hypermutR

Hypermutation is a phenomena that affects HIV-1 introducing large numbers of mutations into some sequences. It manifests in the datasets as sequences in which large numbers of Guanine was mutated to Adenine, specifically when that Guanine was surrounded by a particular pattern. The hypermut 2.0 tool available from https://www.hiv.lanl.gov/content/sequence/HYPERMUT/hypermut.html is a frequently used tool to detect and remove hypermutated sequences. We wrote a new implementation of the hypermut 2.0 algorithm in R, which is available in the hypermutR package on CRAN.

1 Algorithm

The hypermut algorithm compares each sequence in an alignment to some ancestral sequence (usually approximated by the consensus sequence of the alignment), tallying the frequency of specific mutations. Hypermutation occurs when a G which is followed by an A or G (denoted by R in the IUPAC convention) and then by an A, G or T (denoted by a D in the IUPAC convention) mutates to an A. More compactly, when GRD become ARD, the mutation is flagged as possibly due to hypermutation. In order to distinguish between true hypermutation and the generally expected level of mutation, a baseline must be established. The baseline is established by tallying G to A mutations when the G is followed immediately by either a C or T (denoted by a Y in the IUPAC convention) or when the G is followed by an A or a G (denoted by R in the IUPAC convention) and then a C. More compactly, when GY becomes AY or GRC becomes ARC, the mutations are tallied as the baseline mutation rate against which the potential hypermutations must be compared.

A one-sided Fisher's exact test is used to compare the proportion of GRD positions that became ARD positions to the proportion of GY or GRC positions that became either AY or ARC positions. When the p-value of the test is smaller than some threshold, with the default set to 0.1 as in (Abrahams et al., 2009), then the individual sequence is flagged as a hypermutant and either the sequence is removed from the dataset, or the mutated bases (the A's followed by RD) are replaced by an R to indicate that we are uncertain whether the mutation was a random mutation or if it was the result of hypermutation.

In order to be a position of interest (either a control or hypermutation position), all that is required is a G in the ancestral sequence. To classify the position into either a hypermutation or control position, only the query sequence is considered. If the two positions following the position that contains the G in the ancestral sequence matches RD, then it is a hypermutation position, else it is a control position. The two

downstream positions in the ancestral sequence are not considered. This implies the assumption that the two downstream positions in the ancestral sequence mutates before the position of interest.

2 Installation Instructions for Ubuntu

Make sure you have a recent version of R. Follow the instructions in the following link to set up the correct repository for apt. http://stackoverflow.com/questions/10476713/how-to-upgrade-r-in-ubuntu.

Make sure that both r-base and r-base-dev is installed:

```
sudo apt-get install r-base r-base-dev
```

Next, install devtools' depedancies with apt-get:

```
sudo apt-get install libssl-dev libxml2-dev libcurl4-gnutls-dev
```

Then, from within R, install devtools:

```
install.packages('devtools', repo = 'http://cran.rstudio.com/')
```

Finally, install hypermutR. From a local file:

```
library(devtools)

install_local('/path/to/file/hypermutR_x.y.z.tar.gz')
```

Please note that you must use `install_local` from `devtools` - `install.packages` will not work. Change `/path/to/file` to the path to the installation file on your computer and x.y.z to match the installation file you have.

Or using the bit_bucket repo:

```
library(devtools)

install_bitbucket('hivdiversity/hypermutR',  auth_user = 'username',

  password = 'password')
```

Finally, hypermutR includes a script that can be run from the commandline. You need to put this script somewhere convenient ('`/usr/bin`' for example)

```
file.symlink(from = file.path(find.package('hypermutR'), 'hypermutR.R'), to =
'/usr/bin')
```

3 Usage instructions

From within an R session: Within R

```
library(hypermutR)
```

```
help('remove_hypermutation')
```

This will display the help for the main function in hypermutR.

From the command line

```
hypermutR -h
```

or (depending on your installation):

```
hypermutR.R -h
```

This will display help for all the options and an example call to hypermutR.

4 Implementation

The implementation of the algorithm can be found in the `hypermutR` R package on CRAN. It has a command line interface built with the `optparse` package (Davis, 2017) providing control over 5 variables (Table 1). The `remove_hypermutation` function is a wrapper that calls `ancestor_processing` to obtain the ancestral sequence to compare the query sequences to, calls `deduplicate_seqs` to remove duplicate sequences for performance reasons, then loops over each unique sequence, comparing it to the ancestral sequence with `scan_seq` and finally collates the results.

Table 1: Parameters that can be controlled by the command line user interface of the hypermutR package.

Name	Description
`input_file`	String specifying the path to and the name of the fasta file containing the alignment of the sequences.
`ouput_file`	String specifying the path to and the name of the file that will contain the resulting sequences (with hypermutated sequences either removed or corrected).
`p_value`	The threshold to use when deciding if the p-value produced by the Fisher test indicates that there is hypermutation present in the sequence.

ancestor	Either 'consensus' to indicate that the consensus sequences must be computed, or 'first' to indicate that the first sequence in the dataset should be considered to be the ancestral sequence, or the ancestral sequence itself.
fix_with	If omitted, hypermutants will be removed. If a single letter is specified, then hypermutants will be corrected by replacing the hypermutated base with the specified letter.

The package is designed to depend exclusively on packages from CRAN (and none from Bioconductor), meaning that the seqinr (Charif & Lobry, 2007) package is used to read and write fasta formatted files. The seqinr package stores sequence data in objects of class SeqFastadna. Formatting data as SeqFastadna objects yields one of two configurations. The first configuration is a vector of character strings, in which each character string is a sequence, with the optional attributes: names, Annot, and class. The alternate structure is a list in which each element represents a single sequence. Each element consists of a vector of single letters of class character with the same optional attributes as the first configuration. The seqinr package provides fasta file access with the read.fasta function which stores data in the second format, the list of vectors of single letters. For consistency, hypermutR also uses the list-based format to store sequence data.

Three options exists for specifying the ancestral sequence to compare the query sequences in the dataset to. If the value 'consensus' is specified via the ancestor parameter, a consensus sequence will be computed from the sequences in the input file. The letter that most frequently occurs is placed in the consensus sequence. In the case of ties, the first letter, when arranged alphabetically, is used. The second option is to include the ancestral sequence as the first sequence in the input file and to set the value of the ancestor parameter to 'first'. In this case, the first sequence will be removed from the dataset before proceeding. Lastly, the ancestor parameter can be assigned the ancestral sequence itself. The only validation that is performed on the last of the three options is to check that the sequence assigned to ancestor has the same length as the sequences in the input file.

The scan_seq operation is slow, so the dataset is deduplicated with the deduplicate_seqs function to improve performance. The dataset is converted to a vector of character strings and the unique sequences are selected with the unique function. Looping over the unique sequences, a list is constructed in which each element corresponds to a unique sequence. Each element is also a list with

the elements `the_seq` containing the actual sequences and `dup_names`, a `vector` of character strings listing the names of all sequences that matches the unique sequences stored in `the_seq`.

The remove_hypermutation function loops over the unique sequences returned by the deduplicate_seqs function. On each unique sequence, the scan_seq function is called. The ancestral sequence provided by the process_ancestors function is also passed to the scan_seq function. The scan_seq function simultaneously passes two sliding windows along the ancestral and query sequences. The sliding window is of length 3, corresponding to the potentially hypermutated position and the 2 downstream positions.

At each position, the size of the window is increased until it covers 3 non-gap characters in the query sequence. If a G is located at the first position of the window, the position is considered a position of interest and the query sequence is inspected to classify it as either a hypermutation or control position, incrementing either the `num_potential_mut` variable or the `num_potential_control` variable. The query sequence is checked next and if the G mutated to an A, then the tally of the number of possible hypermutations (`num_mut`) or the number of control mutations (`num_control`) is incremented.

The return value from `scan_seq` is a `list` that contains the number of mutated hypermutation and control positions, the total number of potential hypermutation and control positions, the p-value of the one-sided Fischer exact test, the (possibly corrected) query sequence and the `data.frame` that catalogs each individual position.

The `remove_hypermut` function binds the `data.frames` that catalog each position together into a full log, called `all_mut_pos`, of all positions of interest in all sequences. After comparing the p-value to the p-value cutoff passed into the `remove_hypermutation` function, each sequence is stored in either a `list` that contains all hypermutated sequences (`seq_hypermutants`) or a `list` that contains all non-hypermutated sequences (`seq_result`). The `remove_hypermut` function returns these three results: `all_mut_pos`, `seq_result`, and `seq_hypermutants`.

The user interface (UI) script, `hypermut.R`, located in the `inst` folder in the package root, writes the return values from `remove_hypermut` to disk. The value of the `input_file` parameter dictates the file name used for `seq_result`. The '.fasta' extension on the value of `input_file` is replaced with '_hypermutants.fasta' to construct the file name for `seq_hypermutants`. Lastly, the file name for the `all_mut_pos data.frame` is obtained by replacing the '.fasta' extension of the `input_file` parameter with '_mut_pos.csv'.

5 Tests

The hypermutR package has a full suite of unit tests built with the testthat package. As per the guidelines of testthat, the testing code is located in the tests/testthat/ subfolder of the package root. The modular design of hypermutR allows the construction of tests that precisely test the functioning of small specialized pieces of code. The organization of the tests mirror that of the code, with matching file names, but 'test_' prepended to the names of the files that contain the test code. The contents of each test file is organized hierarchically into contexts, tests and expectations (Wickham, 2011). An expectation is a single simple requirement that a return value of one of the functions of hypermutR must meet. For example, the class of the return value from the remove_hypermut function must be list. Expectations that cover a set of tightly related operations are grouped together into tests. Tests are further grouped into contexts which provides extra information to help locate the code covered by the context in question.

Each function in hypermutR, except those designed to simulate test scenarios, are covered by a number of expectations checking the format of the output as well as the correctness of a sample of the elements of the return value. A number of tests checks the result of applying the wrapper function remove_hypermut to the ld_seqs and hd_seqs data sets, described later in the section called Data Simulation, in which some sequences were hypermutated. These tests serve as integration tests ensuring that the entire process of removing hypermutated sequences works as expected.

Hypermutation is simulated with the sim_hyper function. Given a sequence dataset, sim_hyper will mutate a specified number of hypermutation and control positions in a given number of sequences. The number of sequences in which to mutations are to be introduced is specified by the parameter n1. The n2 and n3 parameters control the number of hypermutation and control positions to mutate respectively. Each of the parameters may be between zero and one to specify a proportion of sequences or positions. If the parameter values are larger than one then they specify the exact number of sequences or positions. The return value is a named vector of type atomic assigned the class SeqFastadna in which each element of the vector is a DNA sequence.

6 Benchmarks and Comparisons

To ensure that the implementation of hypermutR matches that of the hypermut 2.0 tool on the LANL website and to document any discrepancies, a large number of edge cases were constructed and processed with both hypermutR and the hypermut 2.0 tool. The results of this comparison is shown

in Table 2. In a single case, the `hypermutR` package and the LANL implementation yields different results. This is the case where a control position was deleted in the query sequence. We chose to maintain this mismatch because it is consistent with the behavior when a hypermutation position gets deleted from the query sequence. Furthermore, this is an extremely rare edge case requiring a frameshift deletion in the query sequence.

Table 2: Edge cases evaluated and compared with the Hypermut 2.0 evaluation on the LANL website.

Case	Ancestral sequence	Query Sequence	Result	p-value	Comment
A control position at the first position.	GCACTCAAT	ACACTCAAT	0, 0, 1, 1	1	match
A control position at the last position.	CCACTCGCT	CCACTCACT	0, 0, 1, 1	1	match
A control position was deleted in the ancestral sequence.	ACT-CTACTACT	ACTACTACTACT	0, 0, 0, 0	1	match
A control position was deleted in the query sequence.	ACTGCTACTACT	ACT-CTACTACT	0, 0, 0, 1	1	LANL Result: 0, 0, 0, 0
A hypermutation position at the first position.	GAACTCAAT	AAACTCAAT	1, 1, 0, 0	1	match
A hypermutation position at the last position.	CCACTCGAT	CCACTCAAT	1, 1, 0, 0	1	match
A hypermutation position was deleted in the ancestral sequence.	ACT-AAACTACT	ACTAAAACTACT	0, 0, 0, 0	1	match
A hypermutation position was deleted in the query sequence.	ACTGAAACTACT	ACT-AAACTACT	0, 1, 0, 0	1	match
Control pattern only in the ancestral sequence.	ACTGCTACT	ACTAAAACT	1, 1, 0, 0	1	match
Control pattern only in the query sequence.	ACTGATACT	ACTACCACT	0, 0, 1, 1	1	match
Gaps in a control pattern in both sequences.	ACTGC-ACT	ACTAC-ACT	0, 0, 1, 1	1	match
Gaps in a control pattern in the ancestral sequence.	ACTGC-ACT	ACTACTACT	0, 0, 1, 1	1	match
Gaps in a control pattern in the query sequence.	ACTGCTACT	ACTAC-ACT	0, 0, 1, 1	1	match
Gaps in a hypermutation pattern in both sequences.	ACTGA-ACT	ACTAA-ACT	1, 1, 0, 0	1	match
Gaps in a hypermutation pattern in the ancestral sequence.	CCAGA-TACT	CCAAAATACT	1, 1, 0, 0	1	match
Gaps in a hypermutation pattern in the query sequence.	ACTGAAACT	ACTAA-ACT	1, 1, 0, 0	1	match
Hypermutation pattern only in the ancestral sequence.	ACTGATACT	ACTACTACT	0, 0, 1, 1	1	match
Hypermutation pattern only in the query sequence.	ACTGCTACT	ACTAAAACT	1, 1, 0, 0	1	match
More control mutations than hypermutations.	GAGAGAGAGAGAGC GCGCGCGCGCGC	GAGAGAGAGAGAAC ACACACACACAC	0, 6, 6, 6	1	match
More hypermutation mutations than control mutations.	GAGAGAGAGAGAGC GCGCGCGCGCGC	AAAAAAAAAAAAGC GCGCGCGCGCGC	6, 6, 0, 6	0.0011	match
Overlapping control positions in the ancestral sequence.	ACTGGCACT	ACTAACACT	0, 0, 2, 2	1	match
Overlapping hypermutation positions in the ancestral sequence.	ACTGGAACT	ACTAAAACT	2, 2, 0, 0	1	match
The alignment is of length 2.	GA	AA	0, 0, 0, 0	1	match
The alignment is of length 3 with hypermutation.	GAA	AAA	1, 1, 0, 0	1	match

The alignment is of length 3 without hypermutation.	CAA	AAA	0, 0, 0, 0	1	match
The alignment is of length 4 with hypermutation.	CGAA	CAAA	1, 1, 0, 0	1	match
The alignment is of length 4 without hypermutation.	ACAA	AGAA	0, 0, 0, 0	1	match
The ancestral sequence ends with gaps.	ACTGCTGAAA--	ACTACTAAAACT	1, 1, 1, 1	1	match
The ancestral sequence starts with gaps.	--TGCTGAAACT	ACTACTAAAACT	1, 1, 1, 1	1	match
The query sequence ends with gaps.	ACTGCTGAAACT	ACTACTAAAA--	1, 1, 1, 1	1	match
The query sequence starts with gaps.	ACTGCTGAAACT	--TACTAAAACT	1, 1, 1, 1	1	match

7 Data simulation

The HVTN 503/Phambili study followed HIV negative subjects monitoring for HIV-1 infection to evaluate an HIV-1 vaccine (Gray et al., 2011). For testing purposes, we took the PID Illumina MiSeq sequence data from two time points (HVTN503-162400146-1011, referred to as low diversity or LD dataset, and HVTN503-162450071-1056, referred to as the high diversity or HD dataset) and built phylogenetic trees with RAxML. The setting specified for RAxML together with their explanations are listed in Table 3. Using the trees produced by RAxML, a random subtype-C sequence was selected (referred to as the seed sequence) from LANL (C.ZA.08.707PKE34F2.HM623575), restricted to the same amplicon as the real dataset and mutated according to these trees.

To simulate test data, the trees were loaded into R in a `data.frame` in which each row represents an edge. The `data.frame` contain three columns, the first one listing the ancestor, the second one listing the descendant and the last one the length of the edge. The simulation is initiated by assigning the seed sequence to the descendant in the first row of the dataset. The ancestor is then constructed by randomly mutating the seed sequence until it diverged by the edge length. The newly simulated ancestor sequence is the used to generate the other sequences that are directly related to it. This process is continued until all the sequences in the entire tree (including the internal nodes) are generated. To introduce extra variablility into the datasets, a `mutation_booster` variable was used. This variable was set to 0.5, 1 or 2 and the branch lengths were multiplied by this variable enabling the generation datasets with differing levels of diversity while keeping the underlying phylogeny unchanged. These simulated datasets are referred to by appending _bx to their source dataset where x is the factor by which the branch lengths were multiplied. For example, the dataset constructed by multiplying the branch lengths of the LD dataset by 2 is called the LD_b2 dataset.

To evaluate the simulated datasets, RAxML was used to draw trees from the simulated datasets which was then visually compared to the tree of the real datasets (Figure 1). While minor changes to the trees occurred, the datasets based on the LD dataset maintains a star like phylogeny and the datasets based on the HD dataset exhibits more complex behavior. This simulation formed part of a larger project, and only the LD_b1 and HD_b2 datasets were used (with the names ld_seqs and hd_seqs respectively).

Table 3: RAxML settings used to draw trees from which the testing datasets were simulated.

Setting	Description
-f a	Perform rapid bootstrap analysis and search for the best-scoring maximum likelihood tree in one program run.
-x 12345	Seed for the random number generator used by the rapid bootstrap analysis.
-p 12345	Seed for the random number generator used in the parsimony inferences.
-# 100	The number of bootstrap analyses to run on distinct starting trees.
-m GTRGAMMA	The model used for the nucleotide substitutions. The general time reversible model with optimization of the substitution rates and the GAMMA model of rate heterogeneity.

8 Future Work and Conclusions

The `hypermutR` package is a high quality implementation of the `hypermut 2.0` algorithm that can be used offline. It is available via CRAN, has a comprehensive suite of unit tests and detailed documentation. Many edge cases were evaluated against the version that is available from the LANL website and all except one were found to match. Correcting the single mismatching case would introduce inconsistency into the handling of hypermutation and control positions in `hypermutR`, and thus it was left as is. Currently, `hypermutR` lacks the ability to specify custom patterns and does not support the data formats implemented in Biostrings (Pages, Aboyoun, Gentleman, & DebRoy, 2017).

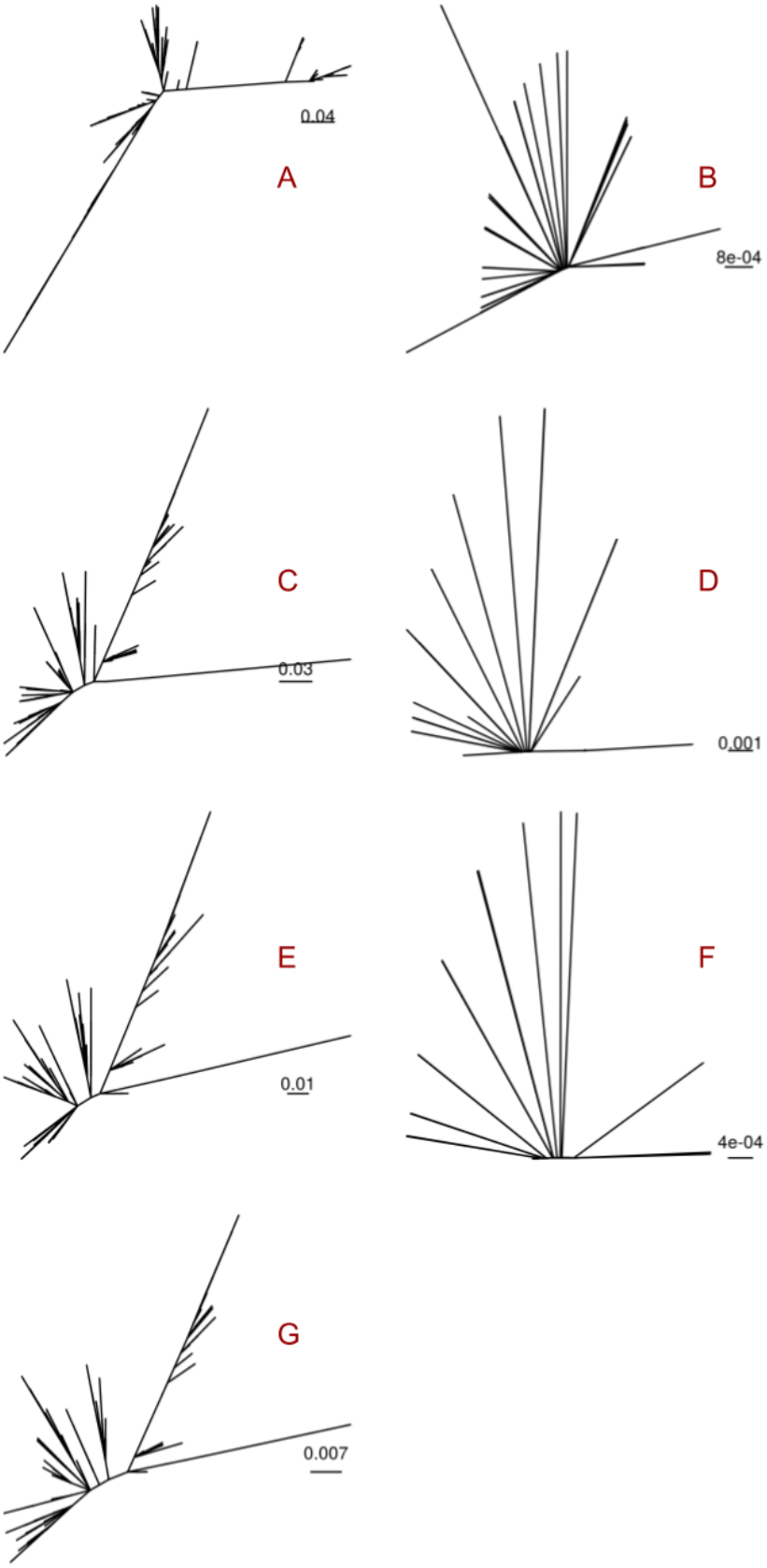

Figure 1: The two trees constructed from the datasets (A and B) and the five trees (C-G) built from the simulated datasets. The left column (A, C, E, G) are based on the HD dataset and the right column (B, D, F) is based on the LD dataset. The first row (A,B) is the trees constructed from the datasets, the second row (C,D) is constructed with the branch lengths doubled, the third row (E,F) is the trees constructed with the branch lengths unmodified and the last row (G) has the branch lengths halved. Due to the very low levels of diversity in the datasets based on the LD dataset, the branch lengths were not halved for this dataset. These datasets are referred to by appending _bx to their source dataset where x is the factor by which the branch lengths were multiplied. For example, the dataset from which tree D was drawn is called LD_b2 and HD_b0.5 for tree G.