Bioconductor `BSgenome` packages.

```

library(universalmotif)
data(ArabidopsisPromoters)

## 1. Read sequences from disk (in fasta format):

library(Biostrings)

# The following `read*()` functions are available in Biostrings:
# DNA: readDNAStringSet
# DNA with quality scores: readQualityScaledDNAStringSet
# RNA: readRNAStringSet
# Amino acid: readAAStringSet
# Any: readBStringSet

sequences <- readDNAStringSet("/path/to/sequences.fasta")

run_meme(sequences, ...)

## 2. Extract from a `BSgenome` object:

library(GenomicFeatures)
library(TxDb.Athaliana.BioMart.plantsmart28)
library(BSgenome.Athaliana.TAIR.TAIR9)

# Let us retrieve the same promoter sequences from ArabidopsisPromoters:
gene.names <- names(ArabidopsisPromoters)

# First get the transcript coordinates from the relevant `TxDb` object:
transcripts <- transcriptsBy(TxDb.Athaliana.BioMart.plantsmart28,
                             by = "gene")[gene.names]

# There are multiple transcripts per gene, we only care for the first one
# in each:

transcripts <- lapply(transcripts, function(x) x[1])
transcripts <- unlist(GRangesList(transcripts))

# Then the actual sequences:

# Unfortunately this is a case where the chromosome names do not match
# between the two databases

seqlevels(TxDb.Athaliana.BioMart.plantsmart28)
#> [1] "1" "2" "3" "4" "5" "Mt" "Pt"
seqlevels(BSgenome.Athaliana.TAIR.TAIR9)
#> [1] "Chr1" "Chr2" "Chr3" "Chr4" "Chr5" "ChrM" "ChrC"

# So we must first rename the chromosomes in `transcripts`:
seqlevels(transcripts) <- seqlevels(BSgenome.Athaliana.TAIR.TAIR9)

# Finally we can extract the sequences
promoters <- getPromoterSeq(transcripts,
                             BSgenome.Athaliana.TAIR.TAIR9,

```

```

upstream = 1000, downstream = 0)

run_meme(promoters, ...)

```

Once the sequences are ready, there are few important options to keep in mind. One is whether to conserve the output from MEME. The default is not to, but this can be changed by setting the relevant option:

```
run_meme(sequences, output = "/path/to/desired/output/folder")
```

The second important option is the search function (`objfun`). Some search functions such as the default `classic` do not require a set of background sequences, whilst some do (such as `de`). If you choose one of the latter, then you can either let MEME create them for you (it will shuffle the target sequences) or you can provide them via the `control.sequences` parameter.

Finally, choose how you'd like the data imported into R. Once the MEME program exits, `run_meme()` will import the results into R with `read_meme()`. At this point you can decide if you want just the motifs themselves (`readsites = FALSE`) or if you'd like the original sequence sites as well (`readsites = TRUE`, the default). Doing the latter gives you the option of generating higher order representations for the imported MEME motifs as shown here:

```

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

```

There are a wealth of other MEME options available, such as the number of desired motifs (`nmotifs`), the width of desired motifs (`minw`, `maxw`), the search mode (`mod`), assigning sequence weights (`weights`), using a custom alphabet (`alph`), and many others. See the output from `run_meme()` for a brief description of the options, or visit the online manual for more details.

Session info

```

#> R version 4.0.0 (2020-04-24)
#> Platform: x86_64-apple-darwin17.0 (64-bit)
#> Running under: macOS Mojave 10.14.6
#>
#> Matrix products: default
#> BLAS: /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRblas.dylib
#> LAPACK: /Library/Frameworks/R.framework/Versions/4.0/Resources/lib/libRlapack.dylib
#>
#> locale:
#> [1] C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
#>
#> attached base packages:
#> [1] stats4 parallel stats graphics grDevices utils datasets
#> [8] methods base
#>
#> other attached packages:
#> [1] TFBSTools_1.26.0 Logolas_1.12.0 dplyr_0.8.5
#> [4] ggtree_2.2.0 ggplot2_3.3.0 MotifDb_1.30.0
#> [7] GenomicRanges_1.40.0 GenomeInfoDb_1.24.0 Biostrings_2.56.0
#> [10] XVector_0.28.0 IRanges_2.22.1 S4Vectors_0.26.0
#> [13] BiocGenerics_0.34.0 universalmotif_1.6.2
#>
#> loaded via a namespace (and not attached):
#> [1] colorspace_1.4-1 grImport2_0.2-0
#> [3] ellipsis_0.3.0 base64enc_0.1-3

```

```

#> [5] aplot_0.0.4          rGADEM_2.36.0
#> [7] farver_2.0.3         bit64_0.9-7
#> [9] AnnotationDbi_1.50.0 R.methodsS3_1.8.0
#> [11] motifStack_1.32.0   knitr_1.28
#> [13] ade4_1.7-15         jsonlite_1.6.1
#> [15] splitstackshape_1.4.8 Rsamtools_2.4.0
#> [17] seqLogo_1.54.1      gridBase_0.4-7
#> [19] annotate_1.66.0      GO.db_3.11.0
#> [21] png_0.1-7           R.oo_1.23.0
#> [23] BiocManager_1.30.10 readr_1.3.1
#> [25] compiler_4.0.0      httr_1.4.1
#> [27] rvcheck_0.1.8       assertthat_0.2.1
#> [29] Matrix_1.2-18       lazyeval_0.2.2
#> [31] htmltools_0.4.0     tools_4.0.0
#> [33] gtable_0.3.0        glue_1.4.0
#> [35] TFMpvalue_0.0.8     GenomeInfoDbData_1.2.3
#> [37] reshape2_1.4.4      tinytex_0.22
#> [39] Rcpp_1.0.4.6        Biobase_2.48.0
#> [41] vctrs_0.2.4         ape_5.3
#> [43] nlme_3.1-147        rtracklayer_1.47.0
#> [45] ggseqlogo_0.1       gbrd_0.4-11
#> [47] xfun_0.13           CNEr_1.24.0
#> [49] stringr_1.4.0       ps_1.3.2
#> [51] lifecycle_0.2.0     powerlaw_0.70.6
#> [53] gtools_3.8.2        XML_3.99-0.3
#> [55] zlibbioc_1.34.0     MASS_7.3-51.6
#> [57] scales_1.1.0        BSgenome_1.56.0
#> [59] hms_0.5.3           SummarizedExperiment_1.18.1
#> [61] RColorBrewer_1.1-2  yaml_2.2.1
#> [63] memoise_1.1.0       MotIV_1.44.0
#> [65] stringi_1.4.6       RSQLite_2.2.0
#> [67] SQUAREM_2020.2     highr_0.8
#> [69] tidytree_0.3.3      caTools_1.18.0
#> [71] BiocParallel_1.22.0 bibtex_0.4.2.2
#> [73] Rdpack_0.11-1       rlang_0.4.6
#> [75] pkgconfig_2.0.3     matrixStats_0.56.0
#> [77] bitops_1.0-6        pracma_2.2.9
#> [79] evaluate_0.14       lattice_0.20-41
#> [81] purrr_0.3.4         htmlwidgets_1.5.1
#> [83] GenomicAlignments_1.24.0 treeio_1.12.0
#> [85] patchwork_1.0.0     labeling_0.3
#> [87] bit_1.1-15.2        processx_3.4.2
#> [89] tidyselect_1.0.0    plyr_1.8.6
#> [91] magrittr_1.5         bookdown_0.18
#> [93] R6_2.4.1            DelayedArray_0.14.0
#> [95] DBI_1.1.0           pillar_1.4.3
#> [97] withr_2.2.0         KEGGREST_1.28.0
#> [99] RCurl_1.98-1.2      tibble_3.0.1
#> [101] crayon_1.3.4        rmarkdown_2.1
#> [103] jpeg_0.1-8.1        grid_4.0.0
#> [105] data.table_1.12.8   blob_1.2.1
#> [107] digest_0.6.25       xtable_1.8-4
#> [109] tidyr_1.0.2         R.utils_2.9.2
#> [111] munsell_0.5.0       DirichletMultinomial_1.30.0

```

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 (6): 526–38.
- Bailey, T. L., and C. Elkan. 1994. "Fitting a Mixture Model by Expectation Maximization to Discover Motifs in Biopolymers." *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology* 2: 28–36.
- Fitch, Walter M. 1983. "Random Sequences." *Journal of Molecular Biology* 163 (2): 171–76.
- Jiang, M., J. Anderson, J. Gillespie, and M. Mayne. 2008. "uShuffle: A Useful Tool for Shuffling Biological Sequences While Preserving K-Let Counts." *BMC Bioinformatics* 9 (192).
- Propp, J. G., and D. W. Wilson. 1998. "How to Get a Perfectly Random Sample from a Generic Markov Chain and Generate a Random Spanning Tree of a Directed Graph." *Journal of Algorithms* 27: 170–217.