Searching Biological Sequences for Research

Erik S. Wright

February 26, 2024

Contents

1 Introduction

Sequence searching is an essential part of biology research. The word *research* even originates from a word in Old French meaning 'to search'. Yet, the sheer amount of biological sequences to comb through can make (re)search feel like finding a needle in a haystack. To avoid heading out on a wild goose chase, it's important to master the ins and outs of searching. The goal of this vignette is to help you leave no stone unturned as you scout out homologous sequences.

2 Getting Started

2.1 Startup

To get started we need to load the DECIPHER package, which automatically loads a few other required packages.

```
> library(DECIPHER)
```

Searching makes use of the `IndexSeqs` and `SearchIndex` functions. Help can be accessed through:

```
> ? SearchIndex
```

Once DECIPHER is installed, the code in this tutorial can be obtained via:

```
> browseVignettes("DECIPHER")
```

2.2 Gathering the evidence

There are umpteen reasons to search through biological sequences. For the purposes of this vignette, we are going to focus on finding homologous proteins in a genome. In this case, our pattern (query) is the protein sequence and the subject (target) is a 6-frame translation of the genome. Feel free to follow along with your own (nucleotide or protein) sequences or use those in the vignette:

```
> # specify the path to your file of pattern (query) sequences:
> fas1 <- "<<path to pattern FASTA file>>"
> # OR use the example protein sequences:
> fas1 <- system.file("extdata",
          "PlanctobacteriaNamedGenes.fas.gz",
          package="DECIPHER")
> # read the sequences into memory
> pattern <- readAAStringSet(fas1)
> pattern
AAStringSet object of length 2497:
       width seq                                               names
   [1]   227 MAGPKHVLLVSEHWDLFFQTKE...VGYLFSDDGDKKFSQQDTKLS A0A0H3MDW1|Root;N...
   [2]   394 MKRNPHFVSLTKNYLFADLQKR...GKREDILAACERLQMAPALQS O84395|Root;2;6;1...
   [3]   195 MAYGTRYPTLAFHTGGIGESDD...GFCLTALGFLNFENAEPAKVN Q9Z6M7|Root;4;1;1...
   [4]   437 MMLRGVHRIFKCFYDVVLVCAF...TASFDRTWRALKSYIPLYKNS Q46222|Root;2;4;9...
   [5]   539 MSFKSIFLTGGVVSSLGKGLTA...FIEFIRAAKAYSLEKANHEHR Q59321|Root;6;3;4...
   ...   ... ...
[2493]  1038 MFEEVLQESFDEREKKVLKFWQ...EGTDWDLNGEPTKIIIKKSEY Q6MDY1|Root;6;1;1...
[2494]   102 MVQIVSQDNFADSIASGLVLVD...VERSVGLKDKDSLVKLISKHQ Q9PJK3|Root;NoEC;...
[2495]   224 MKPQDLKLPYFWEDRCPKIENH...NLWRSKGEKIFCTEFVKRVGI Q9PL91|Root;2;1;1...
[2496]   427 MLRRLFVSTFLIFGMVSLYAKD...KIVIGLGEKRFPSWGGFPNNQ Q256H8|Root;NoEC;...
[2497]   344 MLTLGLESSCDETACALVDAKG...GIHPCARYHWESISASLSPLP Q822Y4|Root;2;3;1...
```

Protein search is more accurate than nucleotide search, so we are going to import a genome and perform 6-frame translation to get the subject sequences. Feel free to carry on without translating the sequences if you are searching nucleotides or otherwise would prefer to skip translation. Note that `SearchIndex` only searches the nucleotides in the direction they are provided, so if you desire to search both strands then you will need to combine with the `reverseComplement` as shown below.

```
> # specify the path to your file of subject (target) sequences:
> fas2 <- "<<path to subject FASTA file>>"
> # OR use the example subject genome:
> fas2 <- system.file("extdata",
          "Chlamydia_trachomatis_NC_000117.fas.gz",
          package="DECIPHER")
> # read the sequences into memory
> dna <- readDNAStringSet(fas2)
> dna
DNAStringSet object of length 1:
      width seq                                               names
[1] 1042519 GCGGCCGCCCGGGAAATTGCTA...GTTGGCTGGCCCTGACGGGGTA NC_000117.1 Chlam...
> subject <- subseq(rep(dna, 3), 1:3) # 3-frames
> subject <- c(subject, reverseComplement(subject)) # 6-frames
> subject <- suppressWarnings(translate(subject)) # 6-frame translation
> subject
```

```
AAStringSet object of length 6:
     width seq                                                 names
[1] 347506 AAAREIAKRWEQRVRDLQDKGAA...GCVHK*VRGSFRSEQVGWP*RG NC_000117.1 Chlam...
[2] 347506 RPPGKLLKDGSKELEIYKIKVLH...AAYTSECADHLEANKLAGPDGV NC_000117.1 Chlam...
[3] 347505 GRPGNC*KMGAKS*RSTR*RCCT...WLRTQVSARII*KRTSWLALTG NC_000117.1 Chlam...
[4] 347506 YPVRASQLVRF*MIRALTCVRSH...QHLYLVDL*LFAPIF*QFPGRP NC_000117.1 Chlam...
[5] 347506 YPVRASQLVRF*MIRALTCVRSH...QHLYLVDL*LFAPIF*QFPGRP NC_000117.1 Chlam...
[6] 347505 YPVRASQLVRF*MIRALTCVRSH...VQHLYLVDL*LFAPIF*QFPGR NC_000117.1 Chlam...
```

3 Searching for hits between pattern and subject sequences

Once the sequences are imported, we need to build an *InvertedIndex* object from the `subject` sequences. We can accomplish this with `IndexSeqs` by specifying the k-mer length (*K*). If you don't know what value to use for *K*, then you can specify *sensitivity*, *percentIdentity*, and *patternLength* in lieu of *K*. Here, we want to ensure we find 99% (0.99) of sequences with at least 70% identity to a pattern with 300 or more residues. Note that *sensitivity* is defined as a fraction, whereas *percentIdentity* is defined as a percentage.

```
> index <- IndexSeqs(subject,
          sensitivity=0.99,
          percentIdentity=70,
          patternLength=300,
          processors=1)
================================================================================

Time difference of 0.93 secs
> index
  An InvertedIndex built with:
   * Amino acid sequences: 6
   * Total k-mers: 1,266,786
   * Alphabet: A, C, DE, FWY, G, H, ILMV, N, P, Q, RK, ST
   * K-mer size: 5
   * Step size: 1
```

Printing the `index` shows that we created an *InvertedIndex* object containing over 1 million 5-mers in a reduced amino acid alphabet with 12 symbols. Before we can find homologous hits to our protein sequences with `SearchIndex`, we must decide what *type* of results we desire. The default *type*, `"one"`, is to return the best hit per subject sequence (i.e., *one* per subject). Alternatively, we could request only the `"top"` hit per pattern sequence to obtain up to a single hit per `pattern` sequence. For the purposes of this vignette, we are going to ask for `"all"` hits above the *minScore*. If we do not specify *minScore* then it is automatically set based on the size of the *InvertedIndex*.

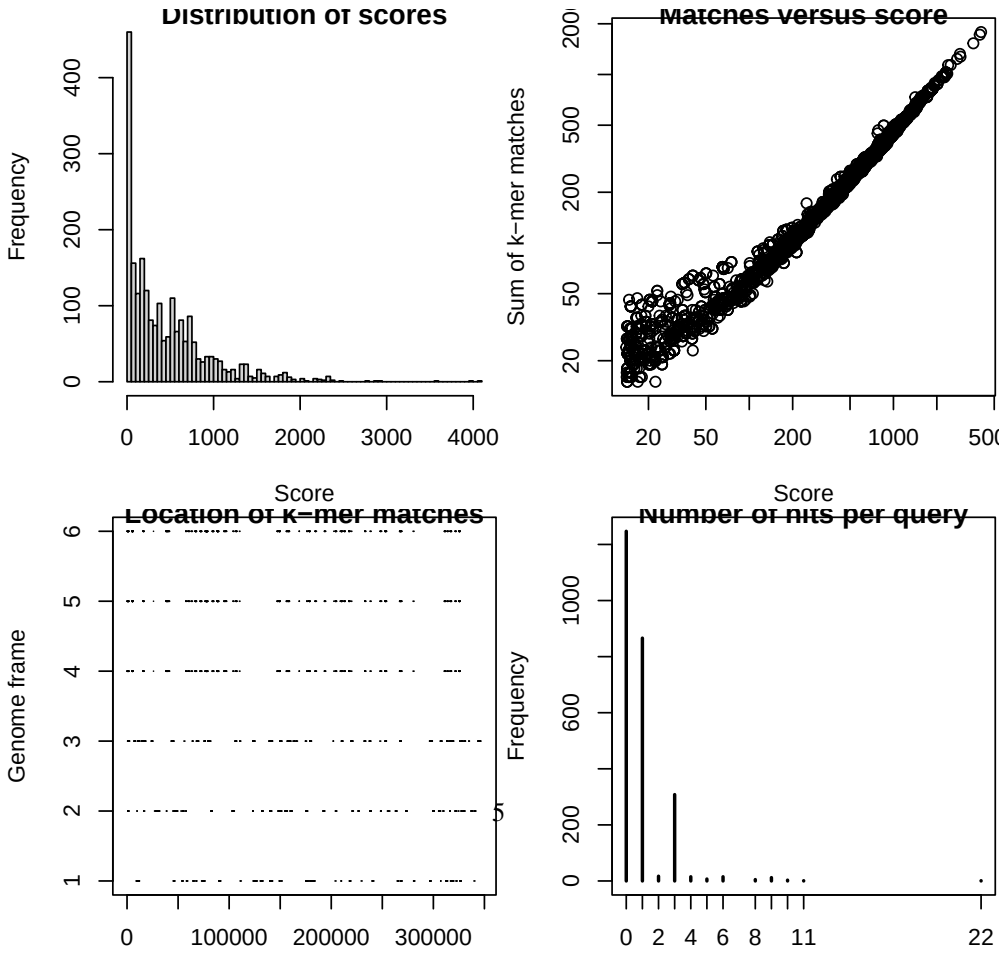
```
> hits <- SearchIndex(pattern,
          index,
          type="all",
          processors=1)
================================================================================

Time difference of 7.19 secs
> dim(hits)
[1] 2220    4
```

```
> head(hits)
  Pattern Subject     Score     Position
1       2       2  902.3000 1, 394, ....
2       3       3  394.5042 3, 67, 1....
3       5       3 1225.5021 1, 539, ....
4      10       1  554.5294 5, 13, 2....
5      11       1  251.8765 5, 16, 2....
6      12       1  856.2874 1, 375, ....
```

The result of our search is a *data.frame* with four columns: Pattern (index in `pattern`), Subject (index in `subject`), Score, and Position (of k-mer matches). We can take a closer look at the number of hits per protein, their scores, and locations (Fig. 1):

```
> layout(matrix(1:4, nrow=2))
> hist(hits$Score,
         breaks=100,
         xlab="Score",
         main="Distribution of scores")
> plot(NA,
         xlim=c(0, max(width(subject))),
         ylim=c(1, 6),
         xlab="Genome position",
         ylab="Genome frame",
         main="Location of k-mer matches")
> segments(sapply(hits$Position, `[`, i=3), # third row
         hits$Subject,
         sapply(hits$Position, `[`, i=4), # fourth row
         hits$Subject)
> plot(hits$Score,
         sapply(hits$Position,
                 function(x)
                         sum(x[2,] - x[1,] + 1)),
         xlab="Score",
         ylab="Sum of k-mer matches",
         main="Matches versus score",
         log="xy")
> plot(table(tabulate(hits$Pattern, nbins=length(pattern))),
         xlab="Hits per pattern sequence",
         ylab="Frequency",
         main="Number of hits per query")
```



The calculated *Score* for each search hit is defined by the negative log-odds of observing the hit by chance. We see that most scores were near zero, but there were many high scoring hits. Hits tended to be clustered along specific frames of the genome, with some genome regions devoid of hits. Also, most pattern (protein) sequences were found at zero or one location in the genome. As expected, a hit's score is correlated with the length of k-mer matches, although distance between matches lowers the score. One protein was found many times more than all the others. We can easily figure out which protein was found the most times:

```
> w <- which.max(tabulate(hits$Pattern))
> hits[hits$Pattern == w,]
     Pattern Subject    Score     Position
1363    1622       1 19.24231 331, 335....
1364    1622       2 16.72343 226, 232....
1365    1622       2 19.00421 144, 148....
1366    1622       2 28.60440 203, 207....
1367    1622       3 15.53163 360, 364....
1368    1622       3 15.07425 228, 232....
1369    1622       3 15.28678 329, 334....
1370    1622       3 36.66560 461, 465....
1371    1622       3 23.51312 217, 223....
1372    1622       3 40.15768 329, 333....
1373    1622       3 24.29044 226, 231....
1374    1622       3 70.97239 618, 622....
1375    1622       3 37.29174 636, 640....
1376    1622       4 14.40465 8, 16, 3....
1377    1622       4 50.25471 460, 464....
1378    1622       4 28.52114 357, 361....
1379    1622       5 14.40465 8, 16, 3....
1380    1622       5 50.25471 460, 464....
1381    1622       5 28.52114 357, 361....
1382    1622       6 14.40466 8, 16, 3....
1383    1622       6 50.25471 460, 464....
1384    1622       6 28.52115 357, 361....
> names(pattern)[w]
[1] "Q9Z899|Root;NoEC;pmp6"
```

The most frequent protein turned out to be from the class polymorphic membrane proteins (i.e., *pmp*) commonly found in our target genome (*Chlamydia*). Likely these hits are to multiple paralogous genes on the genome, as can be seen by the wide distribution of scores.

4 Aligning the search hits between pattern and subject

So far, we've identified the location and score of search hits without alignment. Aligning the hits would provide us with their local start and stop boundaries, percent identity, and the locations of any insertions or deletions. Thankfully, alignment is elementary once we've completed our search.

```
> aligned <- AlignPairs(pattern=pattern,
         subject=subject,
         pairs=hits,
         processors=1)
================================================================================

Time difference of 0.84 secs
```

```
> head(aligned)
  Pattern PatternStart PatternEnd Subject SubjectStart SubjectEnd Matches
1       2            1        394       2       148163     148556     394
2       3            1        195       3       141952     142146     173
3       5            1        539       3        68143      68681     539
4      10            1        370       1       280713     281082     297
5      11            1        369       1       280713     281080     199
6      12            1        375       1       280713     281087     375
  Mismatches AlignmentLength    Score PatternGapPosition PatternGapLength
1          0             394 2432.598
2         22             195 1153.484
3          0             539 3385.075
4         73             370 1946.666
5        168             370 1274.818                246                 1
6          0             375 2354.977
  SubjectGapPosition SubjectGapLength
1
2
3
4
5                273                 2
6
```

The `AlignPairs` function returns a *data.frame* containing the Pattern (i.e., `pattern` index), Subject (i.e., `subject` index), their start and end positions, the number of matched and mismatched positions, the alignment length and its score, as well as the location of any gaps in the *pattern* or *subject*. We can use this information to calculate a percent identity, which can be defined a couple of different ways (Fig. 2).

```
> PID1 <- aligned$Matches/(aligned$Matches + aligned$Mismatches)
> PID2 <- aligned$Matches/aligned$AlignmentLength
> layout(matrix(1:4, ncol=2))
> plot(hits$Score, PID2,
         xlab="Hit score",
         ylab="Matches / (Aligned length)")
> plot(hits$Score, aligned$Score,
         xlab="Hit score",
         ylab="Aligned score")
> plot(aligned$Score, PID1,
         xlab="Aligned score",
         ylab="Matches / (Matches + Mismatches)")
> plot(PID1, PID2,
         xlab="Matches / (Matches + Mismatches)",
         ylab="Matches / (Aligned length)")
```

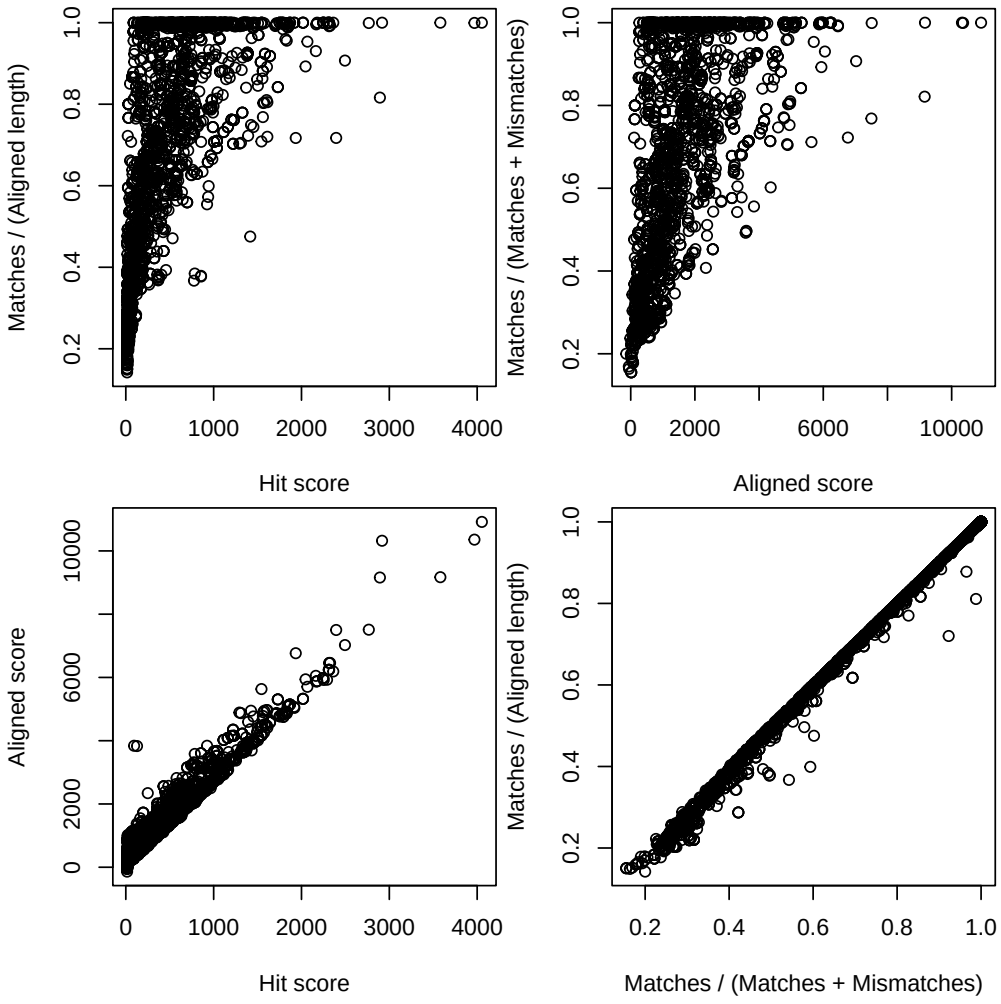


Figure 2: Scatterplots of different scores and methods of formulating percent identity.

AlignPairs gives us everything we need to align the sequences except the alignments themselves. If needed, we can easily get the pairwise alignments using the location(s) of gaps (i.e., "-") in the pattern and subject sequences.

```
> patterns <- replaceAt(subseq(pattern[aligned$Pattern],
                aligned$PatternStart,
                aligned$PatternEnd),
         aligned$PatternGapPosition,
         lapply(aligned$PatternGapLength,
                function(x)
                        sapply(x,
                                function(l)
                                        paste(rep("-", l), collapse=""))))
> subjects <- replaceAt(subseq(subject[aligned$Subject],
                aligned$SubjectStart,
                aligned$SubjectEnd),
         aligned$SubjectGapPosition,
         lapply(aligned$SubjectGapLength,
                function(x)
                        sapply(x,
                                function(l)
                                        paste(rep("-", l), collapse=""))))
> c(patterns[1], subjects[1]) # view the first pairwise alignment
AAStringSet object of length 2:
    width seq                                                 names
[1]   394 MKRNPHFVSLTKNYLFADLQKRV...SLGKREDILAACERLQMAPALQS O84395|Root;2;6;1...
[2]   394 MKRNPHFVSLTKNYLFADLQKRV...SLGKREDILAACERLQMAPALQS NC_000117.1 Chlam...
```

5 Calibrating an expect value (E-value) from hit scores

SearchIndex returns a score with significant matches above *minScore*. However, it is often useful to compute an expect value (E-value) representing the number of times we expect to see a hit at least as high scoring in a database of the same size. A lesser known fact is that E-values are a function of the substitution matrix, *gapOpening* penalty, *gapExtension* penalty, and other search parameters, so E-values must be empirically determined.

There are two straightforward ways to calibrate E-values: (1) create an equivalent database of random sequences with matched composition to the input, or (2) search for the reverse of the sequences under the assumption that reverse hits are unexpected (i.e., false positives). The second approach is more conservative, because we will find more hits than expected if its underlying assumption is not true. Here, we will try the second approach:

```
> revhits <- SearchIndex(reverse(pattern), # reverse the query
         index, # keep the same target database
         minScore=10, # set low to get many hits
         type="all", # get all hits, as in the original query
         processors=1)
================================================================================

Time difference of 6.75 secs
> dim(revhits)
[1] 148   4
```

Next, our goal is to fit the distribution of background scores (i.e., reverse hits), which is reasonably well-modeled by an exponential distribution. We will bin the reverse hits' scores into intervals of one score unit between `10` and `100`. Then we will use the fact that the integral of $e^{-rate*x}$ (with respect to x) is $e^{-rate*x}$ to fit the *rate* of the distribution. Note how there is an outlying point that violated the assumption reversed sequences should not have strong hits (Fig. 3). We can use the *sum of absolute error* (L1 norm) rather than the *sum of squared error* (L2 norm) to make the fit more robust to outliers. We will perform the fit in log-space to emphasize points across many orders of magnitude.

```
> X <- 10:100 # score bins
> Y <- tabulate(.bincode(revhits$Score, X), length(X) - 1)
> Y <- Y/length(pattern) # average per query
> w <- which(Y > 0) # needed to fit in log-space
> plot(X[w], Y[w],
         log="y",
         xlab="Score",
         ylab="Average false positives per query")
> fit <- function(rate) # integrate from bin start to end
         sum(abs((log((exp(-X[w]*rate) -
                 exp(-X[w + 1]*rate))*length(subject)) -
                 log(Y[w]))))
> o <- optimize(fit, c(0.01, 2)) # optimize rate
> lines(X[-length(X)], (exp(-X[-length(X)]*o$minimum) -
         exp(-(X[-1])*o$minimum))*length(subject))
> rate <- o$minimum
> print(rate)
[1] 0.4353657
```

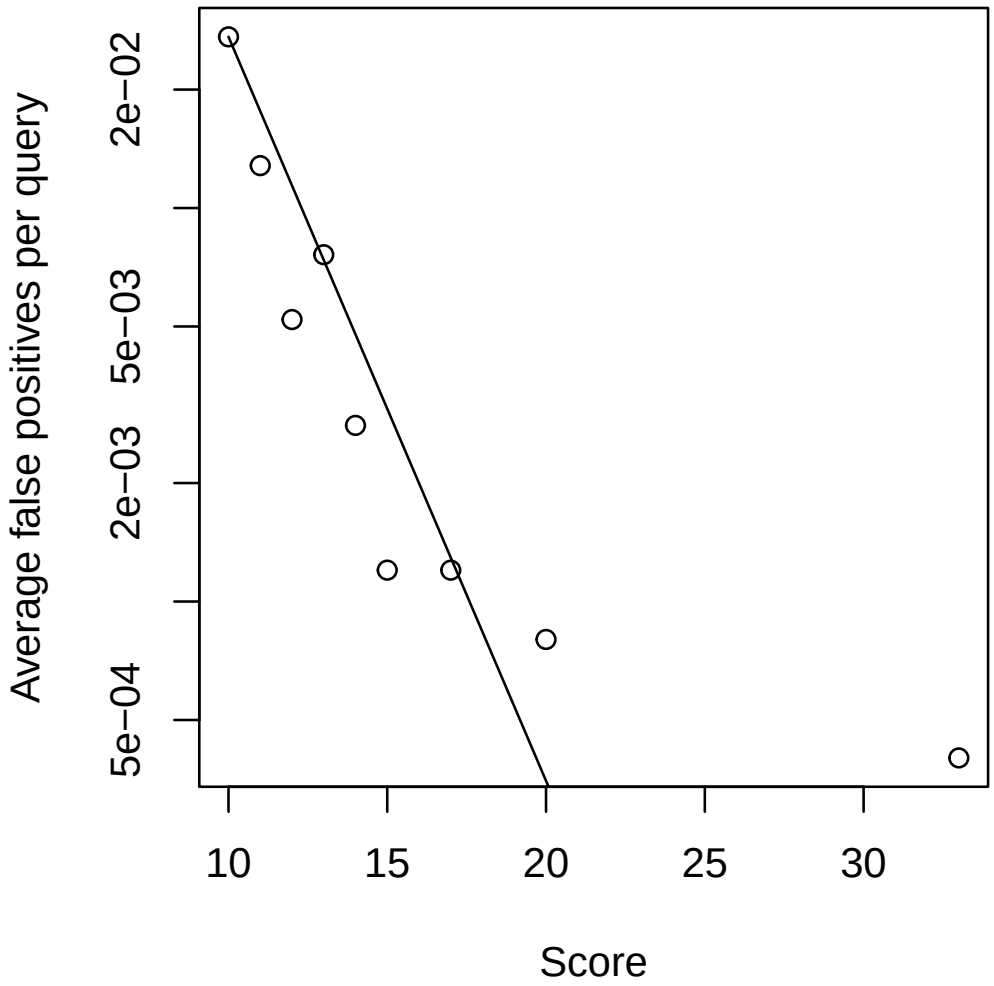


Figure 3: Fitting an exponential distribution to the score background.

Now that we've optimized the *rate* parameter, it is feasible to convert our original scores into E-values. We are interested in the number of false positive hits expected across all queries at every value of `Score`. This differs from the standard definition of E-value, which is defined on a per query basis. However, since we are performing multiple queries it is preferable to apply a multiple testing correction for the number of searches. We can convert our original scores to E-values, as well as define score thresholds for a given number of acceptable false positives across all `pattern` sequences:

```
> # convert each Score to an E-value
> Evalue <- exp(-rate*hits$Score)*length(subject)*length(pattern)
> # determine minimum Score for up to 1 false positive hit expected
> log(1/length(subject)/length(pattern))/-rate
[1] 22.08397
```

As can be seen, for this particular combination of dataset and parameters, a score threshold of `22` is sufficient to only permit one combined false positive across all queries. Since E-value is a function of the dataset's size and the specific search parameters, you should calibrate the E-value for each set of searches performed. Once you have calibrated the *rate*, it is straightforward to find only those hits that are statistically significant:

```
> # determine minimum Score for 0.05 (total) false positive hits expected
> threshold <- log(0.05/length(subject)/length(pattern))/-rate
> hits <- hits[hits$Score > threshold,]
```

6 Maximizing search sensitivity to find distant hits

We've already employed some strategies to improve search sensitivity: choosing a small value for k-mer length and step size, searching amino acids rather than nucleotides, and masking low complexity regions and repeats. Although k-mer search is very fast, sometimes k-mers alone are insufficient to find distant homologs. In these cases, search sensitivity can be improved by providing the `subject` (target) sequences, which causes `SearchIndex` to extend k-mer matches to their left and right. This is as simple as adding a single argument:

```
> # include the target sequences to increase search sensitivity
> hits <- SearchIndex(pattern,
          index,
          subject, # optional parameter
          type="all",
          processors=1)
================================================================================

Time difference of 16.07 secs
> dim(hits)
[1] 5190    4
> head(hits)
  Pattern Subject      Score     Position
1       2       2 1159.81193 1, 394, ....
2       3       3  532.70804 1, 71, 1....
3       4       2   17.50102 15, 19, ....
4       4       2   14.48302 30, 35, ....
5       5       1   14.97818 5, 16, 7....
6       5       2   14.49230 5, 16, 2....
```

We can see that search took longer when providing `subject` sequences, but the number of hits also increased. The *dropScore* parameter, which controls the degree of extension, can be adjusted to balance sensitivity and speed. In this manner, high sensitivity can be achieved by providing `subject` sequences in conjunction with a low value of k-mer length and *dropScore*.

7 Session Information

All of the output in this vignette was produced under the following conditions:

- R Under development (unstable) (2024-01-16 r85808), `x86_64-pc-linux-gnu`
- Running under: `Ubuntu 22.04.3 LTS`
- Matrix products: default
- BLAS: `/home/biocbuild/bbs-3.19-bioc/R/lib/libRblas.so`
- LAPACK: `/usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.10.0`
- Base packages: base, datasets, graphics, grDevices, methods, stats, stats4, utils
- Other packages: BiocGenerics 0.49.1, Biostrings 2.71.2, DECIPHER 2.31.3, GenomeInfoDb 1.39.6, IRanges 2.37.1, S4Vectors 0.41.3, XVector 0.43.1
- Loaded via a namespace (and not attached): bitops 1.0-7, compiler 4.4.0, crayon 1.5.2, DBI 1.2.2, GenomeInfoDbData 1.2.11, KernSmooth 2.23-22, RCurl 1.98-1.14, tools 4.4.0, zlibbioc 1.49.0