Package 'loomR'

April 23, 2018

Type Package

Title An R interface for loom files

Version 0.2.0

Date 2018-23-18

Description An interface for the single-cell RNAseq-oriented loom format. Loom files are an HDF5-based format for storing and interacting with large single-cell RNAseq datasets. loomR provides an interface for working with loom files in a loom-specific way; we provide routines for validating loom files, iterating with chunks through data within the loom file, and provide a platform for other packages to build support for loom files.

URL https://github.com/mojaveazure/loomR http://loompy.org/

BugReports https://github.com/mojaveazure/loomR/issues

Depends R (>= 3.2.2),
R6,
hdf5r,
iterators,
itertools

Imports utils,
Matrix,
methods

Suggests Seurat

Collate 'internal.R'
'loom.R'
'package.R'

License GPL-3

Encoding UTF-8

LazyData true

SystemRequirements HDF5 (>= 1.8.13)

RoxygenNote 6.0.1

R topics documented:

`loomR-package` *An R interface for loom files*

Description

loomR provides an interface for working with loom files in a loom-specific way. We provide routines for validating loom files, iterating with chunks through data within the loom file, and provide a platform for other packages to build support for loom files. Unlike other HDF5 pacakges, loomR actively protectes a loom file's structure, enabling the user to focus on their analysis and not worry about the integrity of their data.

Semantics

Throughout all loomR-related documentation and writing, the following styles for distinguising between loom files, `loom` objects, and loomR will and be used. When talking about loom files, or the actual HDF5 file on disk, the word 'loom' will be written in normal text. Capitalization will be done based on a language's rules for capitalization in sentences. For English, that means if the word 'loom' appears at the beginning of a sentence and is being used to refer to a loom file, it will be capilatized. Otherwise, it will be lowercase. For `loom` objects, or the object within R, the word 'loom' will always be lowercase and written in monospaced text. When referring to the pacakge loomR, it will always be written in normal text with the 'l', 'o's, and 'm' lowercased and the 'R' uppercased. This style will be used throughout documentation for loomR as well as any vignettes and tutorials produced by the authors.

Loom Files

Loom files are an HDF5-based format for storing and interacting with large single-cell RNAseq datasets. Each loom file has at least six parts to it: the raw expression data (`matrix`), groups for gene- and cell-metadata (`row_attrs` and `col_attrs`, respectively), groups for gene-based and cell-based cluster graphs (`row_graphs` and `col_graphs`, respectively), and `layers`, a group containing alternative representations of the data in `matrix`. Each dataset within the loom file has rules as to what size it may be, creating a structure for the entire loom file and all the data within. This structure is enforced to ensure that data remains intact and retriveable when spread across the various datasets in the loom file.

`matrix` The dataset that sets the dimensions for most other datasets within a loom file. This dataset has 'n' genes and 'm' cells. Due to the way that loomR presents data, this will appear as 'm'

rows and 'n' columns. However, other HDF5 libraries will generally present the data as 'n' rows and 'm' columns

`row_attrs` **and** `col_attrs` These are one- or two-dimensional datasets where a specific dimension is of length 'n', for row attributes, or 'm', for column attributes. Within loomR, this must be the second dimension of two-dimensional datasets, or the length of one-dimensional datasets Most other HDF5 libraries will show this specific dimension as the first dimension for two-dimensional datasets, or the length of one-dimensional datasets.

`row_graphs` **and** `col_graphs` Unlike other datasets within a loom file, these are not controlled by `matrix`. Instead, within these groups are groups for specific graphs. Each graph group will have three datasets that represent the graph in coordinate format: `a` for row indices, `b` for column indices, and `w` for values. Each dataset within a graph must be one-dimensional and all datasets within a graph must be the same length. Not all graphs must be the same length as each other.

`layers` Each dataset within `layers` must have the exact same dimensions as `matrix`

Chunk-based iteration

As loom files can theoretically hold million-cell datasets, performing analysis on these datasets can be impossible due to the memory requirements for holding such a dataset in memory. To combat this problem, `loom` objects offer native chunk-based iteration through the `batch.scan`, `batch.next`, `map`, and `apply` methods. This section will cover the former two methods; the latter two are covered in the loomR tutorial.

`batch.scan` and `batch.next` are the heart of all chunk-based iteration in the `loom` object. These two methods make use of `itertools::ichunk` object to chunk through the data in a loom file. Due to the way that R works, `batch.scan` initializes the iterator and `batch.next` moves through the iterator.

The `batch.scan` method will break a dataset in the loom file into chunks, based on a chunk size given to it. `batch.scan` will work on any dataset, except for two-dimensional attributes and any graph dataset. When iterating over `matrix` and the layers, the `MARGIN` argument tells the `loom` object which way to chunk the data. A `MARGIN` of 1 will chunk over genes while a `MARGIN` of 2 will chunk over cells. For one-dimmensional attributes, `MARGIN` is ignored. `batch.scan` returns an integer whose length is the number of iterations it takes to iterate over the dataset selected.

Pulling data in chunks is done by `batch.next`. This method simply returns the next chunk of data. If `return.data = FALSE` is passed, `batch.next` will instead return the indices of the next chunk. When using these methods, we recommend storing the results of `batch.scan` and iterating through this vector to keep track of where the `loom` object is in the iteration.

```
# Set up the iterator on the `loom` object lfile
batch <- lfile$batch.scan(dataset.use = 'matrix', MARGIN = 2)
# Iterate through the dataset, pulling data
# If `return.data = FALSE` is passed, the indices
# of the next chunk will be returned instead
for (i in batch) {
  data.use <- lfile$batch.next()
}
```

Extending loomR

The `loom` class is the heart of loomR. This class is written in the R6 object style and can be extended in three ways. For each of the following, one be discretionary when `return` is used instead of `invisible`. As `loom` object are merely handles to loom files, any function or method that modifies the file should not need to return anything. However, we recommend always returning the `loom` object invisibly, using `invisible`. While not necessary for functionality, it means that objects in a user's environment won't get overwritten if they try to reassign their `loom` object to the output of a function. For functions and methods that don't modify the loom file, and instead return data, then the `return` function should be used.

The first way to extend `loom` objects is by subclassing the object and making a new R6 class. This allows new classes to declare custom R6 methods and gain access to all of the `loom` object's methods, including S3- and S4-style methods. New classes can also overwrite any methods for `loom` objects, allowing the extender to change the core behaviour of `loom` objects. While this option allows the greatest control and access to the `loom` object, it involves the greatest amount of work as one would need to write a new R6 class and all the associated boilerplate code. As such, we recommend subclassing `loom` objects when a new class is needed, but would advise developers to use the other methods of extending `loom` objects for simpler tasks.

The second way is by using S4-style methods can be written for `loom` objects. loomR exports the `loom` class as an S4 class, allowing one to write highly-specialized methods that enforce class-specificity and can change behaviour based on the classes of other objects provided to a function. S4 methods look like normal functions to the end user, but can do different things based on the class, or classes, of objects passed to it. This allows for highly-customized routines without cluttering a package's namespace, as only the generic function is exported. S4 methods can also be written for generics exported by other packages, assuming the said generic has been imported before writing new methods. Furthermore, generics and methods can be kept internally, and R will dispatch the appropriate method as if the generic was exported. However, S4 methods have the drawback of not autocompleting arguments in the terminal or RStudio. This means that the user may need to keep documentation open while using these methods, which detracts from the user-friendliness of these methods. Finally, while there is less boilerplate in declaring S4 generics and methods than declaring R6 classes and methods, there is still more to write than our last method. As such, we recommend S4 methods for anyone who needs method dispatch for internal functions only.

```
#' @export SomeFunction
methods::setGeneric(
  name = 'SomeFunction',
  def = function(object, ...) {
    return(standardGeneric(f = 'SomeFunction))
  }
)

# Note, no extra Roxygen notes needed
methods::setMethod(
  f = 'SomeFunction',
  signature = c('object' = 'loom'),
  definition = function(object, loom.param, ...) {
    # do something
  }
)
```

As R6 objects are based on S3 objects, the final way to extend `loom` objects is by writing S3-style methods. These methods involve the least amount of boilerplate to set up. S3 generics are written just like normal functions, albiet with a few differences. Firstly, they have two arguments: the argument that determines the class for dispatching and `...` to pass other arguments to methods. Finally, the only thing an S3 generic needs to do is call `UseMethod` to allow R to dispatch based on the class of whatever the object is. Unlike S4 methods, S3 methods provide tab-autocompletion for method-specific arguments, providing help messages along the way. This means that S3 methods are more user-friendly than S4 methods. Like S4 methods, S3 methods can use S3 generics declared by other packages, with the same assumptions about imports applying here as well. However, S3 methods cannot be kept internally, and must be exported for R to properly dispatch the method. This means that a package's namespace will have n + 1 functions declared for every S3 generic, where n is the number of classes a method is declared for and the one extra is for the generic. Furthermore, as the methods themselves are exported, anyone can simply use the method directly rather than go through the generic and have R dispatch a method based on object class. Despite these drawbacks, S3 methods are how we recommend one extends loomR unless one needs the specific features of R6 classes or S4-style methods.

```
#' @export somefunction
somefunction <- function(object, ...) {
  UseMethod('somefunction', object)
}

#' @export somefunction.loom
#' @method somefunction loom
somefunction.loom <- function(object, loom.param, ...) {
  # do something
}
```

`combine` *Combine loom files*

Description

Combine loom files

Usage

```
combine(looms, filename, chunk.size = 1000, order.by = NULL,
  overwrite = FALSE, display.progress = TRUE, ...)
```

Arguments

`looms`	A list of `loom` objects or paths to loom files
`filename`	Name for resultant loom file
`chunk.size`	How many rows from each input loom should we stream to the merged loom file at any given time?

order.by	Optional row attribute to order each input loom by, must be one dimensional
overwrite	Overwrite filename if already exists?
display.progress	Display progress as we're copying over data

Value

A loom object connected to filename

See Also

loom

---

connect *Connect to a loom file*

Description

Connect to a loom file

Usage

```
connect(filename, mode = "r", skip.validate = FALSE)
```

Arguments

filename	The loom file to connect to
mode	How do we connect to it? Pass 'r' for read-only or 'r+' for read/write. If mode is 'r+', loomR will automatically add missing required groups during validation
skip.validate	Skip the validation steps, use only for extremely large loom files

Value

A loom file connection

See Also

loom

create *Create a loom object*

Description

Create a loom object

Usage

```
create(filename, data, gene.attrs = NULL, cell.attrs = NULL,
  layers = NULL, chunk.dims = "auto", chunk.size = 1000,
  do.transpose = TRUE, calc.numi = FALSE, overwrite = FALSE,
  display.progress = TRUE)
```

Arguments

filename	The name of the new loom file
data	The data for /matrix. If cells are rows and genes are columns, set do.transpose = FALSE; otherwise, set do.transpose = TRUE
gene.attrs	A named list of vectors with extra data for genes, each vector must be as long as the number of genes in data
cell.attrs	A named list of vectors with extra data for cells, each vector must be as long as the number of cells in data
chunk.dims	A one- or two-length integer vector of chunksizes for /matrix, defaults to 'auto' to automatically determine chunksize
chunk.size	How many rows of data should we stream to the loom file at any given time?
do.transpose	Transpose the input? Should be TRUE if data has genes as rows and cells as columns
calc.numi	Calculate number of UMIs and genes expressed per cell? Will store in 'col_attrs/nUMI' and 'col_attrs/nGene', overwriting anything passed to cel.attrs; To set a custom threshold for gene expression, pass an integer value (eg. calc.numi = 5 for a threshold of five counts per cell)
overwrite	Overwrite an already existing loom file?

Value

A connection to a loom file

See Also

loom

loom *A class for connections loom files*

Description

A class for connections loom files

Usage

```
lfile <- loomR::connect(filename = 'myfile.loom')
```

Format

An R6::R6Class object

Fields

`version` Version of loomR object was created under

`shape` Shape of `/matrix` in genes (columns) by cells (rows)

`chunksize` Chunks set for this dataset in columns (cells) by rows (genes)

`matrix` The main data matrix, stored as columns (cells) by rows (genes)

`layers` Additional data matricies, the same shape as `/matrix`

`col.attrs` Extra information about cells

`row.attrs` Extra information about genes

Methods

`add.graph(a, b, w, name, MARGIN, overwrite)`, `add.graph.matrix(mat, name, MARGIN, overwrite)` Add a graph to the loom object; can add either in coorindate format (`add.graph`) or matrix format (`add.graph.matrix`). Stores graph in coordinate format as `[row, col]_graphs/name/a` (row indices), `[row, col]_graphs/name/b` (column indices), and `[row, col]_graphs/name/w` (values)

`a` Integer vector of row indices for graph, must be the same lengths as `b` and `w`

`b` Integer vector of column indices for graph, must be the same lengths as `a` and `w`

`w` Numeric vector of values for graph, must be the same lengths as `a` and `b`

`mat` Graph provided as a matrix (sparse or dense) or data.frame

`name` Name to store graph, will end up being `col_graphs/name` or `row_graphs/name`, depending on `MARGIN`

`MARGIN` Store the graph in `row_graphs` (1) or `col_graphs` (2), defaults to 2

`overwrite` Can overwrite existing graph?

`add.layer(layer, chunk.size, overwrite)` Add a data layer to this loom file, must be the same dimensions as `/matrix`

`layer` A named list of matrices to be added as layers

chunk.size Number of rows from each layer to stream at once, defaults to 1000

overwrite If a layer already exists, overwrite with new data, defaults to FALSE

add.attribute(attribute, MARGIN, overwrite) Add extra information to this loom file.

attribute A named list where the first dimmension of each element as long as one dimension of /matrix

MARGIN Either 1 for genes or 2 for cells

overwrite Can overwrite existing attributes?

add.row.attribute(attribute), add.col.attribute(attribute) Add row or column attributes

get.attribute.df(MARGIN, attribute.names, row.names, col.names) Get a group of row or column attributes as a data frame, will only return attributes that have one dimension

MARGIN Either '1' or '2' to get row- or column-attributes, respectively

attribute.names A vector of attribute dataset basenames

row.names Basename of the rownames dataset

col.names Basename of the colnames dataset

get.graph(name, MARGIN) Get a graph as a sparse matrix

name Name of the graph, can be either the basename or full name of the grpah

MARGIN Load the graph from row_graphs (1) or col_graphs (2), defaults to 2. Ignored if full path to graph is passed to name

batch.scan(chunk.size, MARGIN, index.use, dataset.use, force.reset),batch.next(return.data) Scan a dataset in the loom file from index.use[1] to index.use[2], iterating by chunk.size.

chunk.size Size to chunk MARGIN by, defaults to self$chunksize

MARGIN Iterate over genes (1) or cells (2), defaults to 2

index.use Which specific values of dataset.use to use, defaults to 1:self$shape[MARGIN] (all values)

dataset.use Name of dataset to use, can be the name, not group/name, unless the name is present in multiple groups

force.reset Force a reset of the internal iterator

return.data Return data for a given chunk, if FALSE, returns the indices across MARGIN for said chunk

apply(name, FUN, MARGIN, chunk.size, dataset.use, overwrite, display.progress, ...) Apply a function over a dataset within the loom file, stores the results in the loom file. Will not make multidimensional attributes.

name Full name ('group/name') of dataset to store results to

FUN Function to apply

MARGIN Iterate over genes (1) or cells (2), defaults to 2

index.use Which specific values of dataset.use to use, defaults to 1:self$shape[MARGIN] (all values)

chunk.size Size to chunk MARGIN by, defaults to self$chunksize

dataset.use Name of dataset to use

overwrite Overite name if already exists

display.progress Display progress

... Extra parameters to pass to FUN

`map(FUN, MARGIN, chunk.size, index.use, dataset.use, display.progress, expected, ...)`
Map a function onto a dataset within the loom file, returns the result into R.

`FUN`

`MARGIN` Iterate over genes (1) or cells (2), defaults to 2

`chunk.size` Size to chunk `MARGIN` by, defaults to `self$chunksize`

`index.use` Which specific values of `dataset.use` to use, defaults to `1:self$shape[MARGIN]` (all values)

`dataset.use` Name of dataset to use

`display.progress` Display progress

`...` Extra parameters to pass to `FUN`

`add.cells(matrix.data, attributes.data = NULL, layers.data = NULL, display.progress = TRUE)`
Add cells to a loom file.

`matrix.data` A list of m2 cells where each entry is a vector of length n (num genes, `self$shape[1]`)

`attributes.data` A list where each entry is named for one of the datasets in `self[['col_attrs']]`; each entry is a vector of length m2.

`layers.data` A list where each entry is named for one of the datasets in `self[['layers']]`; each entry is an n-by-m2 matrix where n is the number of genes in this loom file and m2 is the number of cells being added.

`display.progress` Display progress

`add.loom(other, other.key, self.key, ...)` Add the contents of another loom file to this one.

`other` An object of class `loom` or a filename of another loom file

`other.key` Row attribute in `other` to add by

`self.key` Row attribute in this loom file to add by

`...` Ignored for now

See Also

loomR, hdf5r::H5File

`subset.loom` *Subset a loom file*

Description

Subset a loom file

Usage

```
## S3 method for class 'loom'
subset(x, m = NULL, n = NULL, filename = NULL,
  chunk.size = 1000, overwrite = FALSE, display.progress = TRUE, ...)
```

Arguments

x	A loom object
m	Rows (cells) to subset, defaults to all rows
n	Columns (genes) to subset, defaults to all columns
filename	Filename for new loom object, defaults to ...
chunk.size	Chunk size to iterate through x
overwrite	Overwrite filename if already exists?
display.progress	Display progress bars?
...	Ignored for now

Value

A loom object connected to filename

See Also

loom

Index