hypermutR

Phillip Labuschagne[1] and Paul Edlefsen[2]

Hypermutation is a phenomenon that affects HIV-1 by 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 (Rose & Korber, 2000). 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. However, this tool is only available as a web interface, making batch processing of large volumes cumbersome. Since we wanted to use this tool as part of a data processing pipeline, we implemented the algorithm as an R package called hypermutR.

1.1.1 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 (depending on user input) either the sequence is removed from the dataset, or the mutated bases (the A's followed by RD) are replaced

[1] South African Medical Research Council Bioinformatics Unit, South African5National Bioinformatics Institute, University of the Western Cape., Cape Town, South Africa.
[2] Fred Hutchinson Cancer Research Center, Vaccine and Infectious Disease Division, Seattle, United States.

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 query 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. Note that, since the mechanism that induces hypermutation only affects positions that match GRD, the fact that only the query sequence is considered, means that if the downstream positions in the ancestral sequence does not match RD, then the Hypermut algorithm implicitly assumes that those downstream positions in the query sequence derived from the ancestral sequence by first mutating these positions to match RD, and then the hypermutation mechanism operated on the GRD signal to mutate the G into an A.

1.1.2 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')
```

1.1.3 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.

1.1.4 Implementation

The implementation of the algorithm can be found in the `hypermutR` R package. 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 exist 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. If at a specific position the true, but unknown, ancestral sequence contained a G, but hypermutation changed more than 50% of the sequences at that position into an A, then this consensus approach will assign an A to that position in the sequence used as an ancestral reference sequence. 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 and is entirely deterministic (so yields the same result for duplicate inputs), 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 Fisher's exact test, the (possibly corrected) query sequence and the `data.frame` that catalogs each individual position. As noted above, the return sequence will only be corrected if the `fix_with` option is specified (typically we choose "R" for this option to indicate residual uncertainty between "A" and "G" in these cases, but the fix_with value might instead be specified as "G" to undo the hypermutation change).

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'.

1.1.5 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 check the result of applying the wrapper function `remove_hypermut` to the `ld_seqs` and `hd_seqs` data sets, described later in section **Error! Reference source not found.**, 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 equal to or larger than one then they specify the exact number of sequences or positions. The parameter can also be assigned the value 'all' in which case it will signify that all the positions or sequences should be affected. 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.

1.1.6 Benchmarks and Comparisons

The Hypermut tool on the LANL website is a well-established tool with more than 200 citations. To ensure that the implementation of `hypermutR` is correct and faithfully reproduced the results of the latest version of the Hypermut 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 are shown in Table 2. Very short sequences were chosen so that each result could be computed manually to ensure that the result is consistent with the description of the algorithm.

The construction of the windows to compare between the ancestral and query sequences is the most complex step. Decisions had to be made regarding the handling of insertions, deletions, mutations in the downstream pattern, overlaps between potential sites and the handling of very short sequences. The first eight cases checks the behavior when the position of interest is either at the start or end of the sequence and what happens when the position is demolished by an insertion or deletion. In one of these cases, 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.

Should the ancestral or query sequences be scanned for the pattern? Cases 9 and 10 investigate this question. The pattern in the control sequence is used to classify the position in to a hypermutation or a control position. This implies the assumption that the two downstream positions in the ancestral sequence mutates before the position of interest. Cases 17 and 18 duplicates this investigation, but the description is written from the context of a hypermutation position instead of a control position as with cases 9 and 10.

Gaps in the pattern does not affect the classification of the position irrespective of the number and the location of the gaps as illustrated in cases 11 through 16.

Fisher's exact test can test the two-sided hypothesis that the ratio of mutated to non-mutated control positions is unequal to the ratio of mutated to non-mutated hypermutation positions, or it can test both versions of the one-sided hypothesis (that the ratio of mutated to non-mutated control positions is greater than (or less than) the ratio of mutated to non-mutated hypermutation positions). Cases 19 and 20 confirms that it is a one-sided test checking that the ratio of mutated to non-mutated control positions is equal to or less than the ratio of mutated to non-mutated hypermutation positions.

As shown by cases 21 and 22, positions are allowed to overlap. Letters that form part of the downstream pattern for one positions can themselves be a position of interest and can also be part of the downstream pattern of another position.

If the sequences are shorter that the length of the pattern of a single position, then it is treated as if there are no positions in the sequence (Case 23). Very short sequences (lengths 3 and 4) are treated as normal sequences which can have up to 1 (or 2 in the case of a sequence of length 4) positions of interest (Cases 24 to 27).

Finally, cases 28 to 31 show that gaps at the starts of ends of sequences have no effect on the algorithm aside from the fact that they exclude the first and last positions from being considered as positions of interest.

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
1. A control position at the first position.	GCACTCAAT	ACACTCAAT	0, 0, 1, 1	1	match
2. A control position at the last position.	CCACTCGCT	CCACTCACT	0, 0, 1, 1	1	match
3. A control position was deleted in the ancestral sequence.	ACT-CTACTACT	ACTACTACTACT	0, 0, 0, 0	1	match
4. A control position was deleted in the query sequence.	ACTGCTACTACT	ACT-CTACTACT	0, 0, 0, 1	1	LANL Result: 0, 0, 0, 0
5. A hypermutation position at the first position.	GAACTCAAT	AAACTCAAT	1, 1, 0, 0	1	match
6. A hypermutation position at the last position.	CCACTCGAT	CCACTCAAT	1, 1, 0, 0	1	match
7. A hypermutation position was deleted in the ancestral sequence.	ACT-AAACTACT	ACTAAAACTACT	0, 0, 0, 0	1	match
8. A hypermutation position was deleted in the query sequence.	ACTGAAACTACT	ACT-AAACTACT	0, 1, 0, 0	1	match
9. Control pattern only in the ancestral sequence.	ACTGCTACT	ACTAAAACT	1, 1, 0, 0	1	match
10. Control pattern only in the query sequence.	ACTGATACT	ACTACCACT	0, 0, 1, 1	1	match
11. Gaps in a control pattern in both sequences.	ACTGC-ACT	ACTAC-ACT	0, 0, 1, 1	1	match
12. Gaps in a control pattern in the ancestral sequence.	ACTGC-ACT	ACTACTACT	0, 0, 1, 1	1	match
13. Gaps in a control pattern in the query sequence.	ACTGCTACT	ACTAC-ACT	0, 0, 1, 1	1	match
14. Gaps in a hypermutation pattern in both sequences.	ACTGA-ACT	ACTAA-ACT	1, 1, 0, 0	1	match
15. Gaps in a hypermutation pattern in the ancestral sequence.	CCAGA-TACT	CCAAAATACT	1, 1, 0, 0	1	match
16. Gaps in a hypermutation pattern in the query sequence.	ACTGAAACT	ACTAA-ACT	1, 1, 0, 0	1	match
17. Hypermutation pattern only in the ancestral sequence.	ACTGATACT	ACTACTACT	0, 0, 1, 1	1	match
18. Hypermutation pattern only in the query sequence.	ACTGCTACT	ACTAAAACT	1, 1, 0, 0	1	match
19. More control mutations than hypermutations.	GAGAGAGAGAGAGC GCGCGCGCGCGC	GAGAGAGAGAGAAC ACACACACACAC	0, 6, 6, 6	1	match
20. More hypermutation mutations than control mutations.	GAGAGAGAGAGAGC GCGCGCGCGCGC	AAAAAAAAAAAAGC GCGCGCGCGCGC	6, 6, 0, 6	0.0011	match
21. Overlapping control positions in the ancestral sequence.	ACTGGCACT	ACTAACACT	0, 0, 2, 2	1	match
22. Overlapping hypermutation positions in the ancestral sequence.	ACTGGAACT	ACTAAAACT	2, 2, 0, 0	1	match
23. The alignment is of length 2.	GA	AA	0, 0, 0, 0	1	match

24. The alignment is of length 3 with hypermutation.	GAA	AAA	1, 1, 0, 0	1	match
25. The alignment is of length 3 without hypermutation.	CAA	AAA	0, 0, 0, 0	1	match
26. The alignment is of length 4 with hypermutation.	CGAA	CAAA	1, 1, 0, 0	1	match
27. The alignment is of length 4 without hypermutation.	ACAA	AGAA	0, 0, 0, 0	1	match
28. The ancestral sequence ends with gaps.	ACTGCTGAAA--	ACTACTAAAACT	1, 1, 1, 1	1	match
29. The ancestral sequence starts with gaps.	--TGCTGAAACT	ACTACTAAAACT	1, 1, 1, 1	1	match
30. The query sequence ends with gaps.	ACTGCTGAAACT	ACTACTAAAA--	1, 1, 1, 1	1	match
31. The query sequence starts with gaps.	ACTGCTGAAACT	--TACTAAAACT	1, 1, 1, 1	1	match

1.1.7 Future Work and Conclusions

The `hypermutR` package is a high quality implementation of the `Hypermut 2.0` algorithm that can be used offline. It 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. In this case, if a control position (GY or GRC) is deleted in the query sequence so that it becomes –Y or –RC, then hypermutR will tally this as an unmutated control position, but LANL will not report it as a control position (mutated or unmutated). This behavior was left in hypermutR, since this is consistent with the behavior if the position is a hypermutation position. In both the LANL and hypermutR implementations, when GRD becomes –RD, it is considered as an unmutated hypermutation position. When performing the test to determine if the sequence is hypermutated, the percentage of control positions that mutated is compared to the percentage hypermutation positions that mutated. By keeping this discrepancy with LANL's handling of this case in hypermutR, we make the processes that calculate these two ratios more consistent with each other.

Currently, `hypermutR` lacks the ability to specify custom downstream patterns for classifying a position into either a hypermutation or control position. The data formats implemented in the popular Biostrings package (Pages et al., 2017) are not supported. A more sophisticated consensus sequence generation approach should be implemented that can handle IUPAC sequences and that has some more nuanced options for dealing with positions with many A's and G's.