Exploring the 1.3 million brain cell scRNA-seq data from 10X Genomics

Aaron Lun[1] ***and Martin Morgan***[2]

[1]Cancer Research UK Cambridge Institute, Cambridge, UK
[2]Roswell Park Cancer Institute, Buffalo, NY

1 November 2018

Package

TENxBrainData 1.2.0

1 Exploring the 1.3 million brain cell scRNA-seq data from 10X Genomics

Package: *TENxBrainData (https://bioconductor.org/packages/3.8/TENxBrainData)*
Author: Aaron Lun (alun@wehi.edu.au (mailto:alun@wehi.edu.au)), Martin Morgan
Modification date: 30 December, 2017
Compilation date: 2018-11-01

The *TENxBrainData (https://bioconductor.org/packages/3.8/TENxBrainData)* package provides a *R* / *Bioconductor* resource for representing and manipulating the 1.3 million brain cell single-cell RNA-seq (scRNA-seq) data set generated by 10X Genomics (https://support.10xgenomics.com/single-cell-gene-expression/datasets/1.3.0/1M_neurons). It makes extensive use of the `r Biocpkg("HDF5Array")` package to avoid loading the entire data set in memory, instead storing the counts on disk as a HDF5 file and loading subsets of the data into memory upon request.

2 Initial work flow

2.1 Loading the data

We use the `TENxBrainData` function to download the relevant files from *Bioconductor*'s ExperimentHub web resource. This includes the HDF5 file containing the counts, as well as the metadata on the rows (genes) and columns (cells). The output is a single `SingleCellExperiment` object from the *SingleCellExperiment (https://bioconductor.org/packages/3.8/SingleCellExperiment)* package. This is equivalent to a `SummarizedExperiment` class but with a number of features specific to single-cell data.

```
library(TENxBrainData)
tenx <- TENxBrainData()
tenx
```

```
## class: SingleCellExperiment
## dim: 27998 1306127
## metadata(0):
## assays(1): counts
## rownames: NULL
## rowData names(2): Ensembl Symbol
## colnames(1306127): AAACCTGAGATAGGAG-1 AAACCTGAGCGGCTTC-1
...
##   TTTGTCAGTTAAAGTG-133 TTTGTCATCTGAAAGA-133
## colData names(4): Barcode Sequence Library Mouse
## reducedDimNames(0):
## spikeNames(0):
```

The first call to `TENxBrainData()` will take some time due to the need to download some moderately large files. The files are then stored locally such that ensuing calls in the same or new sessions are fast.

The count matrix itself is represented as a `DelayedMatrix` from the *DelayedArray (https://bioconductor.org/packages/3.8/DelayedArray)* package. This wraps the underlying HDF5 file in a container that can be manipulated in R. Each count represents the number of unique molecular identifiers (UMIs) assigned to a particular gene in a particular cell.

```
counts(tenx)
```

```
## <27998 x 1306127> DelayedMatrix object of type "integer":
##             AAACCTGAGATAGGAG-1 ... TTTGTCATCTGAAAGA-133
##     [1,]                     0   .                     0
##     [2,]                     0   .                     0
##     [3,]                     0   .                     0
##     [4,]                     0   .                     0
##     [5,]                     0   .                     0
##      ...                     .   .                     .
## [27994,]                     0   .                     0
## [27995,]                     1   .                     0
## [27996,]                     0   .                     0
## [27997,]                     0   .                     0
## [27998,]                     0   .                     0
```

2.2 Exploring the data

To quickly explore the data set, we compute some summary statistics on the count matrix. We increase the *DelayedArray (https://bioconductor.org/packages/3.8/DelayedArray)* block size to indicate that we can use up to 2 GB of memory for loading the data into memory from disk.

```
options(DelayedArray.block.size=2e9)
```

We are interested in library sizes `colSums(counts(tenx))`, number of genes expressed per cell `colSums(counts(tenx) != 0)`, and average expression across cells `rowMeans(counts(tenx)). A naive implement might be

```
lib.sizes <- colSums(counts(tenx))
n.exprs <- colSums(counts(tenx) != 0L)
ave.exprs <- rowMeans(counts(tenx))
```

However, the data is read from disk, disk access is comparatively slow, and the naive implementation reads the data three times. Instead, we'll divide the data into column 'chunks' of about 10,000 cells; we do this on a subset of data to reduce computation time during the exploratory phase.

```
tenx20k <- tenx[, seq_len(20000)]
chunksize <- 5000
cidx <- snow::splitIndices(ncol(tenx20k), ncol(tenx20k) / ch
unksize)
```

and iterate through the file reading the data and accumulating statistics on each iteration.

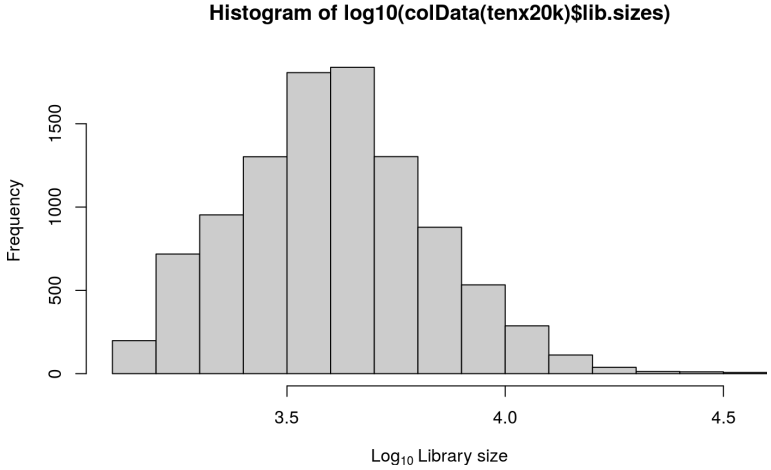
```
lib.sizes <- n.exprs <- numeric(ncol(tenx20k))
tot.exprs <- numeric(nrow(tenx20k))
for (i in head(cidx, 2)) {
    message(".", appendLF=FALSE)
    m <- as.matrix(counts(tenx20k)[,i, drop=FALSE])
    lib.sizes[i] <- colSums(m)
    n.exprs[i] <- colSums(m != 0)
    tot.exprs <- tot.exprs + rowSums(m)
    }
ave.exprs <- tot.exprs / ncol(tenx20k)
```

Since the calculations are expensive and might be useful in the future, we annotate the `tenx20k` object

```
colData(tenx20k)$lib.sizes <- lib.sizes
colData(tenx20k)$n.exprs <- n.exprs
rowData(tenx20k)$ave.exprs <- ave.exprs
```

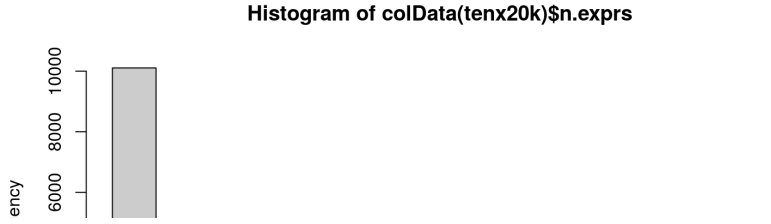
Library sizes follow an approximately log normal distribution, and are surprisingly small.

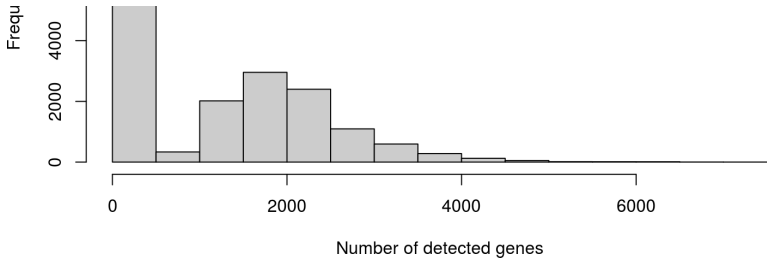
```
hist(
    log10(colData(tenx20k)$lib.sizes),
    xlab=expression(Log[10] ~ "Library size"),
    col="grey80"
)
```



Expression of only a few thousand genes are detected in each sample.

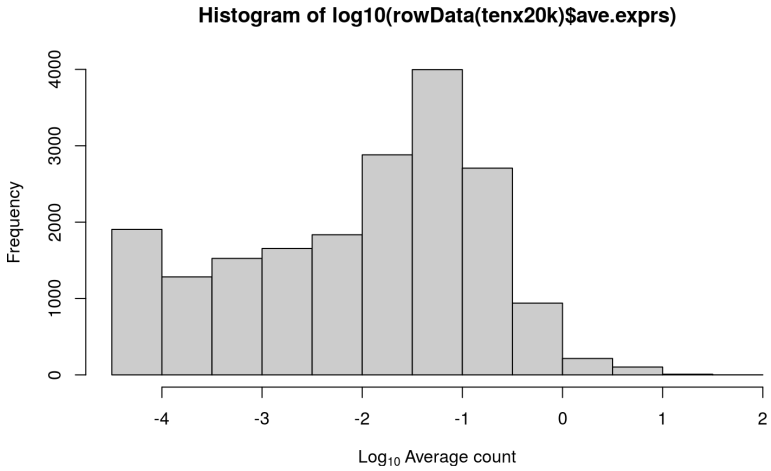
```
hist(colData(tenx20k)$n.exprs, xlab="Number of detected gene
s", col="grey80")
```





Average expression values (read counts) are small.

```
hist(
    log10(rowData(tenx20k)$ave.exprs),
    xlab=expression(Log[10] ~ "Average count"),
    col="grey80"
)
```



We also examine the top most highly-expressing genes in this data set.

```
o <- order(rowData(tenx20k)$ave.exprs, decreasing=TRUE)
head(rowData(tenx20k)[o,])
```

```
## DataFrame with 6 rows and 3 columns
##                Ensembl  Symbol ave.exprs
##            <character> <array> <numeric>
## 1 ENSMUSG00000092341  Malat1   49.1875
## 2 ENSMUSG00000049775  Tmsb4x   24.3624
## 3 ENSMUSG00000072235  Tuba1a  23.27965
## 4 ENSMUSG00000064357 mt-Atp6   17.1341
## 5 ENSMUSG00000064358  mt-Co3  14.32755
## 6 ENSMUSG00000028832   Stmn1  13.96235
```

More advanced analysis procedures are implemented in various *Bioconductor* packages - see the `SingleCell` biocViews for more details.

2.3 Saving computations

Saving the `tenx` object in a standard manner, e.g.,

```
destination <- tempfile()
saveRDS(tenx, file = destination)
```

saves the row-, column-, and meta-data as an *R* object, and remembers the location and subset of the HDF5 file from which the object is derived. The object can be read into a new *R* session with `readRDS(destination)`, provided the HDF5 file remains in it's original location.

3 Improving computational performance

3.1 Parallel computation

Row and column summary statistics can be computed in parallel, for instance using `bpiterate()` in the BiocParallel (https://bioconductor.org/packages/BiocParallel) package. We load the package and start 5 'snow' workers (separate processes).

```
library(BiocParallel)
register(bpstart(SnowParam(5)))
```

This function requires an `iterator` to generate chunks of data. Our iterator returns a function that itself returns the start and end column indexes of each chunk, until there are no more chunks.

```
iterator <- function(tenx, cols_per_chunk = 5000, n = Inf) {
    start <- seq(1, ncol(tenx), by = cols_per_chunk)
    end <- c(tail(start, -1) - 1L, ncol(tenx))
    n <- min(n, length(start))
    i <- 0L
    function() {
        if (i == n)
            return(NULL)
        i <<- i + 1L
        c(start[i], end[i])
    }
}
```

Here is the iterator in action

```
iter <- iterator(tenx)
iter()
```

```
## [1]    1 5000
```

```
iter()
```

```
## [1]  5001 10000
```

```
iter()
```

```
## [1] 10001 15000
```

`bpiterate()` requires a function that acts on each data chunk. It receives the output of the iterator, as well as any other arguments it may require, and returns the summary statistics for that chunk

```
fun <- function(crng, counts, ...) {
    ## `fun()` needs to be self-contained for some parallel
back-ends
    suppressPackageStartupMessages({
        library(TENxBrainData)
    })
    m <- as.matrix( counts[ , seq(crng[1], crng[2]) ] )
    list(
        row = list(
            n = rowSums(m != 0), sum = rowSums(m), sumsq = r
owSums(m * m)
        ),
        column = list(
            n = colSums(m != 0), sum = colSums(m), sumsq = c
olSums(m * m)
        )
    )
}
```

We can test this function as

```
res <- fun( iter(), unname(counts(tenx)) )
str(res)
```

```
## List of 2
##  $ row   :List of 3
##   ..$ n    : num [1:27998] 114 0 0 4 0 ...
##   ..$ sum  : num [1:27998] 120 0 0 4 0 ...
##   ..$ sumsq: num [1:27998] 134 0 0 4 0 ...
##  $ column:List of 3
##   ..$ n    : num [1:5000] 2077 2053 2503 1617 1402 ...
##   ..$ sum  : num [1:5000] 4740 4309 6652 2930 2408 ...
##   ..$ sumsq: num [1:5000] 52772 32577 90380 19598 16622
...
```

Finally, `bpiterate()` requires a function to reduce succesive values returned by `fun()`

```
reduce <- function(x, y) {
    list(
        row = Map(`+`, x$row, y$row),
        column = Map(`c`, x$column, y$column)
    )
}
```

A test is

```
str( reduce(res, res) )
```

```
## List of 2
##  $ row   :List of 3
##   ..$ n    : num [1:27998] 228 0 0 8 0 ...
##   ..$ sum  : num [1:27998] 240 0 0 8 0 ...
##   ..$ sumsq: num [1:27998] 268 0 0 8 0 ...
##  $ column:List of 3
##   ..$ n    : num [1:10000] 2077 2053 2503 1617 1402 ...
##   ..$ sum  : num [1:10000] 4740 4309 6652 2930 2408 ...
##   ..$ sumsq: num [1:10000] 52772 32577 90380 19598 16622
...
```

Putting the pieces together and evaluating the first 25000 columns, we have

```
res <- bpiterate(
    iterator(tenx, n = 5), fun, counts = unname(counts(ten
x)),
    REDUCE = reduce, reduce.in.order = TRUE
)
str(res)
```

```
## List of 2
##  $ row   :List of 3
##   ..$ n    : num [1:27998] 579 1 0 29 0 ...
##   ..$ sum  : num [1:27998] 602 1 0 29 0 ...
##   ..$ sumsq: num [1:27998] 652 1 0 29 0 ...
##  $ column:List of 3
##   ..$ n    : num [1:25000] 1807 1249 2206 1655 3326 ...
##   ..$ sum  : num [1:25000] 4046 2087 4654 3193 8444 ...
##   ..$ sumsq: num [1:25000] 35338 14913 31136 21619 106780
...
```

3.2 Working with Rle-compressed HDF5 data

The 10x Genomics data is also distributed in a compressed format, available from ExperimentHub

```
library(ExperimentHub)
hub <- ExperimentHub()
query(hub, "TENxBrainData")
```

```
## ExperimentHub with 8 records
## # snapshotDate(): 2018-10-31
## # $dataprovider: 10X Genomics
## # $species: Mus musculus
## # $rdataclass: character
## # additional mcols(): taxonomyid, genome, description,
## #   coordinate_1_based, maintainer, rdatadateadded, prepa
rerclass,
## #   tags, rdatapath, sourceurl, sourcetype
## # retrieve records with, e.g., 'object[["EH1039"]]'
##
##            title
##   EH1039 | Brain scRNA-seq data, 'HDF5-based 10X Genomics
' format
##   EH1040 | Brain scRNA-seq data, 'dense matrix' format
##   EH1041 | Brain scRNA-seq data, sample (column) annotati
on
##   EH1042 | Brain scRNA-seq data, gene (row) annotation
##   EH1689 | Brain scRNA-seq data 20k subset, 'HDF5-based 1
0x Genomics' fo...
##   EH1690 | Brain scRNA-seq data 20k subset, 'dense matrix
' format
##   EH1691 | Brain scRNA-seq data 20k subset, sample (colum
n) annotation
##   EH1692 | Brain scRNA-seq data 20k subset, gene (row) an
notation
```

```
fname <- hub[["EH1039"]]
```

The structure of the file can be seen using the `h5ls()` command from rhdf5 (https://bioconductor.org/packages/rhdf5).

```
h5ls(fname)
```

```
##   group       name       otype  dclass        dim
## 0     /       mm10   H5I_GROUP
## 1 /mm10   barcodes H5I_DATASET  STRING    1306127
## 2 /mm10       data H5I_DATASET INTEGER 2624828308
## 3 /mm10 gene_names H5I_DATASET  STRING      27998
## 4 /mm10      genes H5I_DATASET  STRING      27998
## 5 /mm10    indices H5I_DATASET INTEGER 2624828308
## 6 /mm10     indptr H5I_DATASET INTEGER    1306128
## 7 /mm10      shape H5I_DATASET INTEGER          2
```

Non-zero counts are in the `/mm10/data` path. `/mm10/indices` represent the row indices corresponding to each non-zero count. `/mm10/indptr` divides the data and indices into successive columns. For instance

```
start <- h5read(fname, "/mm10/indptr", start=1, count=25001)
head(start)
```

```
## [1]     0  1807  3056  5262  6917 10243
```

retrieves the offsets into `/mm10/data` of the first 25001 columns of data. The offsets are 0-based because HDF5 use 0-based indexing; we will sometimes need to add 1 to facilitate use in *R*.

Here we read the first 25000 columns of data into *R*, using data.table (https://cran.r-project.org/?package=data.table) for efficient computation on this large data.

```
library(data.table)
dt <- data.table(
    row = h5read(fname, "/mm10/indices", start = 1, count =
tail(start, 1)) + 1,
    column = rep(seq_len(length(start) - 1), diff(start)),
    count = h5read(fname, "/mm10/data", start = 1, count = t
ail(start, 1))
)
dt
```

```
##             row column count
##        1: 27995      1     1
##        2: 27921      1    19
##        3: 27918      1    14
##        4: 27915      1    40
##        5: 27914      1    29
##       ---
## 51028822:    63  25000     9
## 51028823:    39  25000     2
## 51028824:    38  25000     1
## 51028825:    17  25000     2
## 51028826:     8  25000     1
```

Row and column summaries are then

```
dt[ ,
    list(n = .N, sum = sum(count), sumsq = sum(count * coun
t)),
    keyby=row]
```

```
##          row     n   sum sumsq
##     1:     1   579   602   652
##     2:     2     1     1     1
##     3:     4    29    29    29
##     4:     6   215   615  3501
##     5:     7     5     5     5
##    ---
## 20191: 27980     7     7     7
## 20192: 27994    22    22    22
## 20193: 27995 14773 28213 77081
## 20194: 27996  1946  2085  2393
## 20195: 27998    16    16    16
```

```
dt[ ,
    list(n = .N, sum = sum(count), sumsq = sum(count * coun
t)),
    keyby=column]
```

```
##        column    n  sum  sumsq
##     1:      1 1807 4046  35338
##     2:      2 1249 2087  14913
##     3:      3 2206 4654  31136
##     4:      4 1655 3193  21619
##     5:      5 3326 8444 106780
##    ---
## 24996:  24996 1142 2091  16827
## 24997:  24997 2610 5772 131212
## 24998:  24998 2209 4492  36600
## 24999:  24999 1049 1791  15795
## 25000:  25000 2015 4043  28809
```

Iterating through 25000 columns of dense data took about 3 minutes of computational time (about 30 seconds elapsed time using 6 cores), compared to just a few seconds for sparse data. Processing the entire sparse data set would still require chunk-wise processing except on large-memory machines, and would benefit from parallel computation. In the later case, processing fewer than 25000 columns per chunk would reduce memory consumption of each chunk and hence allow more processing cores to operate, increasing overall processing speed.

4 Session information

```
sessionInfo()
```

```
## R version 3.5.1 Patched (2018-07-12 r74967)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 16.04.5 LTS
##
## Matrix products: default
## BLAS: /home/biocbuild/bbs-3.8-bioc/R/lib/libRblas.so
## LAPACK: /home/biocbuild/bbs-3.8-bioc/R/lib/libRlapack.so
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] parallel  stats4    stats     graphics  grDevices uti
ls     datasets
## [8] methods   base
##
## other attached packages:
##  [1] data.table_1.11.8           ExperimentHub_1.8.0
##  [3] AnnotationHub_2.14.0        TENxBrainData_1.2.0
##  [5] HDF5Array_1.10.0            rhdf5_2.26.0
##  [7] SingleCellExperiment_1.4.0  SummarizedExperiment_1.1
2.0
##  [9] DelayedArray_0.8.0          BiocParallel_1.16.0
## [11] matrixStats_0.54.0          Biobase_2.42.0
## [13] GenomicRanges_1.34.0        GenomeInfoDb_1.18.0
## [15] IRanges_2.16.0              S4Vectors_0.20.0
## [17] BiocGenerics_0.28.0         knitr_1.20
## [19] BiocStyle_2.10.0
##
## loaded via a namespace (and not attached):
##  [1] xfun_0.4                      lattice_0.20-35
##  [3] snow_0.4-3                    htmltools_0.3.6
##  [5] yaml_2.2.0                    interactiveDisplayBase
_1.20.0
##  [7] blob_1.1.1                    later_0.7.5
##  [9] DBI_1.0.0                     bit64_0.9-7
## [11] GenomeInfoDbData_1.2.0        stringr_1.3.1
## [13] zlibbioc_1.28.0               evaluate_0.12
## [15] memoise_1.1.0                 httpuv_1.4.5
## [17] curl_3.2                      AnnotationDbi_1.44.0
## [19] Rcpp_0.12.19                  xtable_1.8-3
## [21] backports_1.1.2               promises_1.0.1
## [23] BiocManager_1.30.3            XVector_0.22.0
## [25] mime_0.6                      bit_1.1-14
## [27] digest_0.6.18                 stringi_1.2.4
## [29] bookdown_0.7                  shiny_1.1.0
## [31] grid_3.5.1                    rprojroot_1.3-2
## [33] tools_3.5.1                   bitops_1.0-6
## [35] magrittr_1.5                  RCurl_1.95-4.11
```

```
## [37] RSQLite_2.1.1              pkgconfig_2.0.2
## [39] Matrix_1.2-14              rmarkdown_1.10
## [41] httr_1.3.1                 Rhdf5lib_1.4.0
## [43] R6_2.3.0                   compiler_3.5.1
```