Suffix Array Kernel Smoothing for *de novo* discovery of correlative sequence motifs and multi-motif domains: the `sarks` package

Dennis Wylie, Hans A. Hofmann, Boris V. Zemelman

May 19, 2021

This vignette describes the R interface `sarks` to the java implementation of the **S**uffix **Ar**ray **K**ernel **S**moothing, or SArKS, algorithm for identification of correlative motifs and multi-motif domains (MMDs).

> If you use `sarks` in your work, please cite Wylie et al. [2019] (see references).

Contents

1   How to install `sarks`

The `sarks` package relies on `Java` (1.8 or greater) through use of the `rJava` package. Once both of these are installed and correctly configured, you can install `sarks` by running the following code within an R session:

```
> if(!requireNamespace("BiocManager", quietly = TRUE))
+     install.packages("BiocManager")
> BiocManager::install("sarks")
```

2   How to use `sarks`

2.1   Starting `sarks`

Because `sarks` is implemented using `rJava`, and because the default setting for the java virtual machine (JVM) heap space with `rJava` is quite low, you should initialize java with increased heap size *before loading* `sarks` (or any other `rJava`-dependent R packages):

```
> options(java.parameters = '-Xmx8G')  ## sets to 8 gigabytes: modify as needed
> library(rJava)
> .jinit()
> ## ----------------------------------------------------------------------
> ## once you've set the java.parameters and then called .jinit from rJava
> ## (making sure not to load any other rJava-dependent packages prior to
> ##  initializing the JVM with .jinit!),
> ## you are now ready to load the sarks library:
> ## ----------------------------------------------------------------------
> library(sarks)
```

2.2   Input

Aside from the specification of a few analysis parameters to be discussed below, `sarks` requires two pieces of input data:

sequences which may be specified as either a:

FASTA formatted text file (may be gzipped) or, equivalently, a

named character vector

scores which may be specified as either a:

named numeric vector (with names matching those of input sequences) or a

tsv two-column tab-delimited file (containing header line), with columns

1. containing names matching those of input sequences and

2. containing numeric scores assigned to those sequences

For the purposes of this vignette, we'll take the sequences to be the named character vector `simulatedSeqs` included in the `sarks` package. `simulatedSeqs` is a character vector of length 30—representing 30 different (fake) DNA sequences—each 250 characters in length:

```
> options(continue="  ")  ## for vignette formatting, can be ignored
> data(simulatedSeqs)
> length(simulatedSeqs)

[1] 30

> table(nchar(simulatedSeqs))

250
 30
```

Take a look at the first few characters of each sequence:

```
> vapply(simulatedSeqs, function(s) {paste0(substr(s, 1, 10), '...')}, '')
                0                1                2                3                4
"GCGGAGGCTG..." "CGTTGAATGT..." "AGTCAGTTCT..." "AGAGCTTCAG..." "GTTTCTGCCC..."
                5                6                7                8                9
"CTAAGGGCGA..." "ATTAGGTAAA..." "GCTCGGAGGA..." "TTCCTGCCTA..." "ACAATCTGCG..."
               10               11               12               13               14
"CCACAGCGTT..." "TGACGACGCG..." "GCGCACTAGC..." "TCAAAGTAGG..." "GGTACAATCA..."
               15               16               17               18               19
"TATGACACCG..." "CACTCGTATG..." "TGGTCTCGAC..." "GTCTCCCCGA..." "TACGAGGCTC..."
               20               21               22               23               24
"CGGACGCGTG..." "GATGTGCCAT..." "TGAAAGGAGA..." "TAATGTAATG..." "CATCGAGATG..."
               25               26               27               28               29
"CATACTGAGA..." "ACCAACAGTC..." "GCACGACAAA..." "GAAACAGAGG..." "GTTGATCTCA..."
```

Notice that the names of the simulated sequences are just the (string representations of) the numbers "0" through "29".

Now let's look at the `simulatedScores`:

```
> data(simulatedScores)
> simulatedScores

 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1
```

3

```
26 27 28 29
 1  1  1  1
```

Let's check that the names of the scores match those of the sequences:

```
> all(names(simulatedScores) == names(simulatedSeqs))
```

```
[1] TRUE
```

Looking back at the scores themselves, note that for this simple illustrative example the scores are just defined to be 0 for the first 20 sequences (`"0"`–`"19"`) and 1 for the last 10 sequences (`"20"`–`"29"`).

Note regarding SArKS input: the scores input to SArKS are not required to be positive, nor are they required to be integers. Similarly, the sequences input to SArKS do not have to use the DNA alphabet.

2.2.1   SArKS run parameters

Aside from input of data in the form of sequences with matching scores, `sarks` also requires specification of run parameters `halfWindow`, `spatialLength`, and `minGini`.

I would recommend that users try several (2-4) values of `halfWindow` and, if interested in spatial smoothing, `spatialLength`, using the method described in 2.6 below. For `halfWindow`, you might start with `halfWindow` values on the order of

$$\texttt{halfWindow} \in \left\{ \frac{n}{20}, \frac{n}{10}, \frac{n}{5} \right\} \tag{1}$$

where n is the number of input sequences.

To get a sense of the intuition behind the values in (1): If we take $\texttt{halfWindow} = \frac{n}{20}$, we get a full smoothing window of size 2*`halfWindow`+1 $\approx \frac{n}{10}$, which amounts to looking for motifs which would be expected to occur about once in every 10 input sequences.

The user should of course feel free to vary these fractions in either direction based on his or her own data and goals! Especially if the number n of input sequences is very high, so that $\frac{n}{20}$ is very large (say, in the thousands or more), these `halfWindow` values may be larger than optimal.

Another consideration to keep in mind when choosing `halfWindow` values is that there is a link between n, the average sequence length $|w|$, `halfWindow`, the size (really entropy) of the sequence alphabet and the length $\hat{k}$ of the k-mer motif sequences `sarks` is likely to detect. Assuming an approximately uniformly distributed alphabet (often *not* a valid assumption in practice) of a distinct characters:

$$\hat{k} \approx \log_a \frac{n * |w|}{\texttt{halfWindow}} \tag{2}$$

should give a very rough sense of what length the motifs returned are likely to have (though when employing spatial smoothing with merging of consecutive motifs, this will be less accurate). Equation (2) is mosly useful for understanding how changes to `halfWindow` are likely to impact output k-mer lengths, not truly predicting the exact lengths which will be seen.

Aside from the trivial value of `spatialLength=0` used to turn off spatial smoothing entirely, it is a bit more difficult to provide general guidelines for `spatialLength`. Small values (say, 3–10) can be useful to help detect individual motifs even when no larger spatial structure is really expected; in Wylie et al. [2019] we also tried `spatialLength=100` when studying regulation of gene expression as that is on the order of the low end of the length of DNA enhancer regions. Users should feel free to experiment with `spatialLength` values that make sense in their own applications.

Finally, for `minGini`, my current advice would be to start with the value `minGini=1.1` (or `minGini=0` to see what happens without this filter). Interested users should consult sections 2.4.1 and 3.2 (especially 3.2.1) as well as Wylie et al. [2019] to get a better understanding of what this parameter does in order to get a sense of how they might choose better values for their own applications.

2.3 Output

`sarks` then aims to identify short sequence motifs, and potentially longer sequence domains enriched in such motifs (MMDs), where the occurrence of the identified motifs in the input sequences is correlated with the numeric scores assigned to the sequences.

Let's consider a minimal `sarks` workflow (which we'll discuss in more detail below) to illustrate the way in which this output is constructed):

```
> sarks <- Sarks(simulatedSeqs, simulatedScores, halfWindow=4)
> filters <- sarksFilters(halfWindow=4, spatialLength=0, minGini=1.1)
> permDist <- permutationDistribution(sarks, reps=250, filters, seed=123)
> thresholds <- permutationThresholds(filters, permDist, nSigma=2.0)
> peaks <- kmerPeaks(sarks, filters, thresholds)
> peaks[ , c('i', 's', 'block', 'wi', 'kmer', 'windowed')]
       i    s block  wi       kmer windowed
1   1458 4442    24 175    ATACTGAG        1
2   1459 3545    21  31   ATACTGAGA        1
3   1460 3960    22 195   ATACTGAGA        1
4   1461 4519    25   1   ATACTGAGA        1
5   1462 3456    20 193   ATACTGAGA        1
6   1463 5595    29  73    ATACTGAG        1
7   2256 3544    21  30  CATACTGAGA        1
8   2257 3959    22 194  CATACTGAGA        1
9   2258 4518    25   0  CATACTGAGA        1
10  5862 4443    24 176     TACTGAG        1
11  5863 3546    21  32     TACTGAGA        1
12  5864 3961    22 196     TACTGAGA        1
13  5865 4520    25   2     TACTGAGA        1
```

2.3.1 k-mers

For now let's focus specifically on the column `peaks$kmer`, which tells us that SArKS has identified

```
> unique(peaks$kmer)
```

```
[1] "ATACTGAG"    "ATACTGAGA"  "CATACTGAGA" "TACTGAG"    "TACTGAGA"
```

as k-mers whose occurrence in `simulatedSeqs` is associated with higher values of `simulatedScores`. `sarks` provides a convenience function `kmerCounts`:

```
> kmerCounts(unique(peaks$kmer), simulatedSeqs)
   ATACTGAG ATACTGAGA CATACTGAGA TACTGAG TACTGAGA
0         0         0          0       0        0
1         0         0          0       0        0
2         0         0          0       0        0
3         0         0          0       0        0
4         0         0          0       0        0
5         0         0          0       0        0
6         0         0          0       0        0
7         0         0          0       0        0
8         0         0          0       0        0
9         0         0          0       0        0
10        0         0          0       0        0
11        0         0          0       0        0
12        0         0          0       0        0
13        0         0          0       0        0
14        0         0          0       0        0
15        0         0          0       0        0
16        0         0          0       0        0
17        0         0          0       0        0
18        0         0          0       0        0
19        0         0          0       0        0
20        1         1          1       1        1
21        1         1          1       1        1
22        1         1          1       1        1
23        1         1          1       1        1
24        1         1          1       1        1
25        1         1          1       1        1
26        1         1          1       1        1
27        1         1          1       1        1
28        1         1          1       1        1
29        1         1          1       1        1
```

which shows us that each of the identified k-mers appears exactly once in each of the higher-scoring sequences (`"20"`-`"29"`) and never in any of the lower-scoring sequences. Looking a bit more closely at the specific k-mers identified here, you can see that they are all substrings of the longest one, the 10-mer `"CATACTGAGA"`, so that the perfect agreement between the columns of the `kmerCounts` output above are perhaps to be expected. `sarks` provides functionality to help to identify such situations and simplify the output, as discussed further in section 2.3.2 below.

First let's go back to the output `peaks` from the call to `kmerPeaks` (focusing on the 8th row of `peaks`):

```
> peak8 = peaks[8, ]
> peak8[ , c('i', 's', 'block', 'wi', 'kmer', 'windowed')]

     i    s block  wi      kmer windowed
8 2257 3959    22 194 CATACTGAGA        1
```

and consider the columns:

`i` suffix array index

`s` suffix array value s_i

`block` the name of the input sequence from which the indicated k-mer is derived

`wi` the (0-based!) position ω_i of the k-mer within the indicated block/sequence

Considering `block` and `wi` first, this tells us that, setting

```
> block <- simulatedSeqs[[peak8$block]]
> kmerStart <- peak8$wi + 1
> kmerEnd <- kmerStart + nchar(peak8$kmer) - 1
```

(noting the `+ 1` required to define `kmerStart` in 1-based R relative to 0-based `wi`) we should find that

```
> substr(block, kmerStart, kmerEnd)
```

```
[1] "CATACTGAGA"
```

yields the same value as

```
> peak8$kmer
```

```
[1] "CATACTGAGA"
```

2.3.2 Reducing redundancy in `peaks`

In cases such as `"CATACTGAGA"` in the simulated data set analyzed here, `sarks` can simplify k-mer output using a couple of auxiliary functions:

```
> nonRedundantPeaks <- mergedKmerSubPeaks(sarks, filters, thresholds)
> nonRedundantPeaks[ , c('i', 's', 'block', 'wi', 'kmer', 'windowed')]

     i    s block  wi      kmer windowed
1 1462 3456    20 193  ATACTGAGA        1
2 2256 3544    21  30 CATACTGAGA        1
3 2257 3959    22 194 CATACTGAGA        1
4 1458 4442    24 175  ATACTGAG         1
5 2258 4518    25   0 CATACTGAGA        1
6 1463 5595    29  73  ATACTGAG         1
```

Here `mergedKmerSubPeaks` removes redundant k-mer output where multiple k-mer peaks are reported with successive spatial `s` coordinates (e.g., the reported peak `peaks[3, ]` from `block="22"` at `wi=195` and `kmer="ATACTGAGA"` immediately follows `peaks[8, ]` with `block="22"` and `wi=194` and `kmer="CATACTGAGA"`, and is thus merged with that peak).

Further simplification is still possible in this case using:

```
> extendedPeaks <- extendKmers(sarks, nonRedundantPeaks)
> extendedPeaks[ , c('i', 's', 'block', 'wi', 'kmer', 'windowed')]

     i    s block  wi       kmer windowed
1 2259 3455    20 192 CATACTGAGA       NA
2 2256 3544    21  30 CATACTGAGA        1
3 2257 3959    22 194 CATACTGAGA        1
4 2255 4441    24 174 CATACTGAGA       NA
5 2258 4518    25   0 CATACTGAGA        1
6 2260 5594    29  72 CATACTGAGA       NA
```

which detects that even though the full occurrence of `"CATACTGAGA"` in input sequence `"24"` was not reported as a k-mer by SArKS, the k-mer `"ACACTGAG"` in `nonRedundantPeaks[1, ]` is flanked in sequence `"24"` by a `"C"` to the left and an `"A"` to the right and can hence be extended to math the full motif.

2.3.3 Suffix arrays

How are we to interpret the columns `i` and `s` in `peaks`? This requires understanding a bit more about SArKS: Specifically, that the method begins by concatenating all of the input sequences into one big string (called `catSeq` in the underlying java object referenced by `sarks`):

```
> concatenated <- sarks$getCatSeq()
> nchar(concatenated)
```

```
[1] 7530
```

`concatenated` is actually a bit longer than the sum of the lengths of the input sequences because it keeps track of where one sequence ends and another begins using a special (dollar-sign) character. In this way, `concatenated` is divided into separate "blocks," each corresponding to one of the input sequences.

The column `s` in `peaks` then gives us the (0-based!) position of the annotated k-mer in the concatenated sequence:

```
> kmerCatStart <- peak8$s + 1
> kmerCatEnd <- kmerCatStart + nchar(peak8$kmer) - 1
> substr(concatenated, kmerCatStart, kmerCatEnd)
```

```
[1] "CATACTGAGA"
```

But what about `i`? As we will confirm, it is the (0-based) position of the suffix

```
> theSuffix <- substr(concatenated, kmerCatStart, nchar(concatenated))
```

in the list of all suffixes *after they have been lexicographically sorted*:

```
> extractSuffix <- function(s) {
      ## returns suffix of concatenated starting at position s (1-based)
      substr(concatenated, s, nchar(concatenated))
  }
> allSuffixes <- vapply(1:nchar(concatenated), extractSuffix, '')
> sortedSuffixes <- sort(allSuffixes)
```

Sure enough,

```
> i1based <- peak8$i + 1
> sortedSuffixes[i1based] == theSuffix
```

```
[1] TRUE
```

2.3.4   Window averaging (kernel smoothing)

Because the suffixes in `sortedSuffixes` are sorted, the suffixes in a window centered on `i1based` all start with the same few characters (a k-mer):

```
> iCenteredWindow <- (i1based - 4):(i1based + 4)
> iCenteredWindowSuffixes <- sortedSuffixes[iCenteredWindow]
> all(substr(iCenteredWindowSuffixes, 1, 10) == 'CATACTGAGA')
```

```
[1] TRUE
```

For each suffix in `sortedSuffixes`, we can identify which input sequence contributed the block of `concatenated` where the suffix begins:

```
> iCenteredWindow0Based <- iCenteredWindow - 1
> sourceBlock(sarks, i=iCenteredWindow0Based)
```

```
[1] "26" "28" "24" "21" "22" "25" "20" "29" "27"
```

Note that the 9 suffixes in `iCenteredWindowSuffixes` derive from come from 9 of the 10 higher-scoring sequences (all but `"23"`). This is not an accident: since the motif `"CATACT-GAGA"` is only present in high-scoring sequences, the suffixes starting with `"CATACTGAGA"` must derive from blocks associated with high-scoring sequences. Thus the *average* score of the sequences contributing the suffixes specified by `iCenteredWindowSuffixes` must be high (value of 1) as well.

The SArKS algorithm turns this around to identify motifs by hunting for windows in the sorted suffix list where the average score of the corresponding `sourceBlock` sequences is high.

The `windowed` column of the `peaks` output contains the average `sourceBlock` sequence scores for the windows centered around each sorted suffix position `i` and extending to both the left and right by `halfWindow=4` positions. These average values are also referred to as $\hat{y}_i$ in Wylie et al. [2019]; a vector containing all of these values can be obtained from the object `sarks`:

```
> yhat <- sarks$getYhat()
> i0based <- seq(0, length(yhat)-1)
> plot(i0based, yhat, type='l', xlab='i')
```
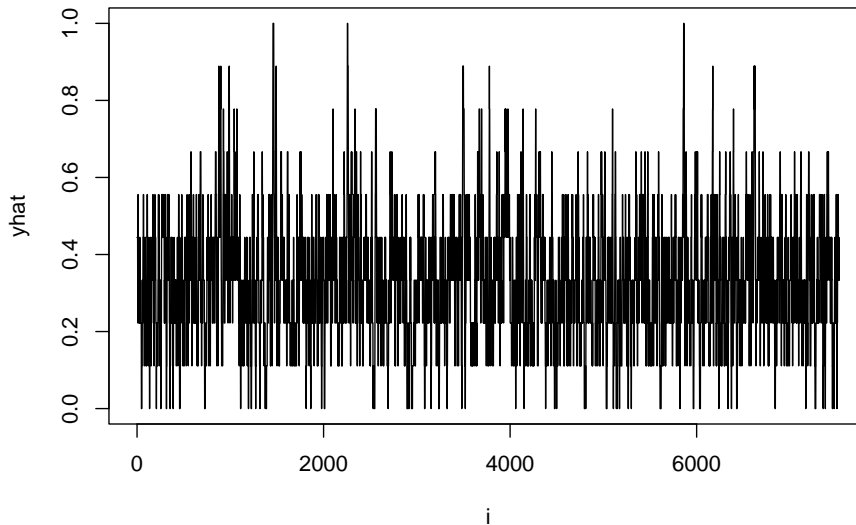
From the plot of $\hat{y}_i$ against i, we can see three peaks at which $\hat{y}_i$ spikes up to 1, corresponding to the ranges $i \in [1458, 1463]$, $i \in [2256, 2258]$, and $i \in [5862, 5865]$ which make up the values in `peaks$i`.

2.4 Permutations and thresholds

How do we know that a given value of $\hat{y}_i$ (or, equivalently, `peaks$windowed`) is high enough to include in the `peaks` output? In order to set such a threshold without having to make possibly unwarranted assumptions about the structure of the input sequences or the distribution of the scores, SArKS employs a permutational approach.

To illustrate the idea here, let's randomly permute the scores—so that the permuted scores have no relationship with which sequences contain `"CATACTGAGA"`—and construct a new `Sarks` object using the permuted scores:

```
> set.seed(12345)
> scoresPerm <- sample(simulatedScores)
> names(scoresPerm) <- names(simulatedScores)
> sarksPerm <- Sarks(simulatedSeqs, scoresPerm, halfWindow=4)
```

If we try using the same procedure we did before to look for k-mer peaks:

```
> permDistNull <- permutationDistribution(
          sarksPerm, reps=250, filters, seed=123)
> thresholdsNull <- permutationThresholds(
          filters, permDistNull, nSigma=2.0)
> peaksNull <- kmerPeaks(sarksPerm, filters, thresholdsNull)
> peaksNull[ , c('i', 's', 'block', 'wi', 'kmer', 'windowed')]

[1] i         s         block    wi        kmer       windowed
<0 rows> (or 0-length row.names)
```

10

we find that SArKS does not detect anything worthy of reporting after we disrupt any association between input sequences and input scores by permuting the scores.

This is to be expected given that the thresholds θ used by SArKS are defined by considering many (here, `reps=250`) such permutations and choosing θ such that only very few random permutations would produce smoothed $\hat{y}_i$ scores exceeding θ. The adjustable parameter `nSigma` controls how stringent the thresholds are: higher `nSigma` values lead to higher thresholds, reducing sensitivity but also reducing the false positive rate.

Once we've set thresholds using `permutationThresholds`, we can estimate the false positive rate, defined here as the frequency of seeing nonempty k-mer result sets when there is really no association between sequence and score:

```
> fpr = estimateFalsePositiveRate(
        sarks, reps=250, filters, thresholds, seed=321)
> fpr$ci

  method x   n mean      lower      upper
1  exact 5 250 0.02 0.00652507 0.04605358
```

indicating a point estimate of 2% for the false positive rate, with a 95% confidence interval of (0.65%, 4.6%). This confidence interval can be made tighter by using a higher number for `reps` in the `estimateFalsePositiveRate` call.

Note regarding random number generator seed for `estimateFalsePositiveRate`: do not use the same seed for `estimateFalsePositiveRate` as was used in `permutationDistribution` call used to set thresholds. The false positive rate estimation procedure assumes that the random permutations used in `estimateFalsePositiveRate` are independent of those used to set thresholds.

2.4.1 Gini impurity filter

When considering how to set SArKS thresholds in order to obtain a reasonable tradeoff between sensitivity and false positive rate, the mysterious `minGini` parameter should be factored in as well. This parameter is discussed in more detail in sections 3.1.7 and 3.2.1, but the basic idea is to filter likely false positive suffix array index positions i out of consideration regardless of their smoothed scores $\hat{y}_i$. These likely false positive indices i come from excessive repetition of the same input sequences contributing repeatedly to the smoothing windows centered on i.

My recommendation is to set `minGini` to a value slightly greater than 1: the value 1.1 seems empirically to work well in many situations. Note that for `minGini` > 1, this filter becomes more stringent the closer to 1 it is set; the opposite is true if you set `minGini` to a value less than 1, and you can turn the filter off entirely by setting it to 0. See section 3.2.1 and Wylie et al. [2019] for more details.

Here we examine the effects of this parameter using the simulated data:

```
> estimateFalsePositiveRate(
        sarks, reps=250, filters, thresholds, seed=321)$ci

  method x   n mean      lower      upper
1  exact 5 250 0.02 0.00652507 0.04605358
```

```
> filtersNoGini <- sarksFilters(halfWindow=4, spatialLength=0, minGini=0)
> estimateFalsePositiveRate(
          sarks, reps=250, filtersNoGini, thresholds, seed=321)$ci

  method  x   n  mean     lower     upper
1  exact 42 250 0.168 0.1238429 0.2202254
```

Of course we could compute a new set of `thresholdsNoGini` using `filtersNoGini`, but the results of this on our ability to detect anything are worth considering:

```
> permDistNoGini <-
          permutationDistribution(sarks, reps=250, filtersNoGini, seed=123)
> thresholdsNoGini <-
          permutationThresholds(filtersNoGini, permDistNoGini, nSigma=2.0)
> peaksNoGini <- kmerPeaks(sarks, filtersNoGini, thresholdsNoGini)
> peaksNoGini[ , c('i', 's', 'block', 'wi', 'kmer', 'windowed')]

[1] i         s         block     wi        kmer      windowed
<0 rows> (or 0-length row.names)
```

2.4.2   Multithreading `sarks` permutational analyses

Setting thresholds and estimating false positive rates using permutation methods is usually the most time-consuming step in the SArKS workflow. To facilitate more rapid turnaround, the java back end of `sarks` supports multithreading for these processes. In order to take advantage of multithreading, you just need to specify how many threads you'd like to use when you invoke the `Sarks` constructor using the `nThreads` argument to that function; all permutation steps performed using the resulting `sarks` object will then use the specified number of threads.

Note regarding `sarks` multithreading: while the different threads performing the permutational analyses share as many data structures as possible to reduce the memory requirements of `sarks`, each thread will need to keep track of its own permuted smoothed scores (and spatially smoothed scores if necessary). This can increase memory requirements quickly, so make sure you are running `sarks` on a machine with large RAM if you are planning to use many threads on a large data set.

2.5   Spatial smoothing

So far we have not taken advantage of the spatial smoothing features in SArKS. While one of the major uses of spatial smoothing is to detect multi-motif domains (MMDs), which are not present in the simple simulated data set included in the `sarks` package, spatial smoothing can also help sharpen the ability of SArKS to detect individual motifs:

```
> sarks <- Sarks(
      simulatedSeqs, simulatedScores, halfWindow=4, spatialLength=3
  )
> filters <- sarksFilters(halfWindow=4, spatialLength=3, minGini=1.1)
> permDist <- permutationDistribution(sarks, reps=250, filters, seed=123)
> thresholds <- permutationThresholds(filters, permDist, nSigma=5.0)
> ## note setting of nSigma to higher value of 5.0 here!
```

```
> peaks <- kmerPeaks(sarks, filters, thresholds)
> peaks[ , c('i', 's', 'block', 'wi', 'kmer', 'spatialWindowed')]

     i    s block  wi       kmer spatialWindowed
1 1458 4442    24 175   ATACTGAG       0.9259259
2 1459 3545    21  31   ATACTGAGA      0.9629630
3 1460 3960    22 195   ATACTGAGA      0.9259259
4 1461 4519    25   1   ATACTGAGA      0.9259259
5 2256 3544    21  30 CATACTGAGA       1.0000000
6 2257 3959    22 194 CATACTGAGA       1.0000000
7 2258 4518    25   0 CATACTGAGA       1.0000000
8 5863 3546    21  32   TACTGAGA       0.9259259
```

Note that here we use `nSigma=5.0`, a much more stringent setting than the `nSigma=2.0` value used when no spatial smoothing was employed; had we tried `nSigma=5.0` without spatial smoothing, SArKS would not have been able to detect the `"CATACTGAGA"` motif.

The `peaks` object returned by `kmerPeaks` when spatial smoothing is employed contains the `i` and `s` coordinates for the left endpoints of spatial windows (windows in `s`-space, not `i`-space) enriched in high $\hat{y}$ scores. Especially when these windows have longer `spatialLength` values, it can also be useful to pick out individual motif "subpeaks" within these spatial windows:

```
> subpeaks <- mergedKmerSubPeaks(sarks, filters, thresholds)
> subpeaks[ , c('i', 's', 'block', 'wi', 'kmer')]

     i    s block  wi       kmer
1 2256 3544    21  30 CATACTGAGA
2 2257 3959    22 194 CATACTGAGA
3 1458 4442    24 175   ATACTGAG
4 2258 4518    25   0 CATACTGAGA
```

In this case, with `spatialLength=3`, this is not such an important step—although it does serve to accomplish some simplification of results as was described when `mergedKmerSub-Peaks` was first introduced in section 2.3.2—but in general it is a critical piece of the SArKS methodology when spatial smoothing is in use.

The `subpeaks` output of `mergedKmerSubPeaks` should generally be regarded as the main individual motif output for SArKS. Section 3.5 provides more details.

When employing spatial smoothing to identify MMDs, you may want to inspect individual input sequences of interest by visualizing the SArKS results:

```
> block22 <- blockInfo(sarks, block='22', filters, thresholds)
> library(ggplot2)
> ggo <- ggplot(block22, aes(x=wi+1))  ## +1 because R indexing is 1-based
> ggo <- ggo + geom_point(aes(y=windowed), alpha=0.6, size=1)
> ggo <- ggo + geom_line(aes(y=spatialWindowed), alpha=0.6)
> ggo <- ggo + geom_hline(aes(yintercept=spatialTheta), color='red')
> ggo <- ggo + ylab('yhat') + ggtitle('Input Sequence "22"')
> ggo <- ggo + theme_bw()
> print(ggo)
```

Input Sequence "22"
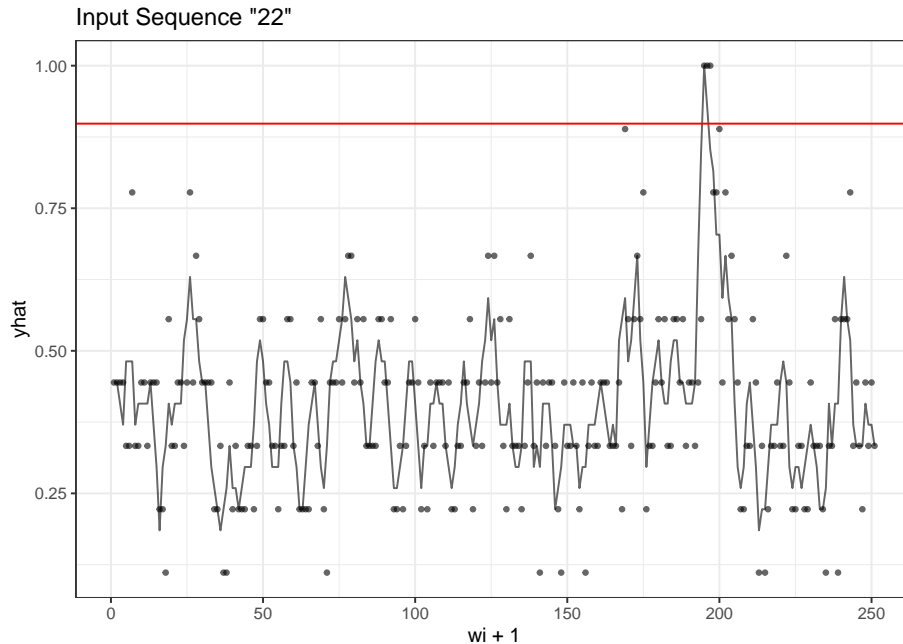
which here clearly shows the spike at the location `wi+1=195` of the `"CATACTGAGA"` motif in sequence `"22"`.

The use of the more stringent `nSigma=5.0` setting reduces the false positive rate relative to the `nSigma=2.0` setting we used when not employing spatial smoothing:

```
> estimateFalsePositiveRate(
        sarks, reps=250, filters, thresholds, seed=321)$ci

  method x    n mean lower      upper
1  exact 0 250    0     0 0.01464719
```

2.6 Varying SArKS parameters

You may be wondering what the point of the `filters` object created in the SArKS workflow is, as it seems to specify the `halfWindow` and `spatialLength` parameters in a manner redundant to their specification in the `Sarks` constructor. Aside from the specification of the `minGini` parameter, the motivation for this step is that we usually don't know exactly what `halfWindow` or `spatialLength` values will yield the most useful output (and the answer to these questions will be different for different data sets and different questions).

To address this, we can use `sarksFilters` to test a variety of combinations of `halfWindow` and `spatialLength` (and, if so desired, `minGini` as well) values like so:

```
> filters <- sarksFilters(
        halfWindow=c(4, 8), spatialLength=c(2, 3), minGini=1.1)
> permDist <- permutationDistribution(sarks, reps=250, filters, seed=123)
> thresholds <- permutationThresholds(filters, permDist, nSigma=5.0)
> peaks <- mergedKmerSubPeaks(sarks, filters, thresholds)
> peaks[ , c('halfWindow', 'spatialLength', 'i', 's', 'block', 'wi', 'kmer')]
```

14

```
   halfWindow spatialLength    i    s block   wi      kmer
1           4              2 2256 3544    21   30 CATACTGAGA
2           4              2 2257 3959    22  194 CATACTGAGA
3           4              2 1458 4442    24  175   ATACTGAG
4           4              2 2258 4518    25    0 CATACTGAGA
5           4              3 2256 3544    21   30 CATACTGAGA
6           4              3 2257 3959    22  194 CATACTGAGA
7           4              3 1458 4442    24  175   ATACTGAG
8           4              3 2258 4518    25    0 CATACTGAGA
9           8              2 1459 3545    21   31    ATACTGA
10          8              2 2261 5173    27  153   CATACTGA
11          8              3 1459 3545    21   31    ATACTGAG
12          8              3 3498 4445    24  178      CTGAGA
13          8              3 5860 4947    26  178      TACTGA
14          8              3 2261 5173    27  153   CATACTGA
15          8              3 3497 5432    28  161       CTGAG
```

Note that, as is generally true with SArKS, the results obtained using larger `halfWindow` values tend to have shorter identified k-mers.

As always, testing multiple parameter sets can increase false positive rates:

```
> estimateFalsePositiveRate(
          sarks, reps=250, filters, thresholds, seed=321)$ci
```

```
  method x   n mean lower      upper
1  exact 0 250    0     0 0.01464719
```

As suggested in section 2.5 above, this can be countered by using increased `nSigma` values when setting thresholds if we plan to test a wider range of possible SArKS parameters.

Note regarding multiple hypothesis testing in SArKS: The false positive rates estimated by SArKS test the rate of detecting any results using any of the specified parameter settings, and are thus a type of family-wise error rate. As long as all parameter combinations tested are included in the `filters` employed in `estimateFalsePositiveRate`, no further adjustment for multiple testing should be applied to the estimated false positive rates.

2.7   Clustering similar k-mers into broader motifs

While there is essentially one unambiguous k-mer `"CATACTGAGA"` of note in the `simulated-Seqs`-`simulatedScores` example, real data sets will generally have more complex sequence motifs allowing for some variation both in length and composition of the patterns. When applying SArKS methodology to real data, many similar k-mers representing the same basic motif may result: `sarks` provides functions for clustering these k-mers into broader motif patterns.

For example, let's say we had obtained the following k-mers using `sarks`:

```
> kmers <- c(
      'CAGCCTGG', 'CCTGGAA', 'CAGCCTG', 'CCTGGAAC', 'CTGGAACT',
      'ACCTGC', 'CACCTGC', 'TGGCCTG', 'CACCTG', 'TCCAGC',
```

```
        'CTGGAAC', 'CACCTGG', 'CTGGTCTA', 'GTCCTG', 'CTGGAAG', 'TTCCAGC'
    )
```

We could cluster these like so:

```
> kmClust <- clusterKmers(kmers, directional=FALSE)
> ## directional=FALSE indicates that we want to treat each kmer
> ##                   as equivalent to its reverse-complement
> kmClust

$CAGCCTGG
[1] "CAGCCTGG" "CAGCCTG"


$CTGGAAC
[1] "CCTGGAA"  "CCTGGAAC" "CTGGAACT" "TCCAGC"    "CTGGAAC"  "CTGGAAG"  "TTCCAGC"


$CACCTGC
[1] "ACCTGC"   "CACCTGC" "CACCTG"   "CACCTGG"


$TGGCCTG
[1] "TGGCCTG"


$CTGGTCTA
[1] "CTGGTCTA"


$GTCCTG
[1] "GTCCTG"
```

The resulting object `kmClust` is a named list: each element of this list is a character vector listing the elements of the vector `kmers` composing the corresponding cluster, while the name of the cluster is a k-mer from `kmers` found to be particularly representative of the cluster.

`sarks` then allows us to count how many times each motif (=cluster of k-mers) occurs in each sequence:

```
> clCounts <- clusterCounts(kmClust, simulatedSeqs, directional=FALSE)
> ## directional=FALSE to count hits of either a kmer from the cluster
> ##                   or the reverse-complement of such a kmer
> ## clCounts is a matrix with:
> ## - one row for each sequence in simulatedSeqs
> ## - one column for each *cluster* in kmClust
> head(clCounts)

  CAGCCTGG CTGGAAC CACCTGC TGGCCTG CTGGTCTA GTCCTG
0        0       0       0       0        0      0
1        0       0       1       0        0      0
2        0       0       0       0        0      0
3        0       0       0       0        0      0
4        0       0       0       0        0      0
5        0       0       0       1        0      1
```

Can also get specific information about the location of these matches:

```
> clLoci <- locateClusters(
      kmClust, simulatedSeqs, directional=FALSE, showMatch=TRUE
  )
> ## showMatch=TRUE includes column specifying exactly what k-mer
> ##                  registered as a hit;
> ##                  this can be very slow, so default is showMatch=FALSE
> clLoci
```

```
  seqid  cluster location    match
1     25 CAGCCTGG      143 CAGCCTGG
2      6  CTGGAAC      190  GCTGGAA
3     16  CTGGAAC      196 AGTTCCAG
4     28  CTGGAAC       36  GTTCCAG
5      1  CACCTGC      242  GCAGGTG
6     28  CACCTGC       25   GCAGGT
7      5  TGGCCTG      216  TGGCCTG
8     19 CTGGTCTA       51 TAGACCAG
9      5   GTCCTG       19   GTCCTG
10     9   GTCCTG       67   GTCCTG
11    13   GTCCTG       86   CAGGAC
12    15   GTCCTG       19   CAGGAC
13    17   GTCCTG      151   CAGGAC
14    19   GTCCTG       76   CAGGAC
15    24   GTCCTG       38   GTCCTG
```

This shows us that, for example, there is one match for the cluster `"CAGCCTGG"` spanning sequence characters 143-150 of sequence `"25"`, and that this is an exact match of the k-mer `"CAGCCTGG"` in its forward orientation.

These results also tells us that there are three matches of k-mers from the cluster `"CTGGAAC"`, in sequences `"6"`, `"16"`, and `"28"`. The specific k-mers found are different in each case:

- sequence `"6"` has a hit for the reverse-complement of k-mer `"TTCCAGC"`, while

- sequence `"16"` has a hit for k-mer `"CTGGAACT"` in reverse-complement orientation and

- sequence `"28"` has a hit for `"CTGGAAC"` also in reverse-complement orientation.

3 How `sarks` works

3.1 Sarks constructor

```
> sarks = Sarks(
      simulatedSeqs, simulatedScores, halfWindow=4, spatialLength=3, nThreads=4
  )
```

The object `sarks` created by the `Sarks` constructor is an R representation of a java object which contains several attributes worth noting: these are described in the next few subsections, the headings of which match the names of the corresponding java Sarks class attributes.

3.1.1 `catSeq`

The concatenated sequence $x = w_0 * w_1 * \cdots * w_{n-1}$, which may be extracted using the R code `sarks$getCatSeq()`.

3.1.2 `sa`

The suffix array s_i mapping sorted suffix index i to spatial position s_i in the concatenated sequence x. Calculated using the java code internally implemented in `SkewSuffixArray` class implementing the skew algorithm initially described in Kärkkäinen and Sanders [2003]. The suffix array may be extracted via the R code `sarks$getSuffixArray()`.

3.1.3 `saInv`

The inverted suffix array i_s mapping spatial position s of the suffix formed by deleting the first s characters of the concatenated sequence x to the position i of this suffix in the sorted list of all suffixes of x. The inverted suffix array may be extracted via the R code `sarks$getInvertedSuffixArray()`.

3.1.4 `scores`

The array of sequence (block) scores y_b. May be extracted via R code `blockScores(sarks)`.

3.1.5 `windowed`

The kernel- (or window-)smoothed scores $\hat{y}_i$ defined by

$$\hat{y}_i = \frac{1}{2\kappa + 1} \sum_{j=i-\kappa}^{i+\kappa} y_{b_j} \tag{3}$$

where:

κ is the specified `halfWindow` value and

b_j is the input sequence block in which suffix with suffix array index j begins.

In R, the value of b_j for any suffix array index j can be assessed using the code `sourceBlock(sarks, i=j)`.

The array of kernel-smoothed scores $\hat{y}_i$ can be extracted using the R code `sarks$getYhat()`.

3.1.6 `spatialWindowed`

The spatially-smoothed scores $\hat{\hat{y}}_i$ (here indexed by suffix array index i, not spatial position s_i) defined by

$$\hat{\hat{y}}_i = \frac{1}{\lambda} \sum_{s=s_i}^{s_i+\lambda-1} \hat{y}_{i_s} \tag{4}$$

where:

λ is the specified `spatialLength` value and

i_s is the sorted suffix index associated with the suffix starting at spatial position s (obtained from the inverted suffix array described in section 3.1.3 above).

The array of spatially-smoothed scores $\hat{\hat{y}}_i$ can be extracted using the R code `sarks$getYdoubleHat()`.

3.1.7 `windGini`

Array whose i^{th} value is the Gini impurity value g_i of smoothing window centered on suffix array index i, defined by

$$g_i = \sum_b f_b^{(i)} \left(1 - f_b^{(i)}\right) \tag{5}$$

where:

$f_b^{(i)} = \frac{1}{2\kappa+1} \sum_{j=i-\kappa}^{j+\kappa} \delta_{b_j b}$ is the fraction of positions within the smoothing window associated
with sequence (or block) b (note $\delta_{b_j b}$ is Kronecker delta taking value 1 if $b_j = b$ and 0 otherwise).

Low values of the Gini impurity values g_i are used by SArKS to filter out likely false positive suffix array positions i: First note that equation (3) may be rewritten

$$\hat{y}_i = \frac{1}{2\kappa+1} \sum_b f_b^{(i)} y_b \tag{6}$$

Now:

- shuffling the sequence/block scores y_b by a random permutation Π,

- recomputing the resulting smoothed scores $\hat{y}_i^{(\Pi)}$ using equation (6),

- noting that by symmetry $\mathbb{V}\left[y_{\Pi(b)}\right] = \mathbb{V}\left[y_{\Pi(b')}\right]$ for all sequences b, b' under random permutation Π and

- neglecting the small interdepence between $y_{\Pi(b)}$ and $y_{\Pi(b')}$ for distinct sequences b and b',

we can approximate

$$\mathbb{V}\left[\hat{y}_i^{(\Pi)}\right] \propto \left[f_b^{(i)}\right]^2 = 1 - g_i \tag{7}$$

Equation (7) tells us that, even under the null hypothesis of no association between sequence w_b and sequence score y_b, at positions i for which the Gini impurity g_i is particularly far below the maximum value of 1, the smoothed scores will tend to be more extreme (high or low) than at other positions. Hence a high score $\hat{y}_i$ for such a position is less informative than would be the same score at a different position j.

The array of Gini impurity values g_i can be extracted using the R code `sarks$getGini()`.

3.1.8 `spatGini`

Array whose i^{th} value is the spatially averaged Gini impurity value $\bar{g}_i$ of smoothing window centered on suffix array index i, defined by

$$\bar{g}_i = \frac{1}{\lambda} \sum_{s=s_i}^{s_i+\lambda-1} g_{i_s} \tag{8}$$

The spatially averaged Gini impurities are again used to filter out likely false-positive positions i, this time when spatial smoothing is employed to calculate $\hat{\hat{y}}_i$ values. The idea behind this filter is that, now neglecting the interdependence between $\hat{y}_i^{(\text{II})}$ and $\hat{y}_j^{(\text{II})}$ for distinct suffix array indices i and j, we can get a crude estimate of the variance

$$\mathbb{V}\left[\hat{\hat{y}}_i^{(\text{II})}\right] = \mathbb{V}\left[\frac{1}{\lambda}\sum_{s=s_i}^{s_i+\lambda-1}\hat{y}_{i_s}^{(\text{II})}\right] \tag{9}$$

from

$$\frac{1}{\lambda^2}\sum_{s=s_i}^{s_i+\lambda-1}\mathbb{V}\left[\hat{y}_{i_s}^{(\text{II})}\right] \propto 1-\bar{g}_i \tag{10}$$

The degree of approximation involved in the independence assumption required to pass from equation (9) to equation (10) depends on how dissimilar the smoothing window compositions $f_b^{(i)}$ are for the various suffix array indices i appearing in equation (9). For the sake of simplicity and tractability, the approach taken in the current implementation of SArKS is to employ the spatial Gini filter as a heuristic parameter whose utility is to be assessed empirically via permutation testing.

3.2 `permutationDistribution`

Once the object `sarks` has been constructed using the `Sarks` constructor function, we can select a range of `halfWindow`, `spatialLength`, and `minGini` parameter values for motif and MMD selection. The R function `sarksFilters` puts all combinations of values of these values (specificied as numeric vectors in R) into a java object to pass along to downstream SArKS functions including `permutationDistribution`:

```
> filters <- sarksFilters(halfWindow=4, spatialLength=c(0, 3), minGini=1.1)
> permDist <- permutationDistribution(
          sarks, reps=250, filters=filters, seed=123)
```

Let $\left(\kappa^{(\alpha)}, \lambda^{(\alpha)}, g_{\min}^{(\alpha)}\right)$ be the resulting SArKS `halfWindow`, `spatialLength`, and `minGini` parameters, with α here indexing the set of desired combinations. The `permutationDistribution` call generates `reps=250` random permutations π_r:

- for each of these permutations π_r and

- for each combination of parameters $\left(\kappa^{(\alpha)}, \lambda^{(\alpha)}, g_{\min}^{(\alpha)}\right)$,

it then computes

- the smoothed scores $\hat{y}_i^{(\alpha,\pi_r)}$ and,

- if applicable, the spatially-smoothed scores $\hat{\hat{y}}_i^{(\alpha,\pi_r)}$.

The resulting `permDist` object is a named list in R. The first two elements, 'windowed' and 'spatial' are data.frames with `reps=250` rows per combination of parameters in `filters`. Both of these have a column named 'max' containing either

- `permDists$windowed$max`:

$$\hat{y}_{\max}^{(\alpha,\pi_r)} = \max_{\{i \mid g_i \geq g_{\min}^{(\alpha)}\}} \hat{y}_i^{(\alpha,\pi_r)} \tag{11}$$

- `permDists$spatial$max`:

$$\hat{y}_{\max}^{(\alpha,\pi_r)} = \max_{\{i \,|\, \bar{g}_i \geq g_{\min}^{(\alpha)}\}} \hat{y}_i^{(\alpha,\pi_r)} \tag{12}$$

3.2.1 Specification of $g_{\min}^{(\alpha)}$ in `sarks`

SArKS currently allows the user to directly set $g_{\min}^{(\alpha)} < 1$ excluding suffix array index values i with $g_i^{(\alpha)} < g_{\min}^{(\alpha)}$ from permutational analysis and $\hat{y}_i$ peak calling if so desired. Given the interpretation afforded by equation (7), however, it can be more convenient to indirectly specify $g_{\min}^{(\alpha)}$ by choosing γ in the following:

$$1 - g_{\min}^{(\alpha)} = (1 + \gamma)\left(1 - \operatorname*{median}_i g_i^{(\alpha)}\right) \tag{13}$$

Together equations (7) and (13) imply that fixing γ restricts analysis to suffix indices i for which the variance of the permuted smoothed scores is at most $(1 + \gamma)$ times the median such variance.

If `minGini` is set to a value ≥ 1, each of the $g_{\min}^{(\alpha)}$ values will be set using equation (13) with $\gamma = $ `minGini` -1.

3.3 `permutationThresholds`

```
> thresholds = permutationThresholds(
          filters=filters, permDist=permDist, nSigma=5.0)
```

SArKS uses a simple method for setting thresholds $\theta^{(\alpha)}$ or $\theta_{\text{spatial}}^{(\alpha)}$ based on the set of randomly generated permutations $\{\pi_r\}$:

$$\theta^{(\alpha)} = \operatorname*{mean}_r\left\{\hat{y}_{\max}^{(\alpha,\pi_r)}\right\} + z\operatorname*{stdev}_r\left\{\hat{y}_{\max}^{(\alpha,\pi_r)}\right\} \tag{14}$$

$$\theta_{\text{spatial}}^{(\alpha)} = \operatorname*{mean}_r\left\{\hat{\bar{y}}_{\max}^{(\alpha,\pi_r)}\right\} + z\operatorname*{stdev}_r\left\{\hat{\bar{y}}_{\max}^{(\alpha,\pi_r)}\right\} \tag{15}$$

The quantity z appearing in equations (14)-(15) is specified by the `nSigma` argument to `permutationThresholds`. Higher values of z trade reduced sensitivity for lower false positive rates.

As currently implemented, equation (14) is only applied for parameter sets α for which no spatial smoothing is done (encoded as `spatialLength` or $\lambda^{(\alpha)} = 0$). For those parameter sets α such that $\lambda^{(\alpha)} > 1$, equation (15) is instead applied to obtain $\theta_{\text{spatial}}^{(\alpha)}$, with $\theta^{(\alpha)} = -\infty$.

3.4 `kmerPeaks`

```
> peaks = kmerPeaks(sarks, filters=filters, thresholds=thresholds)
```

If the option `peakify` is turned on (set to `TRUE`, as it is by default), the set of suffix array indices $I^{(\alpha)}$ or $J_{\text{spatial}}^{(\alpha)}$—depending on whether spatial smoothing is employed or not—defining

the SArKS peak set for parameter combination α is determined by:

$$I^{(\alpha)} = \left\{ i \;\middle|\; \left(\hat{y}_i^{(\alpha)} \geq \theta^{(\alpha)}\right) \wedge \left(\hat{y}_{\eta(i)}^{(\alpha)} \leq \hat{y}_i^{(\alpha)} \geq \hat{y}_{\rho(i)}^{(\alpha)}\right) \wedge \left(g_i^{(\alpha)} \geq g_{\min}^{(\alpha)}\right) \right\} \tag{16}$$

$$J_{\text{spatial}}^{(\alpha)} = \left\{ i \;\middle|\; \left(\hat{\bar{y}}_i^{(\alpha)} \geq \theta_{\text{spatial}}^{(\alpha)}\right) \wedge \left(\hat{\bar{y}}_{\eta(i)}^{(\alpha)} \leq \hat{\bar{y}}_i^{(\alpha)} \geq \hat{\bar{y}}_{\rho(i)}^{(\alpha)}\right) \wedge \left(\bar{g}_i^{(\alpha)} \geq g_{\min}^{(\alpha)}\right) \right\} \tag{17}$$

where:

$\eta(i)$ is the negative spatial shift operator defined by $\eta(i) = i_{s_i-1}$, and

$\rho(i)$ is the positive spatial shift operator defined by $\rho(i) = i_{s_i+1}$.

The condition that $\hat{y}_{\eta(i)}^{(\alpha)} \leq \hat{y}_i^{(\alpha)} \geq \hat{y}_{\rho(i)}^{(\alpha)}$ (or similar for $\hat{\bar{y}}_{\eta(i)}^{(\alpha)}$) restricts the peak set to only those suffix array indices i for which there is not a higher smoothed score immediately spatially adjacent in either direction.

If `peakify` is disabled, this condition is not required, so that instead

$$I^{(\alpha)} = \left\{ i \;\middle|\; \left(\hat{y}_i^{(\alpha)} \geq \theta^{(\alpha)}\right) \wedge \left(g_i^{(\alpha)} \geq g_{\min}^{(\alpha)}\right) \right\} \tag{18}$$

$$J_{\text{spatial}}^{(\alpha)} = \left\{ i \;\middle|\; \left(\hat{\bar{y}}_i^{(\alpha)} \geq \theta_{\text{spatial}}^{(\alpha)}\right) \wedge \left(\bar{g}_i^{(\alpha)} \geq g_{\min}^{(\alpha)}\right) \right\} \tag{19}$$

is used to define $I^{(\alpha)}$ or $J_{\text{spatial}}^{(\alpha)}$.

If no spatial smoothing is employed, we can use $\hat{k}_i^{(\alpha)}$ defined by

$$\hat{k}_i^{(\alpha)} = \frac{1}{2\kappa^{(\alpha)}} \sum_{j=i-\kappa^{(\alpha)}}^{i+\kappa^{(\alpha)}} (1 - \delta_{ij}) \max \left\{ k \leq k_{\max} \mid x[s_j, s_j + k] = x[s_i, s_i + k] \right\} \tag{20}$$

to estimate the characteristic length $\lfloor \hat{k}_i^{(\alpha)} \rceil$ (where $\lfloor \cdots \rceil$ indicates nearest integer) of the k-mer $x[s_i, s_i + \lfloor \hat{k}_i^{(\alpha)} \rceil]$) associated with the smoothing window centered on the suffix with sorted index i. We can then identify the set of k-mers reported by SArKS using parameter set α when not using spatial smoothing (i.e., when `spatialLength` $= \lambda^{(\alpha)} = 0$) by:

$$M^{(\alpha)} = \left\{ x[s_i, s_i + \lfloor \hat{k}_i^{(\alpha)} \rceil]) \mid i \in I^{(\alpha)} \right\} \tag{21}$$

3.5 `mergedKmerSubPeaks`

`> mergedPeaks = mergedKmerSubPeaks(sarks, filters, thresholds)`

The set of suffix array indices marking the left ends of merged k-mer subpeaks when spatial smoothing is employed is given by:

$$I_{\text{spatial}}^{(\alpha)} = \left\{ i \;\middle|\; \left(\delta_i^{(\alpha)} < \lambda^{(\alpha)}\right) \wedge \left(\hat{\bar{y}}_i^{(\alpha)} \geq \theta_{\text{spatial}}^{(\alpha)}\right) \wedge \left[\left(\delta_{\eta(i)}^{(\alpha)} \geq \lambda^{(\alpha)}\right) \vee \left(\hat{\bar{y}}_{\eta(i)}^{(\alpha)} < \theta_{\text{spatial}}^{(\alpha)}\right) \right] \right\} \tag{22}$$

where

$$\delta_j^{(\alpha)} = \min_i \left\{ s_j - s_i \;\middle|\; \left(i \in J_{\text{spatial}}^{(\alpha)}\right) \wedge (s_i \leq s_j) \right\} \tag{23}$$

is the distance from spatial position s_j to the nearest element of $J_{\text{spatial}}^{(\alpha)}$ spatially left of s_j, and

$\delta_i^{(\alpha)} < \lambda^{(\alpha)}$: requires the suffix i to be spatially located within one of the identified MMDs,

$\hat{y}_i^{(\alpha)} \geq \theta_{\mathbf{spatial}}^{(\alpha)}$: requires the singly-smoothed score $\hat{y}_i^{(\alpha)}$ to be above the threshold $\theta_{\mathbf{spatial}}^{(\alpha)}$

and finally, we require either that:

$\delta_{\eta(i)}^{(\alpha)} \geq \lambda^{(\alpha)}$: s_i is at the beginning of MMD, or

$\hat{y}_{\eta(i)}^{(\alpha)} < \theta_{\mathbf{spatial}}^{(\alpha)}$: score of suffix immediately spatially prior to s_i is below threshold $\theta_{\mathbf{spatial}}^{(\alpha)}$—
idea here is that if this condition is not met we want to merge suffix at s_i into suffix initiated spatially to left.

Once we have defined $I_{\text{spatial}}^{(\alpha)}$, SArKS estimates the associated k-mer lengths for merged spatially smoothed suffix array index peaks $i \in I_{\text{spatial}}^{(\alpha)}$ by

$$\hat{\hat{k}}_i^{(\alpha)} = \max_j \left\{ \hat{k}_j^{(\alpha)} + s_j - s_i \ \middle| \ s \in [s_i, s_j] \implies \hat{y}_{i_s}^{(\alpha)} \geq \theta_{\text{spatial}}^{(\alpha)} \right\} \tag{24}$$

where $\hat{k}_j^{(\alpha)}$ is defined by equation (20) and the condition $s \in [s_i, s_j] \implies \hat{y}_{i_s}^{(\alpha)} \geq \theta_{\text{spatial}}^{(\alpha)}$ requires that all spatial positions s between s_i and s_j (including both s_i and s_j) have smoothed scores $\hat{y}_{i_s}$ exceeding $\theta_{\text{spatial}}^{(\alpha)}$ (and should hence be merged together). The resulting motif set is then defined by:

$$M_{\text{spatial}}^{(\alpha)} = \left\{ x\left[s_i, \ s_i + \left\lfloor \hat{\hat{k}}_i^{(\alpha)} \right\rfloor\right) \ \middle| \ i \in I_{\text{spatial}}^{(\alpha)} \right\} \tag{25}$$

3.6 `estimateFalsePositiveRate`

```
> fpr = estimateFalsePositiveRate(sarks, reps=250,
                                   filters=filters, thresholds=thresholds,
                                   seed=123456)
```

The false positive rate associated with a given set of parameters $\left\{ \left(\kappa^{(\alpha)}, \lambda^{(\alpha)}, g_{\min}^{(\alpha)} \right) \right\}$ and thresholds $\left\{ \left(\theta^{(\alpha)}, \theta_{\text{spatial}}^{(\alpha)} \right) \right\}$ is estimated by

- generating a second (independent) permutation set $\{\pi_r'\}$,

- for each combination of parameters α, use equation (11) replacing π_r with π_r' to calculate $\hat{y}_{\max}'^{(\alpha)}$ and equation (12) similarly to calculate $\hat{\hat{y}}_{\max}'^{(\alpha)}$ (if $\lambda^{(\alpha)} > 1$; otherwise, take $\hat{\hat{y}}_{\max}'^{(\alpha)} = \infty$).

The set of reps yielding false positive hits is calculated according to:

$$\text{false positives} = \left\{ r \ \middle| \ \bigvee_\alpha \left[\left(\hat{y}_{\max}'^{(\alpha)} \geq \theta^{(\alpha)} \right) \wedge \left(\hat{\hat{y}}_{\max}'^{(\alpha)} \geq \theta_{\text{spatial}}^{(\alpha)} \right) \right] \right\} \tag{26}$$

where we again take either $\theta^{(\alpha)} = -\infty$ when $\lambda^{(\alpha)} = 0$ or $\theta_{\text{spatial}}^{(\alpha)} = -\infty$ when $\lambda^{(\alpha)} > 1$, so that, depending on the value of $\lambda^{(\alpha)}$, one or the other of the two logical expressions inside the square brackets in equation (26) is trivially true for each α.

The false positive rate can then be estimated by comparing the number of false positive results with the number `reps` of permutations π_r' generated. SArKS uses `binom::binom.exact` to

estimate confidence intervals for the false positive rate according to the Pearson-Klopper method.

Note regarding random number generator seeds: In order that the permutation set $\{\pi_r'\}$ be independent of the initial permutation set $\{\pi_r\}$ used to select thresholds $\left\{\left(\theta^{(\alpha)}, \theta^{(\alpha)}_{\text{spatial}}\right)\right\}$, you should make sure that you do not repeat the same seed for the random number generator in the calls to `permutationDistribution` and `estimateFalsePositiveRate`.

4 Session Info

```
> sessionInfo()
```

```
R version 4.1.0 (2021-05-18)
Platform: x86_64-pc-linux-gnu (64-bit)
Running under: Ubuntu 20.04.2 LTS

Matrix products: default
BLAS:   /home/biocbuild/bbs-3.13-bioc/R/lib/libRblas.so
LAPACK: /home/biocbuild/bbs-3.13-bioc/R/lib/libRlapack.so

locale:
 [1] LC_CTYPE=en_US.UTF-8        LC_NUMERIC=C
 [3] LC_TIME=en_GB              LC_COLLATE=C
 [5] LC_MONETARY=en_US.UTF-8     LC_MESSAGES=en_US.UTF-8
 [7] LC_PAPER=en_US.UTF-8        LC_NAME=en_US.UTF-8
 [9] LC_ADDRESS=en_US.UTF-8      LC_TELEPHONE=en_US.UTF-8
[11] LC_MEASUREMENT=en_US.UTF-8   LC_IDENTIFICATION=en_US.UTF-8

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] ggplot2_3.3.3 sarks_1.4.0   rJava_1.0-4

loaded via a namespace (and not attached):
 [1] compiler_4.1.0         pillar_1.6.1           GenomeInfoDb_1.28.0
 [4] XVector_0.32.0         bitops_1.0-7           tools_4.1.0
 [7] zlibbioc_1.38.0        digest_0.6.27          lifecycle_1.0.0
[10] tibble_3.1.2           gtable_0.3.0           pkgconfig_2.0.3
[13] rlang_0.4.11           DBI_1.1.1              rstudioapi_0.13
[16] parallel_4.1.0         GenomeInfoDbData_1.2.6 withr_2.4.2
[19] dplyr_1.0.6            cluster_2.1.2          Biostrings_2.60.0
[22] generics_0.1.0         S4Vectors_0.30.0       vctrs_0.3.8
[25] IRanges_2.26.0         stats4_4.1.0           grid_4.1.0
[28] tidyselect_1.1.1       glue_1.4.2             R6_2.5.0
[31] fansi_0.4.2            binom_1.1-1            farver_2.1.0
[34] purrr_0.3.4            magrittr_2.0.1         scales_1.1.1
[37] ellipsis_0.3.2         BiocGenerics_0.38.0    assertthat_0.2.1
```

```
[40] colorspace_2.0-1      labeling_0.4.2       utf8_1.2.1
[43] RCurl_1.98-1.3        munsell_0.5.0        crayon_1.4.1
```

5 Notation glossary

$\lfloor r \rceil$	nearest integer to real number r
$u * v$	concatenation of strings u and v
$\lvert u \rvert$	length of string u
$u[i, j)$	substring of u starting at i^{th} character (inclusive) and continuing up until j^{th} character (exclusive) using 0-based indexing
$u[i, j]$	substring of u starting at i^{th} character (inclusive) and continuing through the jth character (inclusive) using 0-based indexing
$\{a \mid C(a)\}$	set containing all elements a satisfying condition $C(a)$
$\{a \mid C_1(a) \wedge C_2(a)\}$	set containing all elements a satisfying conditions $C_1(a)$ and $C_2(a)$
$\{a \mid C_1(a) \vee C_2(a)\}$	set containing all elements a satisfying conditions $C_1(a)$ or $C_2(a)$
$\mathbb{E}[A]$	expectation value of random variable A
$\mathbb{V}[A]$	variance of random variable A
i	suffix array index: 0-based position of suffix in lexicographically sorted list of all suffixes of string x
s_i	suffix array value: 0-based spatial position of suffix with suffix array index i within string x
i_s	suffix array index i associated with spatial position s
b_i	block array value: 0-based position of block/word in which suffix with suffix array index i begins
ω_i	0-based position of suffix with suffix array index i within block b_i
$\hat{y}_i$	kernel smoothed score associated with suffix array index i
κ	half-width of kernel applied to generate $\hat{y}_i$
$\hat{k}_i$	estimate of smoothed k-mer length at suffix array index i
$\eta(i)$	negative spatial shift operator defined by $\eta(i) = i_{s_i-1}$
$\rho(i)$	positive spatial shift operator defined by $\rho(i) = i_{s_i+1}$
θ	threshold value for $\hat{y}_i$ for sequence-smoothed peak calling
I	set of suffix array indices identified as peaks by SArKS
M	set of k-mer motifs derived from suffix array peak set
$f_b^{(i)}$	weighted frequency of block/word b within smoothing window centered on suffix array index i
g_i	Gini impurity of smoothing window centered on suffix array index i
$g_{\min}$	minimum value of smoothing window Gini impurity for inclusion in peak set I
$\bar{g}_i$	spatially-averaged Gini impurity over spatial window starting at position s_i
$\bar{g}_{\min}$	minimum value of spatially-averaged Gini impurity for inclusion in peak set J_{spatial}
$\hat{\hat{y}}_i$	spatially smoothed score associated with suffix array index i
λ	length of spatial kernel applied to generate spatially smoothed scores $\hat{\hat{y}}_i$
$\hat{\hat{k}}_i$	estimate of merged k-mer length at suffix array index i
θ_{spatial}	threshold value for $\hat{\hat{y}}_i$ to call significant spatial windows
J_{spatial}	set of suffix array indices identified as spatial window starting positions
I_{spatial}	set of suffix array indices identified as k-mer starting positions using spatial smoothing
M_{spatial}	set of k-mer motifs derived from suffix array index set I_{spatial} using spatial smoothing
π	permutation of n blocks/words
Π	random variable representing randomly generated permutation
$\hat{y}_i^{(\pi)}$	sequence smoothed scores calculated with word scores permuted by π
$\hat{\hat{y}}_i^{(\pi)}$	spatially smoothed scores calculated with word scores permuted by π

References

J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *International Colloquium on Automata, Languages, and Programming*, pages 943–955. Springer, 2003.

D. Wylie, H. Hofmann, and B. Zemelman. SArKS: de novo discovery of gene expression regulatory motif sites and domains by suffix array kernel smoothing. *Bioinformatics*, 35(20):3944–3952, 2019.