The Magic of Gene Finding

Erik S. Wright

April 25, 2023

Contents

1 Introduction **1**

2 Getting Started **2**
2.1 Startup . . . 2

3 Finding Genes in a Genome **2**
3.1 Importing the genome . . . 2
3.2 Finding genes . . . 2
3.3 Inspecting the output . . . 4

4 Analyzing the Output **4**
4.1 Extracting genes from the genome . . . 4
4.2 Removing genes with too many ambiguities . . . 6
4.3 Taking a look at the shortest genes . . . 6
4.4 Revealing the secrets of gene finding . . . 7
4.5 Taking a closer look at the output . . . 9

5 Incorporating non-coding RNAs **11**
5.1 Examining the intergenic spaces . . . 11
5.2 Finding and including non-coding RNAs . . . 14

6 Annotating the protein coding genes **16**

7 Guaranteeing repeatability **19**

8 Exporting the output **19**

9 Session Information **19**

1 Introduction

This vignette reveals the tricks behind the magic that is *ab initio* gene finding. Cells all have the magical ability to transcribe and translate portions of their genome, somehow decoding key signals from a sea of possibilities. The `FindGenes` function attempts to decipher these signals in order to accurately predict an organism's set of genes. Cells do much of this magic using only information upstream of the gene, whereas `FindGenes` uses both the content of the gene and its upstream information to predict gene boundaries. As a case study, this tutorial focuses on finding genes in the genome of *Chlamydia trachomatis*, an intracellular bacterial pathogen known for causing chlamydia. This genome was chosen because it is relatively small (only 1 Mbp) so the examples run quickly. Nevertheless, `FindGenes` is designed to work with any genome that lacks introns, making it well-suited for prokaryotic gene finding.

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)
```

Gene finding is performed with the function `FindGenes`. Help can be accessed via:

```
> ? FindGenes
```

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

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

3 Finding Genes in a Genome

Ab initio gene finding begins from a genome and locates genes without prior knowledge about the specific organism.

3.1 Importing the genome

The first step is to set filepaths to the genome sequence (in FASTA format). Be sure to change the path names to those on your system by replacing all of the text inside quotes labeled "<<path to ...>>" with the actual path on your system.

```
> # specify the path to your genome:
> genome_path <- "<<path to genome FASTA file>>"
> # OR use the example genome:
> genome_path <- system.file("extdata",
          "Chlamydia_trachomatis_NC_000117.fas.gz",
          package="DECIPHER")
> # read the sequences into memory
> genome <- readDNAStringSet(genome_path)
> genome
DNAStringSet object of length 1:
      width seq                                                names
[1] 1042519 GCGGCCGCCCGGGAAATTGCTA...GTTGGCTGGCCCTGACGGGGTA NC_000117.1 Chlam...
```

3.2 Finding genes

The next step is to find genes in the genome using `FindGenes`, which does all the magic. There are fairly few choices to make at this step. By default, the bacterial and archaeal genetic code is used for translation, including the initiation codons "ATG", "GTG", "TTG", "CTG", "ATA", "ATT", and "ATC". The default *minGeneLength* is `60`, although we could set this lower (e.g., `30`) to locate very short genes or higher (e.g., `90`) for (only slightly) better accuracy. The argument *allowEdges* (default `TRUE`) controls whether genes are allowed to run off the ends of the genome, as would be expected for circular or incomplete chromosomes. Here, we will only set *showPlot* to `TRUE` to display a summary of the gene finding process and *allScores* to `TRUE` to see the scores of all open reading frames (including predicted genes).

```
> orfs <- FindGenes(genome, showPlot=TRUE, allScores=TRUE)
```

Iter	Models	Start	Motif	Init	Fold	UpsNt	Term	RBS	Auto	Stop	Genes
1	1	18.40			_						886
1	1	18.40			\						886
1	1	18.40			|						886
1	1	18.40			/						886
1	1	18.40			_						886
1	1	18.40			\						886
1	1	18.40			|						886
1	1	18.40			/						886
1	1	18.40			_						886
2	12	18.40	0.48	0.90							888
3	15	18.38	0.53	1.16	0.39	1.41					891
4	15	15.59	0.51	1.26	0.46	1.76	0.24				894
5	12	15.62	0.52	1.24	0.47	1.83	0.25	0.95	0.10	0.07	897
6	13	15.61	0.51	1.24	0.51	1.91	0.25	1.06	0.10	0.07	897
7	14	15.60	0.51	1.25	0.52	1.93	0.25	1.04	0.10	0.07	897
8	11	15.59	0.51	1.25	0.50	1.95	0.25	1.03	0.10	0.07	897
9	12	15.57	0.51	1.25	0.51	1.97	0.25	1.06	0.10	0.07	896
10	12	15.57	0.51	1.26	0.48	1.99	0.25	1.05	0.11	0.07	896
11	12	15.57	0.51	1.26	0.51	1.99	0.25	1.07	0.11	0.07	896

```
Time difference of 35.53 secs
```

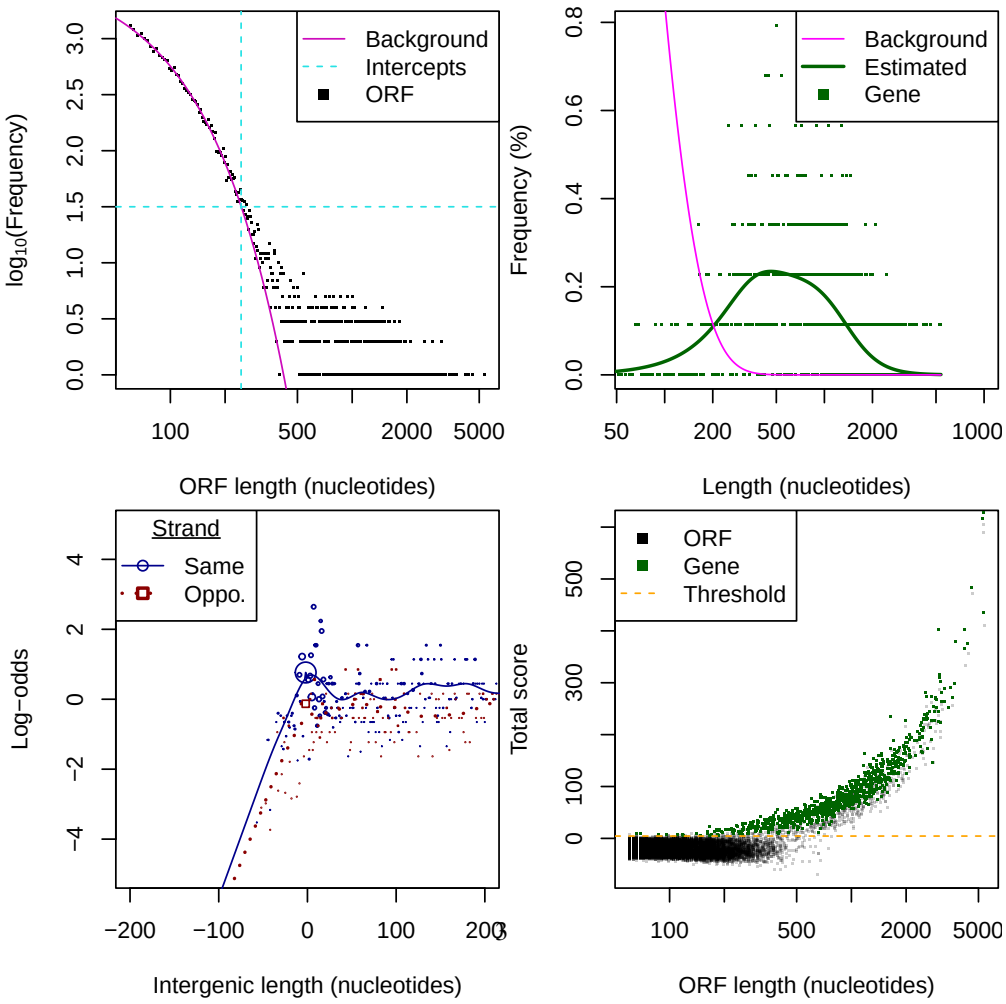


Figure 1: Plot summarizing the gene finding process of `FindGenes`. See `?plot.Genes` for details.

3.3 Inspecting the output

And presto! With a wave of our magic wand we now have our gene predictions in the form of an object belonging to class *Genes*. Now that we have our genes in hand, let's take a look at them:

```
> orfs
Genes object of size 109,490 specifying:
896 protein coding genes from 66 to 5,361 nucleotides.
108,594 open reading frames from 60 to 5,370 nucleotides.

  Index Strand Begin  End TotalScore ... Gene
1     1      0     1 1176      77.94 ...    1
2     1      0     2   82      -4.63 ...    0
3     1      0    16 1176      62.07 ...    0
4     1      0    17   82      -8.67 ...    0
5     1      1    41  106     -13.61 ...    0
6     1      1    91  150     -34.87 ...    0
... with 109,484 more rows.
```

Open reading frames are defined by their `Begin` and `End` position, as well as whether they are on the top (i.e., `Strand` is `0`) or bottom strand. Here, genes are flagged by having a non-zero value in the `Gene` column. We see all the open reading frames in the output because *allScores* was set to `TRUE`. If we only want to look at the subset of open reading frames that are predicted as genes, we can subset the object:

```
> genes <- orfs[orfs[, "Gene"]==1,]
```

The `"Gene"` column is one of several describing the open reading frames. Objects of class *Genes* are stored as matrices with many columns:

```
> colnames(genes)
 [1] "Index"                   "Strand"
 [3] "Begin"                   "End"
 [5] "TotalScore"              "LengthScore"
 [7] "CodingScore"             "CodonModel"
 [9] "CouplingScore"           "StartScore"
[11] "StopScore"               "InitialCodonScore"
[13] "TerminationCodonScore"   "RibosomeBindingSiteScore"
[15] "AutocorrelationScore"    "UpstreamNucleotideScore"
[17] "UpstreamMotifScore"      "FoldingScore"
[19] "FractionReps"            "Gene"
```

4 Analyzing the Output

4.1 Extracting genes from the genome

Predictions in hand, the first thing we can do is extract the genes from the genome. This can be easily done using `ExtractGenes`.

```
> dna <- ExtractGenes(genes, genome)
> dna
```

```
DNAStringSet object of length 896:
      width seq
  [1]  1176 GCGGCCGCCCGGGAAATTGCTAAAAGATGGGAG...GGCGTTGGAATAGAGAAGTCGATAGGGAATAA
  [2]   273 ATGCTTTGTAAAGTTTGTAGAGGATTATCTTCT...GATACCACCACCATCATATGGATAGAGAATAA
  [3]   303 ATGACAGAGTCATATGTAAACAAAGAAGAAATC...GATTAGTCAAAGTCCCTACAGTTATCAAATAG
  [4]  1476 ATGTATCGTAAGAGTGCTTTAGAATTAAGAGAT...TGAATGGACTTTTTGACGGAGGAATAGAATAA
  [5]  1467 ATGGGCATAGCACATACTGAATGGGAGTCTGTG...AATTGCTATTAGCAGCTATGCGAGATATGTAA
  ...   ... ...
[892]  3051 ATGCCTTTTTCTTTGAGATCTACATCATTTTGT...TTGTATCCATGGGCTTGAATAGAATCTTTTAA
[893]   303 ATGCTGGCAACAATTAAAAAAATTACTGTGTTG...GAGATTCTCGCCTAGAATGCAAGAAGATATAA
[894]  2637 ATGCGACCTGATCATATGAACTTCTGTTGTCTA...CCCTGGATCTGGGGACCACTTACAGGTTCTAG
[895]    96 ATGTCAAAAAAAAGTAATAATTTACAGACTTTT...ATGAAGAGTTAAGGAAGATTTTTGGATTGTGA
[896]   600 ATGAGCATCAGGGGAGTAGGAGGCAACGGGAAT...AAGCGAACAAGTTGGCTGGCCCTGACGGGGTA
```

We see that the first gene has no start codon and the last gene has no stop codon. This implies that the genes likely connect to each other because the genome is circular and the genome end splits one gene into two. Therefore, the first predicted gene's first codon is not a true start codon and we need to drop this first sequence from our analysis of start codons. We can look at the distribution of predicted start codons with:

```
> table(subseq(dna[-1], 1, 3))
ATG GTG TTG
780  93  22
```

There are the typical three bacterial initiation codons, "ATG", "GTG", and "TTG", and one predicted non-canonical initiation codon: "CTG". Let's take a closer look at genes with non-canonical initiation codons:

```
> w <- which(!subseq(dna, 1, 3) %in% c("ATG", "GTG", "TTG"))
> w
[1] 1
> w <- w[-1] # drop the first sequence because it is a fragment
> w
integer(0)
> dna[w]
DNAStringSet object of length 0
> genes[w]
Genes object of size 0.
```

That worked like a charm, so let's look at the predicted protein sequences by translating the genes:

```
> aa <- ExtractGenes(genes, genome, type="AAStringSet")
> aa
AAStringSet object of length 896:
      width seq
  [1]   392 AAAREIAKRWEQRVRDLQDKGAARKLLNDPLGR...QVEGILRDMLTNGSQTFRDLMRRWNREVDRE*
  [2]    91 MLCKVCRGLSSLIVVLGAINTGILGVTGYKVNL...CLNFLKCCFKKRHGDCCSSKGGYHHHHMDRE*
  [3]   101 MTESYVNKEEIISLAKNAALELEDAHVEEFVTS...DMVTSDFTQEEFLSNVPVSLGGLVKVPTVIK*
  [4]   492 MYRKSALELRDAVVNRELSVTAITEYFYHRIES...ICQVGYSFQEHSQIKQLYPKAVNGLFDGGIE*
  [5]   489 MGIAHTEWESVIGLEVHVELNTESKLFSPARNH...GFLVGQIMKRTEGKAPPKRVNELLLAAMRDM*
  ...   ... ...
[892]  1017 MPFSLRSTSFCFLACLCSYSYGFASSPQVLTPN...HHFGRAYMNYSLDARRRQTAHFVSMGLNRIF*
[893]   101 MLATIKKITVLLLSKRKAGIRIDYCALALDAVE...LDASLESAQVRLAGLMLDYWDGDSRLECKKI*
```

```
[894]   879 MRPDHMNFCCLCAAILSSTAVLFGQDPLGETAL...LHRLQTLLNVSYVLRGQSHSYSLDLGTTYRF*
[895]    32 MSKKSNNLQTFSSRALFHVFQDEELRKIFGL*
[896]   200 MSIRGVGGNGNSRIPSHNGDGSNRRSQNTKGNN...NLDVNEARLMAAYTSECADHLEANKLAGPDGV
```

All of the genes start with a methionine ("M") residue and end with a stop ("*") except the first and last gene because they wrap around the end of the genome. If so inclined, we could easily remove the first and last positions with:

```
> subseq(aa, 2, -2)
AAStringSet object of length 896:
      width seq
  [1]   390 AAREIAKRWEQRVRDLQDKGAARKLLNDPLGRR...QQVEGILRDMLTNGSQTFRDLMRRWNREVDRE
  [2]    89 LCKVCRGLSSLIVVLGAINTGILGVTGYKVNLL...VCLNFLKCCFKKRHGDCCSSKGGYHHHHMDRE
  [3]    99 TESYVNKEEIISLAKNAALELEDAHVEEFVTSM...EDMVTSDFTQEEFLSNVPVSLGGLVKVPTVIK
  [4]   490 YRKSALELRDAVVNRELSVTAITEYFYHRIESH...QICQVGYSFQEHSQIKQLYPKAVNGLFDGGIE
  [5]   487 GIAHTEWESVIGLEVHVELNTESKLFSPARNHF...LGFLVGQIMKRTEGKAPPKRVNELLLAAMRDM
  ...   ... ...
[892]  1015 PFSLRSTSFCFLACLCSYSYGFASSPQVLTPNV...IHHFGRAYMNYSLDARRRQTAHFVSMGLNRIF
[893]    99 LATIKKITVLLLSKRKAGIRIDYCALALDAVEY...QLDASLESAQVRLAGLMLDYWDGDSRLECKKI
[894]   877 RPDHMNFCCLCAAILSSTAVLFGQDPLGETALL...ALHRLQTLLNVSYVLRGQSHSYSLDLGTTYRF
[895]    30 SKKSNNLQTFSSRALFHVFQDEELRKIFGL
[896]   198 SIRGVGGNGNSRIPSHNGDGSNRRSQNTKGNNK...GNLDVNEARLMAAYTSECADHLEANKLAGPDG
```

4.2 Removing genes with too many ambiguities

Genomes sometimes contain ambiguous positions (e.g., "N") within open reading frames. These ambiguities can make an open reading frame look longer than it actually is, giving the illusion of a single gene when none (or more than one) is present. We can easily remove those with more than some magic number (let's say 20 or 5%) of ambiguous positions:

```
> a <- alphabetFrequency(dna, baseOnly=TRUE)
> w <- which(a[, "other"] <= 20 & a[, "other"]/width(dna) <= 5)
> genes <- genes[w]
```

Abracadabra! The genes with many ambiguities have magically disappeared from our gene set.

4.3 Taking a look at the shortest genes

You might think finding short genes (< 90 nucleotides) would require black magic, but `FindGenes` can do it quite well. We can select the subset of short genes and take a look at how repeatedly they were called genes during iteration:

```
> w <- which(width(dna) < 90)
> dna[w]
DNAStringSet object of length 3:
    width seq
[1]    69 ATGAGTCATATAAGTATTCGCAATAGTAGTTATTTATTCGCTAATCCAAATCAAGAAATCGTATTTTGA
[2]    87 ATGAAATTAAAACAAAAGATCGGAATAAAAACAT...GCTCTGGGCCTTGCGAAAAAAACCAAAAAATAG
[3]    66 ATGTTAGTGTGGGTTATGAGACAGCCGCGTTATAACTCGGAAGTAGAGAAAAAACAAGATTTTTAG
> aa[w]
```

```
AAStringSet object of length 3:
    width seq
[1]    23 MSHISIRNSSYLFANPNQEIVF*
[2]    29 MKLKQKIGIKTFGQKKRKALGLAKKTKK*
[3]    22 MLVWVMRQPRYNSEVEKKQDF*
> genes[w]
Genes object of size 3 specifying:
3 protein coding genes from 66 to 87 nucleotides.

      Index Strand  Begin    End TotalScore ... Gene
4606      1      1  44025  44093       9.11 ...    1
9793      1      1  94116  94202       8.08 ...    1
45975     1      1 443090 443155       8.58 ...    1
> genes[w, "End"] - genes[w, "Begin"] + 1 # lengths
 4606  9793 45975
   69    87    66
> genes[w, "FractionReps"]
 4606  9793 45975
    1     1     1
```

We can see from the *FractionReps* column always being 100% that these short genes were repeatedly identified during iteration, suggesting they weren't pulled out of a hat. That's impressive given how short they are!

4.4 Revealing the secrets of gene finding

All of this might seem like hocus pocus, but the predictions made by `FindGenes` are supported by many scores. Some scores are related because they make use of information from the same region relative to the gene boundaries. We can take a look at score correlations with:

```
> pairs(genes[, 5:16], pch=46, col="#00000033", panel=panel.smooth)
```

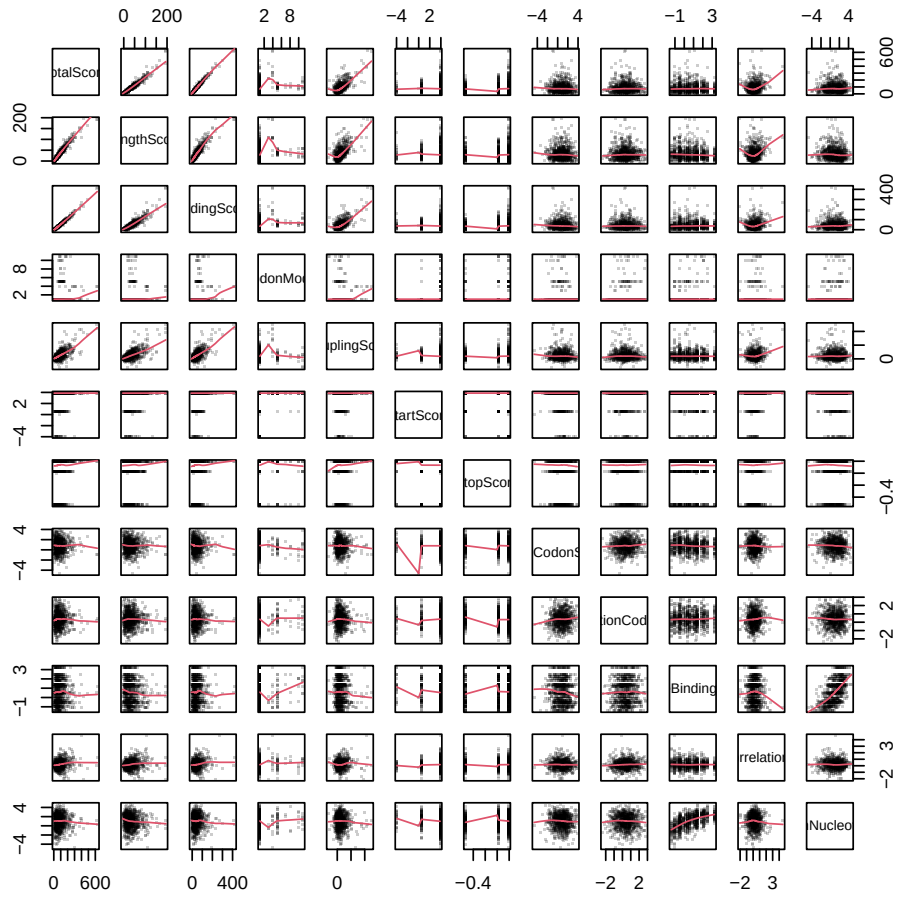


Figure 2: Scatterplot matrix of scores used by `FindGenes` to make gene predictions.

Certainly some of the magic of gene finding is having a lot of scores! We can see that the ribosome binding site score and upstream nucleotide score are the most correlated. This isn't just magical thinking, both scores rely on the same nucleotides immediately upstream of the start codon. The different scores are defined as follows:

1. Upstream signals
   (a) *Ribosome Binding Site Score* - Binding strength and position of the Shine-Delgarno sequence, as well as other motifs in the first bases upstream of the start codon.
   (b) *Upstream Nucleotide Score* - Nucleotides in each position immediately upstream of the start codon.
   (c) *Upstream Motif Score* - K-mer motifs further upstream of the start codon.
2. Start site signals
   (a) *Start Score* - Choice of start codon relative to the background distribution of open reading frames.
   (b) *Folding Score* - Free energy of RNA-RNA folding around the start codon and relative to locations upstream and downstream of the start.
   (c) *Initial Codon Score* - Choice of codons in the first few positions after the start codon.
3. Gene content signals
   (a) *Coding Score* - Usage of codons or pairs of codons within the open reading frame.
   (b) *Codon Model* - Number of the codon model that best fit each open reading frame.
   (c) *Length Score* - Length of the open reading frame relative to the background of lengths expected by chance.
   (d) *Autocorrelation Score* - The degree to which the same or different codons are used sequentially to code for an amino acid.
   (e) *Coupling Score* - Likelihood of observing neighboring amino acids in real proteins.
4. Termination signals
   (a) *Termination Codon Score* - Codon bias immediately before the stop codon.
   (b) *Stop Score* - Choice of stop codon relative to the observed distribution of possible stop codons.

4.5 Taking a closer look at the output

If we have a particular gene of interest, it can sometimes be useful to plot the output of `FindGenes` as the set of all possible open reading frames with the predicted genes highlighted. The `plot` function for a *Genes* object is interactive, so it is possible to pan left and right by setting the *interact* argument equal to `TRUE`. For now we will only look at the beginning of the genome:

```
> plot(orfs, interact=FALSE)
```

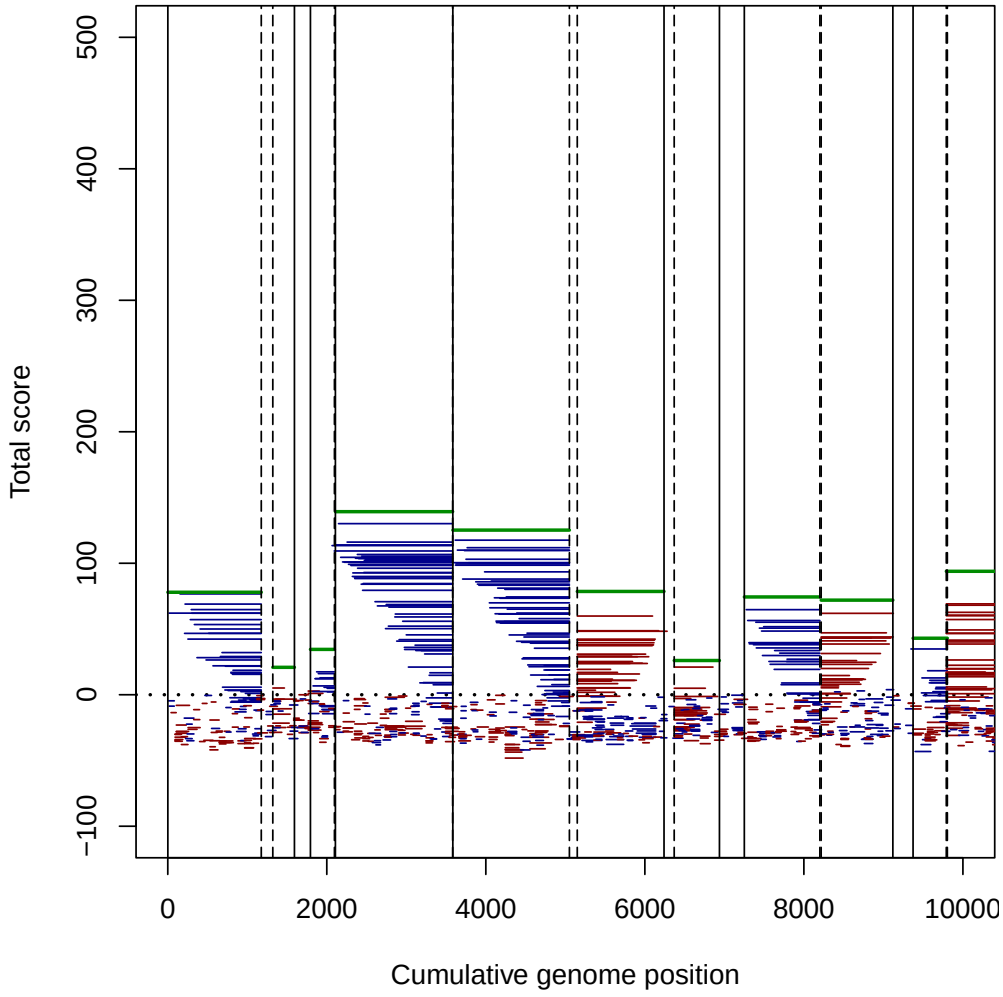


Figure 3: All possible open reading frames (red and blue) with predicted genes highlighted in green.

5 Incorporating non-coding RNAs

5.1 Examining the intergenic spaces

The space between genes sometimes contains promoter sequences, pseudogenes, or non-coding RNA genes that are of interest. We can conjure up the intergenic sequences longer than a given width (e.g., 30 nucleotides) with the commands:

```
> s <- c(1, genes[, "End"] + 1)
> e <- c(genes[, "Begin"] - 1, width(genome))
> w <- which(e - s >= 30)
> intergenic <- unlist(extractAt(genome, IRanges(s[w], e[w])))
> intergenic
DNAStringSet object of length 504:
      width seq                                               names
  [1]   144 ACTGGTATCTACCATAGGTTTG...TATAATCTGAAAGGAAGGCGTT NC_000117.1 Chlam...
  [2]   200 GACGTTTCTCCAACGTAGATGT...TTGACCATGTTTAGGATGGAAG NC_000117.1 Chlam...
  [3]    98 TTTGCAGCATCCTCAAAAAAGG...AGGCTTTATCGCTTTTCCAATA NC_000117.1 Chlam...
  [4]   127 AAAATTATCCCAAAAACAAAAA...ACATAGATTCTAGCACTTCTTA NC_000117.1 Chlam...
  [5]   312 TTTTCCAACTAGGTGTTATGTA...GCTGATTGTATGATGGAGTGGT NC_000117.1 Chlam...
  ...   ... ...
[500]   164 CGCAGGTTAAAAGGGGGATGTT...TTTATCTCTCAGCTTTTGTGTG NC_000117.1 Chlam...
[501]   596 TAAAGATTTTTCTTTTCAGAAG...AAAGAAGATGTCATCAAACAGG NC_000117.1 Chlam...
[502]   174 TCGCCAGGTTTCGAGACAAAGT...AACGCAATGCTAGGCAAGGGAA NC_000117.1 Chlam...
[503]    33 GGGATTCCCCCTAAGAGTCTAAAAGAAGAGGTT               NC_000117.1 Chlam...
[504]   144 AATTAGAAAAATCAGAATTATC...TTTAGTTTTGGGTTGGTTTGTT NC_000117.1 Chlam...
```

Some of these intergenic regions might be similar to each other. We can find related intergenic sequences by clustering those within a certain distance (e.g., 20

```
> intergenic <- c(intergenic, reverseComplement(intergenic))
> names(intergenic) <- c(w, paste(w, "rc", sep="_"))
> clusts <- Clusterize(myXStringSet=intergenic, cutoff=0.3)
Partitioning sequences by 9-mer similarity:

iteration 400 of up to 400 (66.3% coverage)

Time difference of 1.19 secs

Sorting by relatedness within 325 groups:

iteration 29 of up to 29 (100.0% stability)

Time difference of 1.06 secs

Clustering sequences by 9-mer similarity:
================================================================================

Clusters via relatedness sorting: 69.1% (20.9% exclusively)
Clusters via rare 6-mers: 79.1% (30.9% exclusively)
Estimated clustering effectiveness: 99.1%

Time difference of 1.86 secs
```

Since we used `inexact` clustering, the clusters containing the longest sequences will be first. We can look at sequences belonging to the first cluster:

```
> t <- sort(table(clusts$cluster), decreasing=TRUE)
> head(t) # the biggest clusters
120 405  75  82 117 125
  4   4   3   3   3   3
> AlignSeqs(intergenic[clusts$cluster == names(t)[1]], verbose=FALSE)
DNAStringSet object of length 4:
    width seq                                                 names
[1]   128 ----------------------...---------------------- 837
[2]   128 AAGGTTTTTGGTTTGGAGGATCC...---------------------- 52_rc
[3]   128 ----------------------...CTTCATT---------------- 134_rc
[4]   128 ------------------AAC...TTTATTCGCCTAAATAAGTGAGT 297_rc
```

These two long intergenic regions probably contain copies of the ribosomal RNA operon. A signature of non-coding RNAs is that they tend to have higher GC content than expected. We can create a plot of GC content in intergenic regions versus size. Since our genome averages 41% GC content, we can use statistics to add a line for the 95% confidence interval. Only a few intergenic regions have unexpectedly high GC content.

```
> gc <- alphabetFrequency(intergenic, as.prob=TRUE, baseOnly=TRUE)
> gc <- gc[, "G"] + gc[, "C"]
> plot(width(intergenic),
         100*gc,
         xlab="Length (nucleotides)",
         ylab="GC content (%)",
         main="Intergenic regions",
         log="x")
> size <- 10^seq(1, 4, 0.1)
> expected <- 0.413*size
> lines(size, 100*(expected + 1.96*sqrt(expected))/size)
```

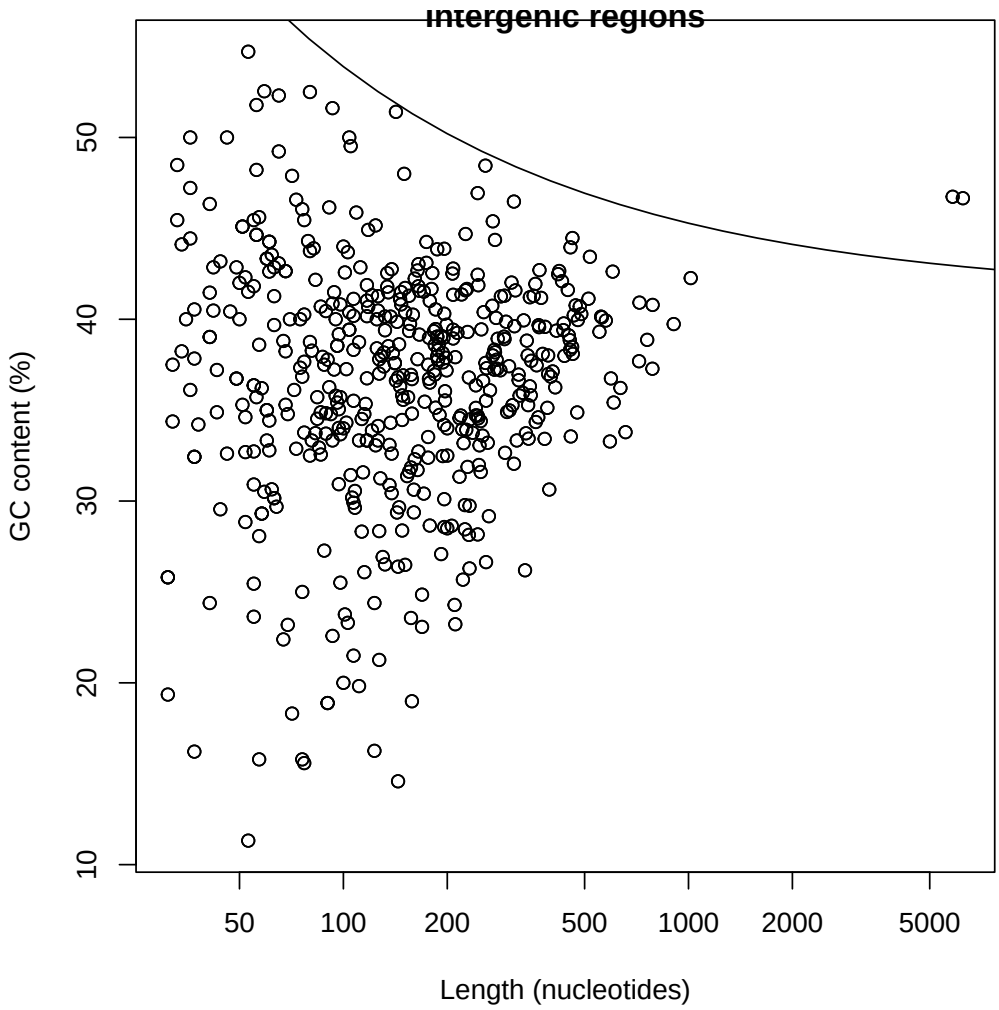


Figure 4: Non-coding RNAs are often found in intergenic regions with unexpectedly high GC content.

5.2 Finding and including non-coding RNAs

The DECIPHER has a separate function, `FindNonCoding`, to find non-coding RNAs in a genome. It searches for predefined models created from multiple sequence alignments of non-coding RNAs. Thankfully DECIPHER includes pre-built models of non-coding RNAs commonly found in bacteria, archaea, and eukarya. Since *C. trachomatis* is a bacterium, we will load the bacterial models. For genomes belonging to organisms from other domains of life, simply replace "Bacteria" with "Archaea" or "Eukarya".

```
> data(NonCodingRNA_Bacteria)
> x <- NonCodingRNA_Bacteria
> names(x)
 [1] "tRNA-Ala"
 [2] "tRNA-Arg"
 [3] "tRNA-Asn"
 [4] "tRNA-Asp"
 [5] "tRNA-Cys"
 [6] "tRNA-Gln"
 [7] "tRNA-Glu"
 [8] "tRNA-Gly"
 [9] "tRNA-His"
[10] "tRNA-Ile"
[11] "tRNA-Leu"
[12] "tRNA-Lys"
[13] "tRNA-Met"
[14] "tRNA-Phe"
[15] "tRNA-Pro"
[16] "tRNA-Ser"
[17] "tRNA-Thr"
[18] "tRNA-Trp"
[19] "tRNA-Tyr"
[20] "tRNA-Val"
[21] "tRNA-Sec"
[22] "rRNA_5S-RF00001"
[23] "rRNA_16S-RF00177"
[24] "rRNA_23S-RF02541"
[25] "tmRNA-RF00023"
[26] "tmRNA_Alpha-RF01849"
[27] "RNase_P_class_A-RF00010"
[28] "RNase_P_class_B-RF00011"
[29] "SsrS-RF00013"
[30] "Intron_Gp_I-RF00028"
[31] "Intron_Gp_II-RF00029"
[32] "SmallSRP-RF00169"
[33] "Cyclic-di-GMP_Riboswitch-RF01051"
[34] "Cyclic-di-AMP_Riboswitch-RF00379"
[35] "T-box_Leader-RF00230"
[36] "Ribosomal_Protein_L10_Leader-RF00557"
[37] "Cobalamin_Riboswitch-RF00174"
[38] "TPP_Riboswitch-RF00059"
[39] "SAM_Riboswitch-RF00162"
[40] "Fluoride_Riboswitch-RF01734"
[41] "FMN_Riboswitch-RF00050"
```

```
[42] "Glycine_Riboswitch-RF00504"
[43] "HEARO-RF02033"
[44] "Flavo_1-RF01705"
[45] "Acido_Lenti_1-RF01687"
[46] "5'_ureB-RF02514"
```

We need to search for these models in our genome before we can incorporate them into our gene calls.

```
> rnas <- FindNonCoding(x, genome)
================================================================================

Time difference of 47 secs
> rnas
Genes object of size 46 specifying:
46 non-coding RNAs from 72 to 2,938 nucleotides.

  Index Strand  Begin    End TotalScore Gene
1     1      1  20663  21082      88.33  -25
2     1      1  42727  42801      69.94   -3
3     1      1  68920  68995      78.62  -15
4     1      0 158662 158736      55.75  -17
5     1      0 158744 158827      52.31  -19
6     1      1 202339 202414      68.64  -10
... with 40 more rows.
```

Tada! `FindNonCoding` outputs an object of class *Genes* just like `FindGenes`. However, non-coding RNAs are denoted with negative numbers in the `Gene` column. Each number corresponds to the model that was identified. We can look at the set of non-coding RNAs that were found in the *C. trachomatis* genome:

```
> annotations <- attr(rnas, "annotations")
> m <- match(rnas[, "Gene"], annotations)
> sort(table(names(annotations)[m]))
RNase_P_class_A-RF00010        SmallSRP-RF00169                tRNA-Asn
                      1                       1                       1
               tRNA-Asp                tRNA-Cys                tRNA-Gln
                      1                       1                       1
               tRNA-Glu                tRNA-His                tRNA-Ile
                      1                       1                       1
               tRNA-Lys                tRNA-Phe                tRNA-Trp
                      1                       1                       1
               tRNA-Tyr           tmRNA-RF00023        rRNA_16S-RF00177
                      1                       1                       2
       rRNA_23S-RF02541         rRNA_5S-RF00001                tRNA-Ala
                      2                       2                       2
               tRNA-Gly                tRNA-Pro                tRNA-Val
                      2                       2                       2
               tRNA-Arg                tRNA-Met                tRNA-Thr
                      3                       3                       3
               tRNA-Ser                tRNA-Leu
                      4                       5
```

There was at least one tRNA gene found for each amino acid, as well as two copies of each ribosomal RNA gene, and the RNaseP and tmRNA genes. Now, we can easily include these non-coding RNAs into our gene calls.

```
> genes <- FindGenes(genome, includeGenes=rnas)
Iter  Models Start Motif  Init  Fold UpsNt  Term   RBS  Auto  Stop Genes
1          1 18.40                 _                                 931
1          1 18.40                 \                                 931
1          1 18.40                 |                                 931
1          1 18.40                 /                                 931
1          1 18.40                 _                                 931
1          1 18.40                 \                                 931
1          1 18.40                 |                                 931
1          1 18.40                 /                                 931
1          1 18.40                 _                                 931
2         12 18.40  0.48  0.91                                       934
3         15 18.38  0.53  1.16  0.40  1.41                           937
4         14 15.60  0.52  1.27  0.45  1.74  0.24                     939
5         12 15.63  0.51  1.25  0.47  1.82  0.25  0.96  0.10  0.07   943
6         12 15.61  0.50  1.25  0.50  1.91  0.25  1.04  0.10  0.07   943
7         11 15.59  0.50  1.26  0.51  1.93  0.25  1.03  0.10  0.07   943
8         11 15.59  0.50  1.26  0.51  1.93  0.25  1.07  0.10  0.07   943

Time difference of 32.03 secs
> genes
Genes object of size 943 specifying:
897 protein coding genes from 66 to 5,361 nucleotides.
46 non-coding RNAs from 72 to 2,938 nucleotides.

  Index Strand Begin  End TotalScore ... Gene
1     1      0     1 1176      78.43 ...    1
2     1      1  1321 1593      21.03 ...    1
3     1      0  1794 2096      34.10 ...    1
4     1      0  2108 3583     139.61 ...    1
5     1      0  3585 5051     125.14 ...    1
6     1      1  5150 6241     106.03 ...    1
... with 937 more rows.
```

That worked like magic! Now the *Genes* object contains both protein coding genes and non-coding RNAs.

6 Annotating the protein coding genes

The magic doesn't have to end there. We can annotate our protein coding genes just like we did with the non-coding RNAs. To master this sleight of hand, we first need a training set of labeled protein sequences. We have exactly that included in DECIPHER for *Chlamydia*.

```
> fas <- system.file("extdata",
         "PlanctobacteriaNamedGenes.fas.gz",
         package="DECIPHER")
> prot <- readAAStringSet(fas)
> prot
```

```
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...
> head(names(prot))
[1] "A0A0H3MDW1|Root;NoEC;chxR"        "O84395|Root;2;6;1;83;dapL"
[3] "Q9Z6M7|Root;4;1;1;19;aaxB"        "Q46222|Root;2;4;99;Multiple;waaA"
[5] "Q59321|Root;6;3;4;2;pyrG"         "P0C0Z7|Root;NoEC;groL"
```

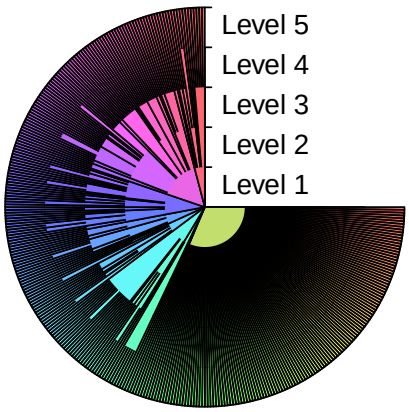
The training sequences are named by their enzyme commission (EC) number and three or four-letter gene name. The process of training a classifier is described elsewhere. For now, let's jump straight to the solution:

```
> trainingSet <- LearnTaxa(train=prot,
         taxonomy=names(prot),
         maxChildren=1)
================================================================================

Time difference of 1.43 secs
> trainingSet
  A training set of class 'Taxa'
   * K-mer size: 9
   * Number of rank levels: 7
   * Total number of sequences: 2497
   * Number of groups: 532
   * Number of problem groups: 0
   * Number of problem sequences: 0
```

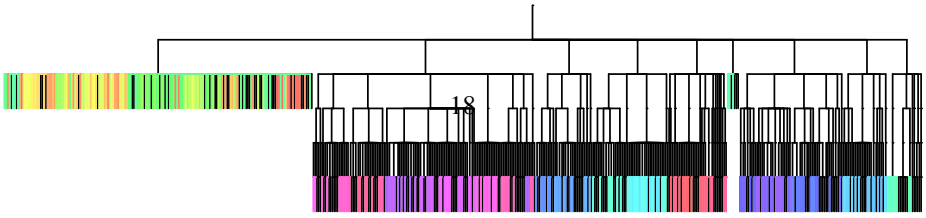
Now we need to take the proteins we just found with `FindGenes` and classify them using our classifier. Finally, we can assign each of our protein coding sequences to a classification in the *Genes* object:

```
> annotations <- sapply(ids, function(x) paste(x$taxon[-1], collapse="; "))
> u_annotations <- unique(annotations)
> genes[w, "Gene"] <- match(annotations, u_annotations) + 1L
> attr(genes, "annotations") <- c(attr(genes, "annotations"),
         setNames(seq_along(u_annotations) + 1L,
                  u_annotations))
> genes
Genes object of size 943 specifying:
897 protein coding genes from 66 to 5,361 nucleotides.
46 non-coding RNAs from 72 to 2,938 nucleotides.

  Index Strand Begin  End TotalScore ... Gene
```

```
> w <- which(genes[, "Gene"] > 0)
> aa <- ExtractGenes(genes[w], genome, type="AAStringSet")
> ids <- IdTaxa(aa,
         trainingSet,
         fullLength=0.99,
         threshold=50,
         processors=1)
================================================================================

Time difference of 3.51 secs
> ids
  A test set of class 'Taxa' with length 897
      confidence taxon
  [1]         9% Root; unclassified_Root
  [2]         2% Root; unclassified_Root
  [3]       100% Root; 6; 3; 5; -; gatC
  [4]       100% Root; 6; 3; 5; 7; gatA
  [5]        99% Root; 6; 3; 5; -; gatB
  ...         ... ...
[893]       100% Root; NoEC; pmpH
[894]         6% Root; unclassified_Root
[895]        99% Root; NoEC; pmpI
[896]         0% Root; unclassified_Root
[897]        11% Root; unclassified_Root
> plot(trainingSet, ids[grep("unclassified", ids, invert=TRUE)])
```



Distribution on taxonomic tree



```
1     1      0     1 1176      78.43 ...    2
2     1      1  1321 1593      21.03 ...    2
3     1      0  1794 2096      34.10 ...    3
4     1      0  2108 3583     139.61 ...    4
5     1      0  3585 5051     125.14 ...    5
6     1      1  5150 6241     106.03 ...    2
... with 937 more rows.
```

With a little wizardry, we can now look at the top annotations in the genome.

```
> annotations <- attr(genes, "annotations")
> m <- match(genes[, "Gene"], annotations)
> head(sort(table(names(annotations)[m]), decreasing=TRUE))
unclassified_Root          tRNA-Leu          tRNA-Ser          tRNA-Arg
              567                 5                 4                 3
         tRNA-Met          tRNA-Thr
                3                 3
```

7 Guaranteeing repeatability

`FindGenes` sometimes uses random sampling to increase speed of the algorithm. For this reason, gene predictions may change slightly if the prediction process is repeated with the same inputs. For some applications this randomness is undesirable, and it can easily be avoided by setting the random seed before using `FindGenes`. The process of setting and then unsetting the seed in R is straightforward:

```
> set.seed(123) # choose a whole number as the random seed
> # then make gene predictions with FindGenes (not shown)
> set.seed(NULL) # return to the original state by unsetting the seed
```

8 Exporting the output

The genes can be exported in a variety of formats, including as a FASTA file with `writeXStringSet`, GenBank (gbk) or general feature format (gff) file with `WriteGenes`, or delimited file formats (e.g., csv, tab, etc.) with `write.table`.

Now that you know the tricks of the trade, you can work your own magic to find new genes!

9 Session Information

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

- R version 4.3.0 RC (2023-04-13 r84269 ucrt), `x86_64-w64-mingw32`
- Running under: `Windows Server 2022 x64 (build 20348)`
- Matrix products: default
- Base packages: base, datasets, grDevices, graphics, methods, parallel, stats, stats4, utils
- Other packages: BiocGenerics 0.46.0, Biostrings 2.68.0, DECIPHER 2.28.0, GenomeInfoDb 1.36.0, IRanges 2.34.0, RSQLite 2.3.1, S4Vectors 0.38.0, XVector 0.40.0

- Loaded via a namespace (and not attached): DBI 1.1.3, GenomeInfoDbData 1.2.10, KernSmooth 2.23-20, RCurl 1.98-1.12, bit 4.0.5, bit64 4.0.5, bitops 1.0-7, blob 1.2.4, cachem 1.0.7, cli 3.6.1, compiler 4.3.0, crayon 1.5.2, fastmap 1.1.1, memoise 2.0.1, pkgconfig 2.0.3, rlang 1.1.0, tools 4.3.0, vctrs 0.6.2, zlibbioc 1.46.0