Sequence Alignment of Short Read Data using Biostrings

Patrick Aboyoun
Fred Hutchinson Cancer Research Center
Seattle, WA 98008

18 November 2009

Contents

1 Introduction

While most researchers use sequence alignment software like ELAND, MAQ, and Bowtie to perform the bulk of short read mappings to a target genome, BioConductor contains a number of string matching/pairwise alignment tools in the Biostrings package that can be invaluable when answering complex scientific questions. These tools are naturally divided into five groups (`matchPDict`, `vmatchPattern`, `pairwiseAlignment`, `matchPWM`, and OTHER) that contain the following functions:

`matchPDict` : `matchPDict`, `countPDict`, `whichPDict`, `vmatchPDict`, `vcountPDict`, `vwhichPDict`

`vmatchPattern` : `matchPattern`, `countPattern`, `vmatchPattern`, `vcountPattern`, `neditStartingAt`, `neditEndingAt`, `isMatchingStartingAt`, `isMatchingEndingAt`, `which.isMatchingStartingAt`, `which.isMatchingEndingA`

`pairwiseAlignment` : `pairwiseAlignment`, `stringDist`

`matchPWM` : `matchPWM`, `countPWM`

OTHER : `matchLRPatterns` (finds singleton paired-end matches), `trimLRPatterns` (trims left and/or right flanking patterns), `matchProbePair` (finds theoretical amplicons),

For detailed information on any of these functions, use `help(≪ function name ≫)` from within R.

Of the functions listed above, the `pairwiseAlignment` function stands out because it creates the most complex output object. When producing more than just the alignment score, this output (either a *PairwiseAlignedXStringSet* or a *PairwiseAlignedFixedSubject*) can be processed by a number of helper functions including those listed in Tables 1 & 2 below.

Function	Description
`[`	Extracts the specified elements of the alignment object
`alphabet`	Extracts the allowable characters in the original strings
`compareStrings`	Creates character string mashups of the alignments
`deletion`	Extracts the locations of the gaps inserted into the pattern for the alignments
`length`	Extracts the number of patterns aligned
`mismatchTable`	Creates a table for the mismatching positions
`nchar`	Computes the length of "gapped" substrings
`nedit`	Computes the Levenshtein edit distance of the alignments
`indel`	Extracts the locations of the insertion & deletion gaps in the alignments
`insertion`	Extracts the locations of the gaps inserted into the subject for the alignments
`nindel`	Computes the number of insertions & deletions in the alignments
`nmatch`	Computes the number of matching characters in the alignments
`nmismatch`	Computes the number of mismatching characters in the alignments
`pattern`, `subject`	Extracts the aligned pattern/subject
`pid`	Computes the percent sequence identity
`rep`	Replicates the elements of the alignment object
`score`	Extracts the pairwise sequence alignment scores
`type`	Extracts the type of pairwise sequence alignment

Table 1: Functions for *PairwiseAlignedXStringSet* and *PairwiseAlignmentFixedSubject* objects.

Table 3 shows the relative strengths and weaknesses of the `matchPDict`, `vmatchPattern`, and `pairwiseAlignment` functional families and hints at how they can be used in tandem to answer multi-faceted questions.

The BSgenome package provides a framework for representing and operating on whole genomes, including methods that perform `vmatchPattern`, `vcountPattern`, `matchPWM`, and `countPWM` over all the chromosomes. The remaining functions mentioned above can be incorporated into `bsapply` looping operations (see `help("bsapply")` for more details).

Function	Description
`aligned`	Creates an *XStringSet* containing either "filled-with-gaps" or degapped aligned strings
`as.character`	Creates a character vector version of `aligned`
`as.matrix`	Creates an "exploded" character matrix version of `aligned`
`consensusMatrix`	Computes a consensus matrix for the alignments
`consensusString`	Creates the string based on a 50% + 1 vote from the consensus matrix
`coverage`	Computes the alignment coverage along the subject
`mismatchSummary`	Summarizes the information of the `mismatchTable`
`summary`	Summarizes a pairwise sequence alignment
`toString`	Creates a concatenated string version of `aligned`
`Views`	Creates an *XStringViews* representing the aligned region along the subject

Table 2: Additional functions for *PairwiseAlignedFixedSubject* objects.

`matchPDict`	`vmatchPattern`	`pairwiseAlignment`
Utilizes a fast string matching algorithm for multiple patterns.	Uses a fast string matching algorithm for multiple subjects.	Not practical for long strings.
Finds all occurrences with up to the specified # of mismatches.	Finds all occurrences with up to the specified # of mismatches / edit distance.	Returns only one of the best scoring alignment.
Supports removal of repeat masked regions.	Supports removal of repeat masked regions.	Cannot handle masked genomes.
Produces limited output: # of times a pattern matches and where they occur.	Produces limited output: # of times a pattern matches and where they occur.	Allows various summaries of alignments.
Does not support insertions or deletions.	Supports insertions and deletions.	Supports insertions and deletions.
Uses a mismatch penalty scheme.	Uses a mismatch penalty or edit distance penalty scheme.	Provides a flexible alignment framework, including quality-based scoring.

Table 3: Comparisons of string matching/alignment methods.

2 Setup

This lab is designed as series of hands-on exercises where the students follow along with the instructor. The first exercise is to load the required packages:

Exercise 1
Start an R session and use the `library` *function to load the ShortRead software package and BSgenome.Mmusculus.UCSC.mm9 genome package along with its dependencies using the following commands:*

```
> suppressMessages(library("ShortRead"))
> library("BSgenome.Mmusculus.UCSC.mm9")
```

Exercise 2
Use the `packageDescription` *function to confirm that the loaded version of the BSgenome package is >= 1.14.1, the Biostrings package is >= 2.14.5 and the IRanges package is >= 1.4.6.*

```
> packageDescription("BSgenome")$Version
```

```
[1] "1.14.1"

> packageDescription("Biostrings")$Version

[1] "2.14.5"

> packageDescription("IRanges")$Version

[1] "1.4.7"
```

Seek assistance from one of the course assistants if you need help updating any of your BioConductor packages.

This lab also requires you have access to sample data.

Exercise 3
The data for this lab is contained in the SeqBasicsTutorial package, which is a custom package prepared for this course and not available on `http://bioconductor.org`. If you don't have the package installed on your system, notify one of the course assistants.

```
> library(SeqBasicsTutorial)
```

3 Pattern and PWM Matching along a Genome

Some ChIP-seq experiments involve finding alignment coverage relative to the locations of known motif signatures or high position weight matrix scored regions. For example, if we believe our sequencing experiment captures CTCF binding, where CTCF is a transcription factor that is a known insulator, we can use the `vcountPattern` and `vmatchPattern` functions to count and find the locations for a candidate CTCF motif, say `"GCCACCAGGGGGCGC"`, in a mouse model.

```
> motifCounts <- vcountPattern("GCCACCAGGGGGCGC", Mmusculus)
> class(motifCounts)
[1] "data.frame"
> head(motifCounts)
  seqname strand count
1    chr1      +     0
2    chr1      -     0
3    chr2      +     1
4    chr2      -     1
5    chr3      +     1
6    chr3      -     3
> sum(motifCounts[,"count"])
[1] 47
> motifLocs <- vmatchPattern("GCCACCAGGGGGCGC", Mmusculus)
> class(motifLocs)
[1] "RangedData"
attr(,"package")
[1] "IRanges"
> motifLocs
RangedData with 47 rows and 2 value columns across 35 spaces
         space                 ranges |   strand          string
   <character>              <IRanges> | <factor>  <DNAStringSet>
1         chr2 [119076727, 119076741] |        + GCCACCAGGGGGCGC
2         chr2 [180082012, 180082026] |        - GCGCCCCCTGGTGGC
```

```
3         chr3 [ 88049951,  88049965] |        + GCCACCAGGGGGCGC
4         chr3 [ 19594062,  19594076] |        - GCGCCCCCTGGTGGC
5         chr3 [ 33817880,  33817894] |        - GCGCCCCCTGGTGGC
6         chr3 [ 96512487,  96512501] |        - GCGCCCCCTGGTGGC
7         chr4 [ 43938863,  43938877] |        + GCCACCAGGGGGCGC
8         chr4 [128256593, 128256607] |        + GCCACCAGGGGGCGC
9         chr4 [101337165, 101337179] |        - GCGCCCCCTGGTGGC
10        chr5 [114502306, 114502320] |        + GCCACCAGGGGGCGC
...
<37 more rows>
```

Before proceeding lets take some time to examine the output of these two activities. The motif counts along the genome are stored in a *data.frame* object. *data.frame* are the standard table objects within R. If you would like to save these counts in a flat file, such as a comma-separated file, you can use the `write.table` supplied in standard R. The motif locations along the genome are stored in a *RangedData* object. This tabular class is defined in the IRanges package and can be exported to a UCSC bed format using the `export` function from the rtracklayer package.

```
> write.table(motifCounts, file = "CTCFMotifCounts.csv", sep = ",")
> library(rtracklayer)
> export(motifLocs, con = "CTCFMotifLocs.bed")
```

More can be learned about these two classes using `help(data.frame)` and `help(RangedData)` respectively.

Returning to the analysis, we can find locations along the mouse genome based on the scoring from a position weight matrix. The SeqBasicsTutorial package contains `ctcfPWM`, which is a PWM for the CTCF transcription factor. (Note: These two operations can take 5-10 minutes.)

```
> data(ctcfPWM)
> pwmCounts <- countPWM(ctcfPWM, Mmusculus, min.score = "85%")
> head(pwmCounts)
  seqname strand count
1    chr1      +  1333
2    chr1      -  1379
3    chr2      +  1659
4    chr2      -  1705
5    chr3      +  1052
6    chr3      -   938
> sum(pwmCounts[,"count"])
[1] 45669
> pwmLocs <- matchPWM(ctcfPWM, Mmusculus, min.score = "85%")
> pwmLocs
RangedData with 45669 rows and 2 value columns across 35 spaces
         space             ranges |   strand               string
   <character>          <IRanges> | <factor>       <DNAStringSet>
1         chr1 [3764256, 3764275] |        + GCAGCCAGGAGGAGGCTCTG
2         chr1 [4506813, 4506832] |        + GTTGCCAATAGGTGGCGCTA
3         chr1 [4760136, 4760155] |        + TTGGCCACCAGGGGGCAGTC
4         chr1 [5313881, 5313900] |        + CAGGCCACCAGGGGTCAGCT
5         chr1 [5659199, 5659218] |        + GGAGCCAACAGGGGGCAGGA
6         chr1 [5857811, 5857830] |        + AAGTCCAGCAGAGGGCACAT
7         chr1 [6073714, 6073733] |        + GAGACCAGAAGAGGGCACCA
8         chr1 [6152483, 6152502] |        + TGTGCCAGAAGAGGGCATCA
```

```
9         chr1 [6372953, 6372972] |        + CTCGCCAGGAGGTGGCTCTC
10        chr1 [6400345, 6400364] |        + GGGGCCAGAAGAGGGCACCA
...
<45659 more rows>
```

Given the ambiguity in CTCF binding, it is no surprise that there are many more sights found via PWM than using an unambiguous string.

4 Finding Possible Contaminants in the Short Reads

The raw base-called sequences that are produced by high-throughput sequencing technologies such as Solexa (Illumina), 454 (Roche), SOLiD (Applied Biosystems), and Helicos tend to contain experiment-related contaminants such as adapters and PCR primers as well as "phantom" sequences such as poly As. The `countPDict`, `vcountPattern`, and `pairwiseAlignment` functions from the Biostrings package allow for the discovery of these troublesome sequences.

These raw base-called sequences can be read with functions like the `readXStringColumns` function and processed with functions like `tables`, which find the most common sequences, from the ShortRead package. While this course will be using pre-processed data for this exercise, the code to find the top short reads looks something like:

```
> library(ShortRead)
> sp <- list(experiment1 = SolexaPath(file.path("path", "to", "experiment1")),
+     experiment2 = SolexaPath(file.path("path", "to", "experiment2")))
> patSeq <- paste("s_", 1:8, "_.*_seq.txt", sep = "")
> names(patSeq) <- paste("lane", 1:8, sep = "")
> topReads <- lapply(structure(seq_len(length(sp)), names = names(sp)),
+     function(i) {
+         print(experimentPath(sp[[i]]))
+         do.call(SplitDataFrameList, lapply(structure(seq_len(length(patSeq)),
+             names = names(patSeq)), function(j, n = 1000) {
+             cat("Reading", patSeq[[j]], "...")
+             x <- tables(readXStringColumns(baseCallPath(sp[[i]]),
+                 pattern = patSeq[[j]], colClasses = c(rep(list(NULL),
+                   4), list("DNAString")))[[1]], n = n)[["top"]]
+             names(x) <- chartr("-", "N", names(x))
+             cat("done.\n")
+             DataFrame(read = DNAStringSet(names(x)), count = unname(x))
+         }))
+     })
```

Exercise 4
Use the `data` function to load the `topReads` object from the SeqBasicsTutorial package.

```
> data(topReads)
```

Exercise 5
Use the `class` function to find the class of the `topReads` object.

```
> class(topReads)

[1] "list"
```

Exercise 6
The `topReads` *object is a list of CompressedSplitDataFrameList objects. Extract the data for experiment 1, lane 1 to find out its content.*

```
> topReads[["experiment1"]][["lane1"]]

DataFrame with 1000 rows and 2 columns
                                    read     count
                            <DNAStringSet> <integer>
1  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA     81237
2  GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA     62784
3  GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTAGAT     57519
4  GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTTGAT     16286
5  GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGGAT     11849
6  GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTATAT     10927
7  ANNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN      8933
8  GNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN      7850
9  TNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN      6652
10 CNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN      6270
...
<990 more rows>
```

Exercise 7
Extract the most common read in each of the 8 lanes for both experiments by nesting an `lapply` *function call in an* `sapply` *function call.*

```
> sapply(topReads, lapply, function(x) as.character(x[["read"]])[1])

      experiment1
lane1 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
lane2 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
lane3 "GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA"
lane4 "GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA"
lane5 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
lane6 "GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA"
lane7 "GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA"
lane8 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
      experiment2
lane1 "GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA"
lane2 "GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA"
lane3 "GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA"
lane4 "GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA"
lane5 "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
lane6 "GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA"
lane7 "GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA"
lane8 "GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA"
```

The `topReads` pre-processed data, loaded in the previous exercise, are in a *list* of *CompressedSplitDataFrameList* objects that represent the read and its corresponding number of occurrences. At a high level, the *list* elements represent two Solexa experiments and the *CompressedSplitDataFrameList* elements representing the 8 lanes of a Solexa run. In both of these experiments, lanes {1-4, 6-8} contain mouse-related experimental data and lane 5 contains data from bacteriophage ϕX174.

The `sapply` function call in the above example, which extracts the most prevalent sequence in each of the lanes, shows that the top read is either `GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA` or all As. Given that

the former sequence is the 33 base pairs of Solexa's genomic DNA/ChIP-seq adapter plus 3 As and the latter sequence of 36 As, it would appear that As are called when there is little information about a particular base.

Finding Poly N Sequences

When data are acquired through the ShortRead package, poly N sequences can be removed using the `polynFilter` function. Since we are operating on pre-processed data, we will have to remove poly N sequences using more rudimentary tools.

Exercise 8
Use the following steps to find the top sequences with with at least 34 nucleotides of a single type (A, C, T, G):

1. *Extract the named vector corresponding to the top sequence counts for experiment 1, lane 1.*
2. *Use the* `alphabetFrequency` *function to find the alphabet frequencies of the reads.*
3. *Use the parallel max,* `pmax`*, function to find the maximum number of occurrences for each of the four bases.*
4. *Create a DNAStringSet whose elements contain at least 34 bases of a single type.*

```
> lane1.1TopReads <- topReads[["experiment1"]][["lane1"]]
> alphabetCounts <- alphabetFrequency(lane1.1TopReads[["read"]],
+     baseOnly = TRUE)
> lane1.1MaxLetter <- pmax(alphabetCounts[, "A"], alphabetCounts[,
+     "C"], alphabetCounts[, "G"], alphabetCounts[, "T"])
> lane1.1PolySingles <- lane1.1TopReads[["read"]][lane1.1MaxLetter >=
+     34]
> length(lane1.1PolySingles)

[1] 115

> head(lane1.1PolySingles)

  A DNAStringSet instance of length 6
    width seq
[1]    36 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[2]    36 CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
[3]    36 AAAAAAAAAAAAAAAAAACAAAAAAAAAAAAAAA
[4]    36 AAAAAAAAAAAAAAAAAACAAAAAAAAAAAAAAA
[5]    36 AAAAAAAAAAAAAAAAAAGAAAAAAAAAAAAAAA
[6]    36 AAAAAAAAAAAAAAAAAATAAAAAAAAAAAAAAA
```

Finding Adapter-Like Sequences

While the Solexa's adapter is known not to map to the mouse genome,

Exercise 9
Show that Solexa's DNA/ChIP-seq adapter doesn't map to the mouse genome by using the `vcountPattern` *function.*

```
> adapter <- DNAString("GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTG")
> adapterCounts <- vcountPattern(adapter, Mmusculus)
> head(adapterCounts)

  seqname strand count
1    chr1      +     0
2    chr1      -     0
3    chr2      +     0
4    chr2      -     0
5    chr3      +     0
6    chr3      -     0

> sum(adapterCounts[, "count"])

[1] 0
```

repeated sequencing of the adapter is a great inefficiency within an experiment. These adapter-like sequences can distort quality assurance of the Solexa data and removing them upstream can help prevent distortions in downstream QA conclusions.

Exercise 10
Use the following steps to find the adapter-like sequences within the top reads:

1. *Create a DNAStringSet object containing the distinct reads by first extracting the top read sequences through nested* `lapply` *operations, then unlisting the result using the* `unlist` *function, then using the* `unique` *function to find the distinct set of reads, and then using the* `sort` *function to sort the sequences in alphabetical order.*

2. *Use the* `isMatchingAt` *function to find the adapter-like sequences.*

3. *Obtain the subset of adapter-like sequences.*

```
> distinctReads <- DNAStringSet(sort(unique(unlist(lapply(topReads,
+     lapply, function(x) as.character(x[["read"]])), use.names = FALSE))))
> whichAdapters <- isMatchingAt(adapter, distinctReads, max.mismatch = 4,
+     with.indels = TRUE)
> adapterReads <- distinctReads[whichAdapters]
> length(adapterReads)

[1] 819

> head(adapterReads)

  A DNAStringSet instance of length 6
    width seq
[1]    36 AATCGGAAGAGCTCGTATGCCGTCTTCTGCTTAGAA
[2]    36 AATCGGAAGAGCTCGTATGCCGTCTTCTGCTTAGAT
[3]    36 AATCGGAAGAGCTCGTATGCCGTCTTCTGCTTATAT
[4]    36 AATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
[5]    36 AATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGGAT
[6]    36 AATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGTAA
```

As the results above show, Solexa's 33-mer adapter is closely related to 819 distinct short reads from the top reads lists.

Exercise 11
Use the following steps to find the number of distinct adapter-like reads and the total number of these reads in each of the 8 lanes for the two experiments:

1. *Use nested* `lapply` *function calls to extract the adapter-like sequences from each of the Solexa lanes.*
2. *Use nested* `sapply` *function calls to get the number of distinct adapter-like sequences.*
3. *Use nested* `sapply` *function calls to get the total number of adapter-like sequences.*

```
> topAdapterReads <- lapply(topReads, lapply, function(x) x[x[["read"]] %in%
+     adapterReads, ])
> sapply(topAdapterReads, sapply, nrow)
```

```
      experiment1 experiment2
lane1         500         226
lane2         303         235
lane3         462         323
lane4         547         305
lane5           0           0
lane6         464         275
lane7         516         284
lane8         343         206
```

```
> sapply(topAdapterReads, sapply, function(x) sum(x[["count"]]))
```

```
      experiment1 experiment2
lane1      265463      158678
lane2      225519      178534
lane3      308251      303996
lane4      456932      290159
lane5           0           0
lane6      343988      255142
lane7      360014      252049
lane8      233244      177058
```

These adapter-like sequences are not wholely without value because they can provide some insight in where base call errors are most likely to occur for a particular sequence.

Exercise 12
Find the distinct sequences from lane 1 of experiment 1 and their associated counts.

```
> lane1.1AdapterCounts <- topAdapterReads[["experiment1"]][["lane1"]][["count"]]
> lane1.1AdapterReads <- topAdapterReads[["experiment1"]][["lane1"]][["read"]]
> length(lane1.1AdapterReads)
```

```
[1] 500
```

```
> head(lane1.1AdapterReads)
```

```
  A DNAStringSet instance of length 6
    width seq
[1]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
[2]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTAGAT
[3]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTTGAT
[4]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGGAT
[5]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTATAT
[6]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTAGAA
```

Exercise 13
Use the `pairwiseAlignment` *function to fit the pairwise alignments of the adapter-like sequences against the adapter then summarize the results using the* `summary` *function.*

```
> lane1.1AdapterAligns <- pairwiseAlignment(lane1.1AdapterReads,
+     adapter, type = "local-global")
> summary(lane1.1AdapterAligns, weight = lane1.1AdapterCounts)

Local-Global Fixed Subject Pairwise Alignment
Number of Alignments:  265463

Scores:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  27.75   57.52   57.52   59.09   65.40   65.40

Number of matches:
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  30.00   32.00   32.00   32.27   33.00   33.00

Top 10 Mismatch Counts:
   SubjectPosition Subject Pattern  Count Probability
1               33       G       A 106988 0.403024150
2               33       G       T  41812 0.157505942
3               20       C       A  12558 0.047306028
4               33       G       C   7298 0.027491590
5               29       G       T   5686 0.021419181
6               20       C       N   2038 0.007677153
7               20       C       T   1996 0.007518939
8               20       C       G   1595 0.006008370
9               14       C       A   1487 0.005601534
10              14       C       T    902 0.003397837
```

Finding Over-Represented Sequences

Another potential source of data contamination is over-represented sequences. These sequences can be found by clustering the short reads.

Exercise 14
First find the distinct sequences from lane 1 of experiment 2 and their associated counts.

```
> lane2.1TopCounts <- topReads[["experiment2"]][["lane1"]][["count"]]
> lane2.1TopReads <- topReads[["experiment2"]][["lane1"]][["read"]]
> length(lane2.1TopReads)

[1] 1000

> head(lane2.1TopReads)

  A DNAStringSet instance of length 6
    width seq
[1]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGAAA
[2]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTAGAT
[3]    36 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
[4]    36 ANNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
[5]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTGGAT
[6]    36 GATCGGAAGAGCTCGTATGCCGTCTTCTGCTTTGAT
```

Exercise 15
Then use the `stringDist` *function to generate the Levenshtein's edit distance amongst the reads, generate nearest-neighbor-based clustering using the* `hclust` *function, and classify the reads into clusters using the* `cutree` *function.*

```
> lane2.1Clust <- hclust(stringDist(lane2.1TopReads), method = "single")
> lane2.1Groups <- cutree(lane2.1Clust, h = 2)
> head(sort(table(lane2.1Groups), decreasing = TRUE))

lane2.1Groups
  1   9   8   3   2  10
226 200 197 161  34  27
```

The example above produces four interesting short read clusters: one representing poly As, one representing Solexa's adapter, and the remaining two coming from an unknown origin.

Exercise 16
Create a set of interesting sequences of unknown origin by using the `intersect` *function to find intersection of one of the interesting clusters with the reverse complement of the other interesting cluster.*

```
> head(reverseComplement(lane2.1TopReads[lane2.1Groups == 9]))

  A DNAStringSet instance of length 6
    width seq
[1]    36 AAATGAGAAATACACACTTTAGGACGTGAAATATGG
[2]    36 AATGAGAAATACACACTTTAGGACGTGAAATATGGC
[3]    36 TGAAAATCACGGAAAATGAGAAATACACACTTTAGG
[4]    36 AGAAATACACACTTTAGGACGTGAAATATGGCGAGG
[5]    36 AATATGGCAAGAAAACTGAAAATCATGGAAAATGAG
[6]    36 AAAATCACGGAAAATGAGAAATACACACTTTAGGAC

> head(lane2.1TopReads[lane2.1Groups == 8])

  A DNAStringSet instance of length 6
    width seq
[1]    36 ACTGAAAATCACGGAAAATGAGAAATACACACTTTA
[2]    36 AAACATCCACTTGACGACTTGAAAAATGACGAAATC
[3]    36 TAGGACGTGGAATATGGCAAGAAAACTGAAAATCAT
[4]    36 GGAATATGGCAAGAAAACTGAAAATCATGGAAAATG
[5]    36 GTAGGACGTGGAATATGGCAAGAAAACTGAAAATCA
[6]    36 TGAAAATCACGGAAAATGAGAAATACACACTTTAGG

> unknownSeqs <- intersect(reverseComplement(lane2.1TopReads[lane2.1Groups ==
+     9]), lane2.1TopReads[lane2.1Groups == 8])
> length(unknownSeqs)

[1] 155

> head(unknownSeqs)
```

```
  A DNAStringSet instance of length 6
    width seq
[1]    36 AAATGAGAAATACACACTTTAGGACGTGAAATATGG
[2]    36 AATGAGAAATACACACTTTAGGACGTGAAATATGGC
[3]    36 TGAAAATCACGGAAAATGAGAAATACACACTTTAGG
[4]    36 AGAAATACACACTTTAGGACGTGAAATATGGCGAGG
[5]    36 AATATGGCAAGAAAACTGAAAATCATGGAAAATGAG
[6]    36 AAAATCACGGAAAATGAGAAATACACACTTTAGGAC
```

Exercise 17
Create a set of interesting sequences and associated counts based upon the intersection created above.

```
> unknownCounts <- lane2.1TopCounts[match(unknownSeqs, lane2.1TopReads)] +
+     lane2.1TopCounts[match(reverseComplement(unknownSeqs), lane2.1TopReads)]
> unknownSeqs <- unknownSeqs[order(unknownCounts, decreasing = TRUE)]
> unknownCounts <- unknownCounts[order(unknownCounts, decreasing = TRUE)]
> length(unknownCounts)

[1] 155

> head(unknownCounts)

[1] 387 375 358 357 354 345
```

These sequences of unknown origin may be related and could potential assemble into a more informative larger sequence. This assembly can be performed using functions from the Biostrings package by first finding a starter, or seeding, sequences that can be grown using pairwise alignments of the starter sequences and the remaining sequences.

Exercise 18
Use the following step to find a starter or seed sequence to use in an assembly process by finding the distinct sequence that closest related to the set of unknown sequences:

1. *Use the* `stringDist` *function to find the number of matches amongst the reads using an overlap alignment with a scoring scheme of* `{match = 1, mismatch = -Inf, gapExtension = -Inf}` *then convert the results into a matrix and loop over the rows to count how many times each distinct read overlap with other distinct reads at least 24 bases in the 36 bases reads.*

2. *Choose the distinct sequence with the most similar distinct sequences using the metric developed in the previous step.*

```
> submat <- nucleotideSubstitutionMatrix(match = 1, mismatch = -Inf)
> whichStarter <- which.max(apply(as.matrix(stringDist(unknownSeqs,
+     method = "substitutionMatrix", substitutionMatrix = submat,
+     gapExtension = -Inf, type = "overlap")), 1, function(x) sum(x >=
+     24)))
> starterSeq <- unknownSeqs[[whichStarter]]
> starterSeq

  36-letter "DNAString" instance
seq: TGAAAATCACGGAAAATGAGAAATACACACTTTAGG
```

Exercise 19
Use the `pairwiseAlignment` *function to generate the pairwise alignments of all sequences against the starter sequence.*

```
> starterAlign <- pairwiseAlignment(unknownSeqs, starterSeq, substitutionMatrix = submat,
+     gapExtension = -Inf, type = "overlap")
```

Exercise 20
Assemble a sequence by using the starter sequence created above and the set of interesting sequences you found. The first step is to find which alignments are in the "prefix" of the starter sequence. These are the sequences that overlap to the left of the start sequence.

```
> whichInPrefix <- (score(starterAlign) >= 10 & start(subject(starterAlign)) ==
+     1 & start(pattern(starterAlign)) != 1)
> prefix <- narrow(unknownSeqs[whichInPrefix], 1, start(pattern(starterAlign[whichInPrefix])) -
+     1)
> prefix <- DNAStringSet(paste(sapply(max(nchar(prefix)) - nchar(prefix),
+     polyn, nucleotides = "-"), as.character(prefix), sep = ""))
> consensusMatrix(prefix, baseOnly = TRUE)

      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
A        1    2    0    0    5    0    0    0    0     0    11    12     0
C        0    0    0    0    0    6    7    0    0     0     0     0     0
G        0    0    3    4    0    0    0    0    9    10     0     0     0
T        0    0    0    0    0    0    0    8    0     0     0     0    13
other   25   24   23   22   21   20   19   18   17    16    15    14    13
      [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24] [,25]
A        14     0     0     0     0     0    20     0    22    23    24    25
C         0     0     0     0    18     0     0     0     0     0     0     0
G         0     0    16    17     0    19     0    21     0     0     0     0
T         0    15     0     0     0     0     0     0     0     0     0     0
other    12    11    10     9     8     7     6     5     4     3     2     1
      [,26]
A         0
C        26
G         0
T         0
other     0

> prefixString <- consensusString(consensusMatrix(prefix, baseOnly = TRUE)[-5,
+     ])
> prefixString

[1] "AAGGACCTGGAATATGGCGAGAAAAC"
```

Exercise 21
The next step is to find which alignments are in the "suffix" of the starter sequence. These are the sequences that overlap to the right of the start sequence.

```
> whichInSuffix <- (score(starterAlign) >= 10 & end(subject(starterAlign)) ==
+     36 & end(pattern(starterAlign)) != 36)
> suffix <- narrow(unknownSeqs[whichInSuffix], end(pattern(starterAlign[whichInSuffix])) +
+     1, 36)
> suffix <- DNAStringSet(paste(as.character(suffix), sapply(max(nchar(suffix)) -
+     nchar(suffix), polyn, nucleotides = "-"), sep = ""))
> consensusMatrix(suffix, baseOnly = TRUE)
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
A       26    0    0    0    0   21   20   19    0    17     0     0     0
C        0   25    0    0    0    0    0    0    0     0     0     0     0
G        0    0   24    0   22    0    0    0    0     0     0    15    14
T        0    0    0   23    0    0    0    0   18     0    16     0     0
other    0    1    2    3    4    5    6    7    8     9    10    11    12
      [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24] [,25]
A         0     0    11     0     0     8     7     6     5     0     0     0
C        13     0     0     0     0     0     0     0     0     4     0     0
G         0    12     0    10     9     0     0     0     0     0     0     2
T         0     0     0     0     0     0     0     0     0     0     3     0
other    13    14    15    16    17    18    19    20    21    22    23    24
      [,26]
A         1
C         0
G         0
T         0
other    25
```

```
> suffixString <- consensusString(consensusMatrix(suffix, baseOnly = TRUE)[-5,
+     ])
> suffixString
```

```
[1] "ACGTGAAATATGGCGAGGAAAACTGA"
```

Exercise 22
Now combine the prefix and suffix with the starter sequence.

```
> extendedUnknown <- DNAString(paste(prefixString, as.character(starterSeq),
+     suffixString, sep = ""))
> extendedUnknown
```

```
  88-letter "DNAString" instance
seq: AAGGACCTGGAATATGGCGAGAAAACTGAAAATCAC...ACACTTTAGGACGTGAAATATGGCGAGGAAAACTGA
```

Exercise 23
Align the set of unknown sequences against the extended sequence.

```
> unknownAlign <- pairwiseAlignment(unknownSeqs, extendedUnknown,
+     substitutionMatrix = submat, gapExtension = -Inf, type = "overlap")
> table(score(unknownAlign))
```

```
 0  1  2  3  4  5  6  7  8  9 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
12 26 26  2  1  1  1  1  1  1  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2 53
```

Exercise 24
Use the `countPDict` *function within nested* `sapply/lapply` *function calls to show the number of reads that map to the unknown sequence in the 8 lanes from the 2 experiments.*

```
> sapply(topReads, lapply, function(x) {
+     if (nrow(x) > 0) {
+         whichNoNs <- (alphabetFrequency(x[["read"]])[, "N"] ==
+             0)
+         x <- x[whichNoNs, ]
```

```
+         pdict <- PDict(x[["read"]])
+         whichMapped <- (countPDict(pdict, extendedUnknown) +
+             countPDict(pdict, reverseComplement(extendedUnknown))) >
+             0
+         sum(x[whichMapped, "count"])
+     }
+ })
```

```
      experiment1 experiment2
lane1 1577        10855
lane2 4627        10482
lane3 1284        10633
lane4 2219        8400
lane5 0           0
lane6 1659        13095
lane7 1823        11099
lane8 4657        14916
```

Exercise 25

Use the `vcountPattern` *function to find which chromosome the extended unknown sequence maps to.*

```
> unknownPatternCount <- vcountPattern(extendedUnknown, Mmusculus)
> tapply(unknownPatternCount[["count"]], unknownPatternCount[["seqname"]],
+     sum)
```

```
        chr1         chr2         chr3         chr4         chr5         chr6
           0            1            0            0            0            0
        chr7         chr8         chr9        chr10        chr11        chr12
           0            0            0            0            0            0
       chr13        chr14        chr15        chr16        chr17        chr18
           0            0            0            0            0            0
       chr19         chrX         chrY         chrM  chr1_random  chr3_random
           0            0            0            0            0            0
 chr4_random  chr5_random  chr7_random  chr8_random  chr9_random chr13_random
           0            0            0            0            0            0
chr16_random chr17_random  chrX_random  chrY_random chrUn_random
           0            0            0            0            0
```

Exercise 26

Finally use the `matchPattern` *function to find the exact location on the chromosome that it maps to.*

```
> mm9Chr2 <- Mmusculus[["chr2"]]
> mm9Ch2View <- matchPattern(extendedUnknown, mm9Chr2)
> mm9Ch2View
```

```
  Views on a 181748087-letter DNAString subject
subject: NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN...GGTTAGGTCTAGGGTTTGCGCCTGGATTACGGGT
views:
       start      end width
[1] 98507289 98507376    88 [AAGGACCTGGAATATGGCGAGAAA...TGAAATATGGCGAGGAAAACTGA]
```

5 Aligning Bacteriophage Reads

Solexa's SOP includes dedicating lane 5 from a set of 8 to sequencing the bacterophage ϕX174 genome, a circular single-stranded genome with 5386 base pairs and the first to be sequenced in 1978. Analyzing the data from this lane can provide a check for a systematic failure of the sequencer.

Exercise 27
Read in one of the lane 5 export files from a Solexa run.

```
> sp <- SolexaPath(system.file("extdata", "ELAND", "080828_HWI-EAS88_0003",
+     package = "SeqBasicsTutorial"))
> phageReads <- readAligned(analysisPath(sp), "s_5_1_export.txt",
+     "SolexaExport")
```

Exercise 28
Find the distinct number of reads and number of times they occurred.

```
> phageReadTable <- tables(sread(phageReads), n = Inf)[["top"]]
```

Exercise 29
Find which distinct reads have uncalled bases and create a "clean" set of reads without any uncalled bases.

```
> whichNotClean <- grep("N", names(phageReadTable))
> head(phageReadTable[whichNotClean])
```

```
ANNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN TNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
                                13320                                11892
CNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN GNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
                                 8978                                 7670
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN AANNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
                                 2308                                 1652
```

```
> cleanReadTable <- phageReadTable[-whichNotClean]
> head(cleanReadTable)
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA GATCTTTGGCGGCACGGAGCCGCGCATCACCTGTA
                                70947                                7561
GATCTCCCGAGCATCACCACATTACTGCGGTTATA CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
                                 6740                                2535
GATCTCCATGGCATCACCACATTACTGCGGTTATA GACGTTTGGTCAGTTCCATCAACATCATAGCCAGA
                                 2323                                 439
```

Exercise 30
Load the `phiX174Phage` *object and extract the New England BioLabs (NEB) version, the one used by Solexa, of the bacterophage ϕX174 genome, and extend the genome 34 bases to "linearize" the circular genome.*

```
> data(phiX174Phage)
> names(phiX174Phage)
```

```
[1] "Genbank" "RF70s"   "SS78"    "Bull"    "G97"     "NEB03"
```

```
> nebPhage <- phiX174Phage[[which(names(phiX174Phage) == "NEB03")]]
> nebPhage <- DNAString(paste(as.character(nebPhage), as.character(substr(nebPhage,
+     1, 34)), sep = ""))
> nebPhage
```

```
  5420-letter "DNAString" instance
seq: GAGTTTTATCGCTTCCATGACGCAGAAGTTAACACT...CAGAGTTTTATCGCTTCCATGACGCAGAAGTTAACA
```

Exercise 31
Show an aligned/unaligned breakdown of the read counts in the "Hoover" Solexa QA plot. This can be accomplished through the following steps:

1. *Use the* `PDict` *function to create pattern dictionaries for the cleaned reads and their reversed complement.*
2. *Use the* `countPDict` *function to find which reads map at least once to the phage genome.*
3. *Create an indicator variable that states whether or not a distinct sequence maps to the phage genome.*

```
> posPDict <- PDict(DNAStringSet(names(cleanReadTable)), max.mismatch = 2)
> negPDict <- PDict(reverseComplement(DNAStringSet(names(cleanReadTable))),
+     max.mismatch = 2)
> whichAlign <- rep(FALSE, length(phageReadTable))
> whichAlign[-whichNotClean] <- (countPDict(posPDict, nebPhage,
+     max.mismatch = 2) + countPDict(negPDict, nebPhage, max.mismatch = 2) >
+     0)
```

Exercise 32
Count the number of distinct reads that map to the genome as well as the overall percentage of reads that map to the genome.

```
> table(whichAlign)

whichAlign
 FALSE   TRUE
312787 196626

> round(sapply(split(phageReadTable, whichAlign), sum)/sum(phageReadTable),
+     2)

FALSE  TRUE
 0.19  0.81
```

Exercise 33
Create a histogram, conditioned on alignment status, that shows the "Hoover" plot mentioned in the ShortRead vignette.

```
> print(histogram(~log10(phageReadTable[phageReadTable > 1]) |
+     whichAlign[phageReadTable > 1], xlab = "log10(Read Counts)",
+     main = "Read Counts by IS(Aligned to Phage)"))
```

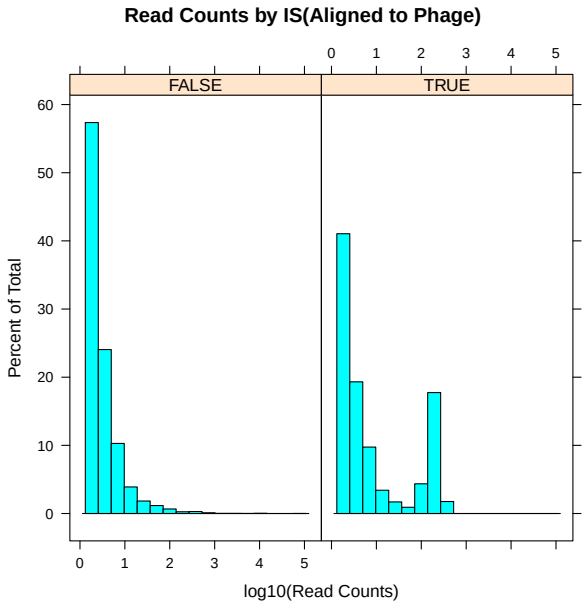

Figure 1: Hoover Plot Deconstructed

```
> toLatex(sessionInfo())
```

- R version 2.10.0 Patched (2009-11-03 r50305), `i386-apple-darwin9.8.0`
- Locale: `en_US.UTF-8/en_US.UTF-8/C/C/en_US.UTF-8/en_US.UTF-8`
- Base packages: base, datasets, graphics, grDevices, methods, stats, utils
- Other packages: Biostrings 2.14.5, BSgenome 1.14.1, BSgenome.Mmusculus.UCSC.mm9 1.3.15, IRanges 1.4.7, lattice 0.17-26, SeqBasicsTutorial 0.0.1, ShortRead 1.4.0
- Loaded via a namespace (and not attached): Biobase 2.6.0, grid 2.10.0, hwriter 1.1, tools 2.10.0

Table 4: The output of `sessionInfo` while creating this vignette.

6 Session Information