abtools Documentation

Release 0.1.1

Bryan Briney

May 18, 2017

Contents

AbTools provides core data structures and methods for analysis of antibody repertoire data. Tools for manipulating sequences, pairwise and multiple sequence alignment and sequence clustering are included.

Additionally, AbTools provides utilities for secondary analysis of antibody sequence data. Error correction, mining NGS datasets for sequences with similarity to known antibody sequences, generating lineage phylogenies, and making repertoire-level comparisons are all covered.

CHAPTER 1

Getting Started

Install

The easiest way to install AbTools locally (on OSX or Linux) is to use pip:

```
$ pip install abtools
```

If you don't have pip, the Anaconda Python distribution contains pip along with a ton of useful scientific Python packages and is a great way to get started with Python.

AbTools does not run natively on Windows, but Windows users can run AbTools with Docker (AbTools is included in the AbStar Docker image):

```
$ docker pull briney/abstar
$ docker run -it briney/abstar
```

Stable and development versions of AbTools can also be downloaded from Github. You can manually install the latest development version of AbTools with:

```
$ git clone https://github.com/briney/abtools
$ cd abtools/
$ python setup.py install
```

Note: If installing manually via setup.py and you don't already have scikit-bio installed, you may get an error when setuptools attempts to install scikit-bio. This can be fixed by first installing scikit-bio with pip:

```
$ pip install scikit-bio
```

and then retrying the manual install of AbTools.

Requirements

- Python 2.7.x (Python 3 compatability is in the works)
- biopython
- celery
- ete2
- matplotlib
- pandas
- pymongo
- scikit bio
- seaborn

Additional dependencies

Several AbTools components have additional non-Python dependencies:

- `abtools.alignment` requires MAFFT and MUSCLE
- `abtools.correct` requires CDHIT and USEARCH
- `abtools.mongodb` requires MongoDB
- `abtools.phylogeny` requires MUSCLE and FastTree
- `abtools.s3` requires s3cmd

If using Docker, all of the the non-Python dependencies are included.

CHAPTER 2

Commandline Use

AbCompare

Overview

AbCompare is used to perform repertoire-level comparison of antibody sequence data using a variety of similarity and divergence measures. Currently, AbCompare compares samples using the frequency of V-gene use, although other comparison types (such as clonality) are planned. However, the underlying similarity and divergence functions are accessible via the AbTools API, so you can compare samples using other characteristics.

Similarity (or divergence) scores are computed by subsampling each dataset and computing the score for the subsamples. This process is repeated many times, and the median score for all of the iterations is returned. In addition to producing a more accurate representation of the true score, it also makes is possible to directly compare datasets of different sizes.

Examples

To compute the Marisita-Horn similarity of two collections, both in the same MongoDB database:

```
$ abcompare -d MyDatabase -1 Collection1 -2 Collection2 -o /path/to/output/
```

If only one collection is provided (via `-1`), then that collection will be iteratively compared to all other collections in the database:

```
$ abcompare -d MyDatabase -1 Collection1 -o /path/to/output/
```

If you leave out collections entirely, all collections in the database will be iteratively compared to all other collections:

```
$ abcompare -d MyDatabase -o /path/to/output/
```

Finally, if you'd like to compare only those collections that share a common prefix (for example, if your collection names are formatted as `SubjectName_Timepoint` and you'd like to compare all the timepoints from a single subject):

```
$ abcompare -d MyDatabase --collection-prefix SubjectName -o /path/to/output/
```

The default comparison method is Marisita-Horn similiarity, but several other methods are provided:

- Marisita-Horn similarity (`'marisita-horn'`)
- Kullback-Leibler divergence (`'kullback-leibler'`)
- Jensen-Shannon similarity (`'jensen-shannon'`)
- Jaccard similarity (`'jaccard'`)
- Bray-Curtis similarity (`'bray-curtis'`)
- Renkonen similarity (`'renkonen'`)
- Cosine similarity (`'cosine'`)

To use an alternate comparison method, pass the method with the `--method` option:

```
$ abcompare -d MyDatabase -o /path/to/output/ --method jaccard
```

The number of sequences used in each iteration (`--chunksize`, default is 100,000) and the number of iterations (`--iterations`, default is 10,000) can also be changed:

```
$ abcompare -d MyDatabase -o /path/to/output --iterations 1000 --chunksize 25000
```

As with other AbTools applications, there are options for connecting to remote MongoDB servers (`--ip` and `--port`) and MongoDB authentication (`--user` and `--password`). A complete list of AbCompare options can be obtained by:

```
$ abcompare --help
```

AbCorrect

Overview

AbCorrect is a full-featured utility for performing error-correction on antibody repertoire sequencing data. Error correction can be performed using Unique Antibody IDs (UAIDs; also known as molecular barcodes) or by identity clustering.

The AbCorrect command-line application is designed to work with antibody sequence data that has already been annotated with AbStar. Although other error correction tools for antibody repertoire sequencing operate on raw data (or, in the case of paired reads, on raw data after read merging), we have found that annotating the sequences with AbStar before error correction allows us to focus clustering and consensus/centroid generation on just the VDJ region of the antibody sequence in the proper orientation. In our hands, this tends to produce more accurate, reproducible results.

In addition to being provided as a command-line application, the core functionality can be accessed through the *AbTools API*, which allows AbCorrect to be integrated into more sophisticated sequence processing pipelines. Below are several examples showing how to use AbCorrect as a command-line application.

Examples

The simplest use case for AbCorrect is to perform error correction on a single JSON file, which is the output from running AbStar on a single FASTA/Q file:

```
$ abcorrect -j /path/to/MyData.json -t /path/to/temp/ -o /path/to/output/
```

This will cluster sequences based on UAIDs (which have been pre-parsed by AbStar) and generate a 'consensus' sequence for each UAID cluster containing at least one sequence ('consensus' is in quotes because it isn't truly a consensus for UAID bins with a single sequence). Output will be a single FASTA file of error-corrected sequences, located at `/path/to/output/MyData.fasta`.

To perform the same operation, but only calculate consensus sequences for UAID clusters with at least 3 sequences:

```
$ abcorrect -j /path/to/MyData.json -t /path/to/temp/ -o /path/to/output/ --min 3
```

If you want to correct errors using UAIDs but you forgot to have AbStar parse them, you can have AbCorrect parse them by passing the length of the barcode (in nucleotides):

```
$ abcorrect -j /path/to/MyData.json -t /path/to/temp/ -o /path/to/output/ --parse-
↪uaids 20 --min 3
```

This will use the first 20nt of the raw merged read as the UAID. If the UAID is at the end of the read (for paired reads, this would be the start of R2), use a negative number for `--parse-uaids`.

To cover the relatively rare case (assuming the UAID length was selected appropriately) where two sequences were tagged with the same barcode, AbCorrect clusters the sequences within each UAID bin and builds a consensus/centroid sequence for each subcluster that passes the `--min` size threshold. To disable this, you can pass the `--largest-cluster-only` option and AbCorrect will only build a consensus/centroid sequence for the largest cluster within each UAID bin.

To perform error-correction using identity clustering instead of UAIDs, you can:

```
$ abcorrect -j /path/to/MyData.json -t /path/to/temp/ -o /path/to/output/ --no-uaids
```

This will cluster the sequences at an identity threshold (default is 0.975, or 97.5% identity) and build a consensus sequemce for each cluster. To cluster with a threshold of 0.96 instead:

```
$ abcorrect -j /path/to/MyData.json -t /path/to/temp/ -o /path/to/output/ -I 0.96 --
↪no-uaids
```

If you have more than one JSON file to be processed, you can pass AbCorrect a directory that contains one or more JSON files and each JSON file will be iteratively processed:

```
$ abcorrect -j /path/to/JSONs/ -t /path/to/temp/ -o /path/to/output/
```

All of the other options (such as the minimum number of sequences for consensus/centroid calculation) remain, although there is currently no way to specifiy different options for each JSON file.

If your AbStar-annotated sequences have already been uploaded to MongoDB, you can still use AbCorrect to perform error correction. Rather than passing JSON files with `-j`, you can pass a MongoDB database name with `-d` and a collection name with `-c`:

```
$ abcorrect -d MyDatabase -c MyCollection -t /path/to/temp/ -o /path/to/output/
```

If you supply just the database name (without a collection), AbCorrect will iteratively process all collections in the supplied database:

```
$ abcorrect -d MyDatabase -t /path/to/temp/ -o /path/to/output/
```

The above example is querying MyDatabase on your local instance of MongoDB. To do the same thing on a remote MongoDB server, you can pass the IP address with `-i` (assuming the default port of `27017`:

```
$ abcorrect -d MyDatabase -i 123.45.67.89 -t /path/to/temp/ -o /path/to/output/
```

If your MongoDB server uses a port other than `27017`, you can provide it using the `--port` option. And if your remote MongoDB server requires authentication, you can supply the username with `--user` and the password with `--password`. If you don't supply both `--user` and `--password`, AbCorrect will attempt to connect to the MongoDB database without authentication.

Finally, to make non-redundant set of sequences, AbCorrect provides the `--nr` option:

```
$ abcorrect -d MyDatabase -t /path/to/temp/ -o /path/to/output/ --nr
```

This uses `sort | uniq`, which is much faster than clustering at 100% identity with CD-HIT.

Warning: Using `--nr` is is not the same as clustering at 100% identity. Two sequences that are different lengths but are otherwise identical will be collapsed when clustering with CD-HIT but will not be collapsed when using `sort | uniq`.

AbFinder

Overview

AbFinder provides methods to mine large datasets of antibody sequences to rapidly identify sequences with high identity to known antibody sequences.

Given a MongoDB database and collection, AbFinder computes identity between one or more 'standard' sequences and all seqeunces in the collection. Default output is an identity/divergence plot, a hexbin plot of germline identity (X-axis) and identity to the standard sequence (Y-axis). AbFinder also updates MongoDB with identity information so that standard identities can be used in subsequent queries.

Examples

To run, AbFinder needs a MongoDB database and collection, an output directory, and a FASTA-formatted file of standard sequences:

```
$ abfinder -d MyDatabase -c MyCollection -s standards.fasta -o /path/to/output/
```

Omitting the collection results in AbFinder iteratively processing each collection in the database. By default, AbFinder assumes that the standard file contains amino acid sequences. If you would like to compute nucleotide identity instead, you can indicate your preference with the `--nucleotide` option:

```
$ abfinder -d MyDatabase -s standards.fasta -o /path/to/output/ --nucleotide
```

AbFinder also assumes that the standard file contains heavy chain sequences, and only heavy chain sequences from MongoDB will be used for comparison. To compare sequences of a different chain (options are `'heavy'`, `'kappa'`, and `'lambda'`), use the `--chain` option:

```
$ abfinder -d MyDatabase -s standards.fasta -o /path/to/output/ --chain kappa
```

If you do not plan on using the identity scores for any sort of downstream analysis, you can save some time and skip the MongoDB updates and just make the identity/divergence figures:

```
$ abfinder -d MyDatabase -s standards.fasta -o /path/to/output/ --no-update
```

There are several other options, mainly related to formatting the identity/divergence figures. A complete list of all options can be obtained with:

```
$ abfinder --help
```

AbPhylogeny

Overview

AbPhylogeny generates figure-quality phylogenetic trees from antibody sequence data. Designed with the ability to color individual sequences by attribute, phylogenetic trees can be drawn that accurately represent data from longitugindal samplings, different sampling locations (peripheral blood, bone marrow, FNA, etc), or categorical genetic characteristics like isotype.

AbPhylogeny can take input on any of three levels:

1. FASTA-formatted sequence files
2. FASTA-formatted multiple sequence alignment
3. Newick-formatted tree file

If given sequence files, AbPhylogeny will perform multiple sequence alignment with MUSCLE, build a tree file from the alignment with FastTree, and draw the tree figure. If given an alignment, AbPhylogeny will build the tree file and draw the figure. If given a tree file, AbPhylogeny will simply draw the figure. In each case, AbPhylogeny will save all intermediate files to the output directory, so intermediates can be used to speed up multiple iterations on the same figure. This is especially helpful when trying multiple variations (colors, fontsizes, etc) of the same figure.

Examples

CHAPTER 3

API

API Examples

API Reference

Core Utilities

abtools.alignment: Pairwise and Multiple Sequence Alignment

abtools.alignment.**mafft**(*sequences=None*, *alignment_file=None*, *fasta=None*, *fmt='fasta'*, *threads=-1*, *as_file=False*, *print_stdout=False*, *print_stderr=False*)

Performs multiple sequence alignment with MAFFT.

MAFFT is a required dependency.

Parameters

- **sequences** (*list*) – Sequences to be aligned. `sequences` can be one of four things:
  1. a FASTA-formatted string
  2. a list of BioPython `SeqRecord` objects
  3. a list of AbTools `Sequence` objects
  4. a list of lists/tuples, of the format `[sequence_id, sequence]`
- **alignment_file** (*str*) – Path for the output alignment file. If not supplied, a name will be generated using `tempfile.NamedTemporaryFile()`.
- **fasta** (*str*) – Path to a FASTA-formatted file of sequences. Used as an alternative to `sequences` when suppling a FASTA file.
- **fmt** (*str*) – Format of the alignment. Options are 'fasta' and 'clustal'. Default is 'fasta'.
- **threads** (*int*) – Number of threads for MAFFT to use. Default is `-1`, which results in MAFFT using `multiprocessing.cpu_count()` threads.

- **as_file** (*bool*) – If `True`, returns a path to the alignment file. If `False`, returns a BioPython `MultipleSeqAlignment` object (obtained by calling `Bio.AlignIO.read()` on the alignment file).

Returns

Returns a BioPython `MultipleSeqAlignment` object, unless `as_file` is `True`, in which case the path to the alignment file is returned.

abtools.alignment.**muscle**(*sequences=None*, *alignment_file=None*, *fasta=None*, *fmt='fasta'*, *as_file=False*, *maxiters=None*, *diags=False*, *gap_open=None*, *gap_extend=None*)

Performs multiple sequence alignment with MUSCLE.

MUSCLE is a required dependency.

Parameters

- **sequences** (*list*) – Sequences to be aligned. `sequences` can be one of four things:
  1. a FASTA-formatted string
  2. a list of BioPython `SeqRecord` objects
  3. a list of AbTools `Sequence` objects
  4. a list of lists/tuples, of the format `[sequence_id, sequence]`
- **alignment_file** (*str*) – Path for the output alignment file. If not supplied, a name will be generated using `tempfile.NamedTemporaryFile()`.
- **fasta** (*str*) – Path to a FASTA-formatted file of sequences. Used as an alternative to `sequences` when suppling a FASTA file.
- **fmt** (*str*) – Format of the alignment. Options are 'fasta' and 'clustal'. Default is 'fasta'.
- **threads** (*int*) – Number of threads for MAFFT to use. Default is `-1`, which results in MAFFT using `multiprocessing.cpu_count()` threads.
- **as_file** (*bool*) – If `True`, returns a path to the alignment file. If `False`, returns a BioPython `MultipleSeqAlignment` object (obtained by calling `Bio.AlignIO.read()` on the alignment file).
- **maxiters** (*int*) – Passed directly to MUSCLE using the `-maxiters` flag.
- **diags** (*int*) – Passed directly to MUSCLE using the `-diags` flag.
- **gap_open** (*float*) – Passed directly to MUSCLE using the `-gapopen` flag. Ignored if `gap_extend` is not also provided.
- **gap_extend** (*float*) – Passed directly to MUSCLE using the `-gapextend` flag. Ignored if `gap_open` is not also provided.

Returns

Returns a BioPython `MultipleSeqAlignment` object, unless `as_file` is `True`, in which case the path to the alignment file is returned.

abtools.alignment.**local_alignment**(*query*, *target=None*, *targets=None*, *match=3*, *mismatch=-2*, *gap_open=-5*, *gap_extend=-2*, *matrix=None*, *aa=False*, *gap_open_penalty=None*, *gap_extend_penalty=None*)

Striped Smith-Waterman local pairwise alignment.

Parameters

- **query** – Query sequence. `query` can be one of four things:

1. a nucleotide or amino acid sequence, as a string
2. a Biopython `SeqRecord` object
3. an AbTools `Sequence` object
4. a list/tuple of the format `[seq_id, sequence]`

- **target** – A single target sequence. `target` can be anything that `query` accepts.
- **targets** (*list*) – A list of target sequences, to be proccssed iteratively. Each element in the `targets` list can be anything accepted by `query`.
- **match** (*int*) – Match score. Should be a positive integer. Default is 3.
- **mismatch** (*int*) – Mismatch score. Should be a negative integer. Default is -2.
- **gap_open** (*int*) – Penalty for opening gaps. Should be a negative integer. Default is -5.
- **gap_extend** (*int*) – Penalty for extending gaps. Should be a negative integer. Default is -2.
- **matrix** (*str, dict*) – Alignment scoring matrix. Two options for passing the alignment matrix:
  - The name of a built-in matrix. Current options are `blosum62` and `pam250`.
  - A nested dictionary, giving an alignment score for each residue pair. Should be formatted such that retrieving the alignment score for A and G is accomplished by:

```
matrix['A']['G']
```

- **aa** (*bool*) – Must be set to `True` if aligning amino acid sequences. Default is `False`.

Returns If a single target sequence is provided (via `target`), a single `SSWAlignment` object will be returned. If multiple target sequences are supplied (via `targets`), a list of `SSWAlignment` objects will be returned.

`abtools.alignment.`**global_alignment**(*query*, *target=None*, *targets=None*, *match=3*, *mismatch=-2*, *gap_open=-5*, *gap_extend=-2*, *score_match=None*, *score_mismatch=None*, *score_gap_open=None*, *score_gap_extend=None*, *matrix=None*, *aa=False*)

Needleman-Wunch global pairwise alignment.

With `global_alignment`, you can score an alignment using different paramaters than were used to compute the alignment. This allows you to compute pure identity scores (match=1, mismatch=0) on pairs of sequences for which those alignment parameters would be unsuitable. For example:

```
seq1 = 'ATGCAGC'
seq2 = 'ATCAAGC'
```

using identity scoring params (match=1, all penalties are 0) for both alignment and scoring produces the following alignment:

```
ATGCA-GC
|| || ||
AT-CAAGC
```

with an alignment score of 6 and an alignment length of 8 (identity = 75%). But what if we want to calculate the identity of a gapless alignment? Using:

```
global_alignment(seq1, seq2,
                 gap_open=20,
                 score_match=1,
                 score_mismatch=0,
                 score_gap_open=10,
                 score_gap_extend=1)
```

we get the following alignment:

```
ATGCAGC
||  |||
ATCAAGC
```

which has an score of 5 and an alignment length of 7 (identity = 71%). Obviously, this is an overly simple example (it would be much easier to force gapless alignment by just iterating over each sequence and counting the matches), but there are several real-life cases in which different alignment and scoring paramaters are desirable.

Parameters

- **query** – Query sequence. `query` can be one of four things:
  1. a nucleotide or amino acid sequence, as a string
  2. a Biopython `SeqRecord` object
  3. an AbTools `Sequence` object
  4. a list/tuple of the format `[seq_id, sequence]`
- **target** – A single target sequence. `target` can be anything that `query` accepts.
- **targets** (*list*) – A list of target sequences, to be proccssed iteratively. Each element in the `targets` list can be anything accepted by `query`.
- **match** (*int*) – Match score for alignment. Should be a positive integer. Default is 3.
- **mismatch** (*int*) – Mismatch score for alignment. Should be a negative integer. Default is -2.
- **gap_open** (*int*) – Penalty for opening gaps in alignment. Should be a negative integer. Default is -5.
- **gap_extend** (*int*) – Penalty for extending gaps in alignment. Should be a negative integer. Default is -2.
- **score_match** (*int*) – Match score for scoring the alignment. Should be a positive integer. Default is to use the score from `match` or `matrix`, whichever is provided.
- **score_mismatch** (*int*) – Mismatch score for scoring the alignment. Should be a negative integer. Default is to use the score from `mismatch` or `matrix`, whichever is provided.
- **score_gap_open** (*int*) – Gap open penalty for scoring the alignment. Should be a negative integer. Default is to use `gap_open`.
- **score_gap_extend** (*int*) – Gap extend penalty for scoring the alignment. Should be a negative integer. Default is to use `gap_extend`.
- **matrix** (*str, dict*) – Alignment scoring matrix. Two options for passing the alignment matrix:
  - The name of a built-in matrix. Current options are `blosum62` and `pam250`.
  - A nested dictionary, giving an alignment score for each residue pair. Should be formatted such that retrieving the alignment score for A and G is accomplished by:

```
matrix['A']['G']
```

- **aa** (*bool*) – Must be set to `True` if aligning amino acid sequences. Default is `False`.

Returns If a single target sequence is provided (via `target`), a single `NWAlignment` object will be returned. If multiple target sequences are supplied (via `targets`), a list of `NWAlignment` objects will be returned.

class `abtools.alignment.`**BaseAlignment**(*query*, *target*, *matrix*, *match*, *mismatch*, *gap_open*, *gap_extend*, *aa*)

Base class for local and global pairwise alignments.

Note: All comparisons between `BaseAlignments` are done on the `score` attribute (which must be implemented by any classes that subclass `BaseAlignment`). This was done so that sorting alignments like so:

```
alignments = [list of alignments]
alignments.sort(reverse=True)
```

results in a sorted list of alignments from the highest alignment score to the lowest.

---

query
Sequence – The input query sequence, as an AbTools `Sequence` object.

target
Sequence – The input target sequence, as an AbTools `Sequence` object.

target_id
str – ID of the target sequence.

raw_query
The raw query, before conversion to a `Sequence`.

raw_target
The raw target, before conversion to a `Sequence`.

class `abtools.alignment.`**SSWAlignment**(*query*, *target*, *match=3*, *mismatch=-2*, *matrix=None*, *gap_open=5*, *gap_extend=2*, *aa=False*)

Structure for performing and analyzing a Smith-Waterman local alignment.

alignment_type
str – Is 'local' for all `SSWAlignment` objects.

aligned_query
str – The aligned query sequence (including gaps).

aligned_target
str – The aligned target sequence (including gaps).

alignment_midline
str – Midline for the aligned sequences, with `|` indicating matches and a gap indicating mismatches:

```
print(aln.aligned_query)
print(aln.alignment_midline)
print(aln.aligned_target)

# ATGC
# || |
# ATCC
```

score
int – Alignment score.

query_begin
int – Position in the raw query sequence at which the optimal alignment begins.

query_end
int – Position in the raw query sequence at which the optimal alignment ends.

target_begin
int – Position in the raw target sequence at which the optimal alignment begins.

target_end
int – Position in the raw target sequence at which the optimal alignment ends.

class abtools.alignment.**NWAlignment**(*query*, *target*, *match=3*, *mismatch=-2*, *gap_open=-5*, *gap_extend=-2*, *score_match=None*, *score_mismatch=None*, *score_gap_open=None*, *score_gap_extend=None*, *matrix=None*, *aa=False*)
Structure for performing and analyzing a Needleman-Wunch global alignment.

alignment_type
str – Is 'global' for all `NWAlignment` objects.

aligned_query
str – The aligned query sequence (including gaps).

aligned_target
str – The aligned target sequence (including gaps).

alignment_midline
str – Midline for the aligned sequences, with `|` indicating matches and a gap indicating mismatches:

```
print(aln.aligned_query)
print(aln.alignment_midline)
print(aln.aligned_target)

# ATGC
# || |
# ATCC
```

score
int – Alignment score.

query_begin
int – Position in the raw query sequence at which the optimal alignment begins.

query_end
int – Position in the raw query sequence at which the optimal alignment ends.

target_begin
int – Position in the raw target sequence at which the optimal alignment begins.

target_end
int – Position in the raw target sequence at which the optimal alignment ends.

abtools.cluster: Sequence Clustering

class abtools.cluster.**Cluster**(*raw_cluster*, *seq_db=None*, *db_path=None*, *seq_dict=None*)
Data and methods for a cluster of sequences.

All public attributes are evaluated lazily, so attributes that require significant processing time are only computed when needed. In addition, attributes are only calculated once, so if you change the Cluster object after accessing attributes, the attributes will not update. Setters are provided for all attributes, however, so you can update them manually if necessary:

```
seqs = [Sequence1, Sequence2, ... SequenceN]
clust = cluster(seqs)

# calculate the consensus
consensus = clust.consensus

# add sequences to the Cluster
more_sequences = [SequenceA, SequenceB, SequenceC]
clust.sequences += more_sequences

# need to recompute the consensus manually
clust.consensus = clust._make_consensus()
```

ids
list – A list of all sequence IDs in the Cluster

size
int – Number of sequences in the Cluster

sequences
list – A list of all sequences in the Cluster, as AbTools `Sequence` objects.

consensus
Sequence – Consensus sequence, calculated by aligning all sequences with MAFFT and computing the `Bio.Align.AlignInfo.SummaryInfo.gap_consensus()`

centroid
Sequence – Centroid sequence, as calculated by CD-HIT.

abtools.cluster.**cluster**(*seqs*, *threshold=0.975*, *out_file=None*, *make_db=True*, *temp_dir=None*, *quiet=False*, *threads=0*, *return_just_seq_ids=False*, *max_memory=800*, *debug=False*)

Perform sequence clustering with CD-HIT.

Parameters

- **seqs** (*list*) – An iterable of sequences, in any format that abtools.sequence.Sequence() can handle
- **threshold** (*float*) – Clustering identity threshold. Default is 0.975.
- **out_file** (*str*) – Path to the clustering output file. Default is to use tempfile.NamedTempraryFile to generate an output file name.
- **temp_dir** (*str*) – Path to the temporary directory. If not provided, '/tmp' is used.
- **make_db** (*bool*) – Whether to build a SQlite database of sequence information. Required if you want to calculate consensus/centroid sequences for the resulting clusters or if you need to access the clustered sequences (not just sequence IDs) Default is True.

Returns A list of Cluster objects, one per cluster.

Return type list

abtools.log: Logging

abtools.log.**setup_logging**(*logfile*, *print_log_location=True*, *debug=False*)
Set up logging using the built-in `logging` package.

A stream handler is added to all logs, so that logs at or above `logging.INFO` level are printed to screen as well as written to the log file.

Parameters

- **logfile** (*str*) – Path to the log file. If the parent directory does not exist, it will be created. Required.
- **print_log_location** (*bool*) – If `True`, the log path will be written to the log upon initialization. Default is `True`.
- **debug** (*bool*) – If true, the log level will be set to `logging.DEBUG`. If `False`, the log level will be set to `logging.INFO`. Default is `False`.

abtools.log.**get_logger**(*name=None*)
Get a logging handle.

As with `setup_logging`, a stream handler is added to the log handle.

Parameters **name** (*str*) – Name of the log handle. Default is `None`.

abtools.pipeline: Utilities for building pipelines of AbTools functions

abtools.pipeline.**initialize**(*log_file*, *project_dir=None*, *debug=False*)
Initializes an AbTools pipeline.

Initialization includes printing the AbTools splash, setting up logging, creating the project directory, and logging both the project directory and the log location.

Parameters

- **log_file** (*str*) – Path to the log file. Required.
- **project_dir** (*str*) – Path to the project directory. If not provided, the project directory won't be created and the location won't be logged.
- **debug** (*bool*) – If `True`, the logging level will be set to `logging.DEBUG`. Default is `FALSE`, which logs at `logging.INFO`.

Returns logger

abtools.pipeline.**make_dir**(*d*)
Makes a directory, if it doesn't already exist.

Parameters **d** (*str*) – Path to a directory.

abtools.pipeline.**list_files**(*d*, *extension=None*)
Lists files in a given directory.

Parameters

- **d** (*str*) – Path to a directory.
- **extension** (*str*) – If supplied, only files that contain the specificied extension will be returned. Default is `False`, which returns all files in `d`.

Returns A sorted list of file paths.

Return type list

abtools.s3: Backup data to S3

abtools.s3.**compress_and_upload**(*data*, *compressed_file*, *s3_path*, *multipart_chunk_size_mb=500*, *method='gz'*, *delete=False*, *access_key=None*, *secret_key=None*)

Compresses data and uploads to S3.

S3 upload uses `s3cmd`, so you must either:

1. Manually configure `s3cmd` prior to use (typically using `s3cmd --configure`).
2. Configure `s3cmd` using `s3.configure()`.
3. Pass your access key and secret key to `compress_and_upload`, which will automatically configure s3cmd.

Parameters

- **data** – Can be one of three things:
  1. Path to a single file
  2. Path to a directory
  3. A list of one or more paths to files or directories
- **compressed_file** (*str*) – Path to the compressed file. Required.
- **s3_path** (*str*) – The S3 path, with the filename omitted. The S3 filename will be the basename of the `compressed_file`. For example:

```
compress_and_upload(data='/path/to/data',
                    compressed_file='/path/to/compressed.tar.gz',
                    s3_path='s3://my_bucket/path/to/')
```

  will result in an uploaded S3 path of `s3://my_bucket/path/to/compressed.tar.gz`
- **method** (*str*) – Compression method. Options are `'gz'` (gzip) or `'bz2'` (bzip2). Default is `'gz'`.
- **delete** (*bool*) – If `True`, the `compressed_file` will be deleted after upload to S3. Default is `False`.
- **access_key** (*str*) – AWS access key.
- **secret_key** (*str*) – AWS secret key.

abtools.s3.**put**(*f*, *s3_path*, *multipart_chunk_size_mb=500*, *logger=None*)

Uploads a single file to S3, using s3cmd.

Parameters

- **f** (*str*) – Path to a single file.
- **s3_path** (*str*) – The S3 path, with the filename omitted. The S3 filename will be the basename of the `f`. For example:

```
put(f='/path/to/myfile.tar.gz', s3_path='s3://my_bucket/path/to/')
```

  will result in an uploaded S3 path of `s3://my_bucket/path/to/myfile.tar.gz`

abtools.s3.**compress**(*d*, *output*, *compress='gz'*, *logger=None*)

Creates a compressed/uncompressed tar file.

Parameters

- **d** – Can be one of three things:
  1. the path to a single file, as a string
  2. the path to a single directory, as a string
  3. an iterable of file or directory paths
- **output** (*str*) – Output file path.
- **compress** – Compression method. Options are `'gz'` (gzip), `'bz2'` (bzip2) and `'none'` (uncompressed). Default is `'gz'`.

`abtools.s3.`**configure**(*access_key=None*, *secret_key=None*, *logger=None*)

Configures s3cmd prior to first use.

If no arguments are provided, you will be prompted to enter the access key and secret key interactively.

Parameters

- **access_key** (*str*) – AWS access key
- **secret_key** (*str*) – AWS secret key

`abtools.sequence`: Sequence utilities

class `abtools.sequence.`**Sequence**(*seq*, *id=None*, *qual=None*, *id_key='seq_id'*, *seq_key='vdj_nt'*)

Container for biological (RNA and DNA) sequences.

`seq` can be one of several things:

1. a raw sequence, as a string
2. an iterable, formatted as `[seq_id, sequence]`
3. a dict, containing at least the ID (default key = 'seq_id') and a sequence (default key = 'vdj_nt'). Alternate `id_key` and `seq_key` can be provided at instantiation.
4. a Biopython `SeqRecord` object
5. an AbTools `Sequence` object

If `seq` is provided as a string, the sequence ID can optionally be provided via `id`. If `seq` is a string and `id` is not provided, a random sequence ID will be generated with `uuid.uuid4()`.

Quality scores can be supplied with `qual` or as part of a `SeqRecord` object. If providing both a SeqRecord object with quality scores and quality scores via `qual`, the `qual` scores will override the SeqRecord quality scores.

If `seq` is a dictionary, typically the result of a MongoDB query, the dictionary can be accessed directly from the `Sequence` instance. To retrive the value for `'junc_aa'` in the instantiating dictionary, you would simply:

```
s = Sequence(dict)
junc = s['junc_aa']
```

If `seq` is a dictionary, an optional `id_key` and `seq_key` can be provided, which tells the `Sequence` object which field to use to populate `Sequence.id` and `Sequence.sequence`. Defaults are `id_key='seq_id'` and `seq_key='vdj_nt'`.

Alternately, the `__getitem__()` interface can be used to obtain a slice from the `sequence` attribute. An example of the distinction:

```
d = {'name': 'MySequence', 'sequence': 'ATGC'}
seq = Sequence(d, id_key='name', seq_key='sequence')

seq['name']  # 'MySequence'
seq[:2]  # 'AT'
```

If the `Sequence` is instantiated with a dictionary, calls to `__contains__()` will return `True` if the supplied item is a key in the dictionary. In non-dict instantiations, `__contains__()` will look in the `Sequence.sequence` field directly (essentially a motif search). For example:

```
dict_seq = Sequence({'seq_id': 'seq1', 'vdj_nt': 'ACGT'})
'seq_id' in dict_seq  # TRUE
'ACG' in dict_seq     # FALSE

str_seq = Sequence('ACGT', id='seq1')
'seq_id' in str_seq  # FALSE
'ACG' in str_seq     # TRUE
```

Note: When comparing `Sequence` objects, they are comsidered equal only if their sequences and IDs are identical. This means that two `Sequence` objects with identical sequences but without user-supplied IDs won't be equal, because their IDs will have been randomly generated.

---

fasta
str – Returns the sequence, as a FASTA-formatted string

Note: The FASTA string is built using `Sequence.id` and `Sequence.sequence`.

fastq
str – Returns the sequence, as a FASTQ-formatted string

If `Sequence.qual` is `None`, then `None` will be returned instead of a FASTQ string

reverse_complement
str – Returns the reverse complement of `Sequence.sequence`.

region(*start=0*, *end=None*)
Returns a region of `Sequence.sequence`, in FASTA format.

If called without kwargs, the entire sequence will be returned.

Parameters

- **start** (*int*) – Start position of the region to be returned. Default is 0.
- **end** (*int*) – End position of the region to be returned. Negative values will function as they do when slicing strings.

Returns A region of `Sequence.sequence`, in FASTA format

Return type str

Secondary Annotation

abtools.compare: Repertoire-level comparison

abtools._compare.**aggregate**(*data*)
Counts the number of occurances of each item in 'data'.

Input data: a list of values.

Output a dict of bins and counts.

abtools._compare.**mh_similarity**(*sample1*, *sample2*)
Calculates the Marista-Horn similarity for two samples.

Parameters

- **sample1** – list of frequencies for sample 1
- **sample2** – list of frequencies for sample 2

Returns Marista-Horn similarity (between 0 and 1)

Return type float

abtools._compare.**kl_divergence**(*s1*, *s2*)
Calculates the Kullback-Leibler divergence for two samples.

Parameters

- **sample1** – probability distribution for sample 1
- **sample2** – probability distribution for sample 2

Returns Kullbeck-Leibler similarity

Return type float

abtools._compare.**js_similarity**(*s1*, *s2*)
Calculates the Jensen-Shannon similarity for two samples.

Parameters

- **sample1** – probability distribution for sample 1
- **sample2** – probability distribution for sample 2

Returns Jensen-Shannon similarity (between 0 and 1)

Return type float

abtools._compare.**shannon_entropy**(*prob_dist*)
Calculates the Shannon entropy for a single probability distribution.

Parameters **prob_dist** – probability distribution, must sum to 1

Returns Shannon entropy

Return type float

abtools._compare.**jaccard_similarity**(*s1*, *s2*)
Calculates the Jaccard similarity for two samples.

Parameters

- **sample1** – list of frequencies for sample 1
- **sample2** – list of frequencies for sample 2

Returns Jaccard similarity (between 0 and 1)

Return type float

abtools._compare.**renkonen_similarity**(*s1*, *s2*)
Calculates the Renkonen similarity (also known as the percentage similarity) for two samples.

Parameters

- **s1** – probability distribution for sample 1
- **s2** – probability distribution for sample 2

Returns Renkonen similarity (between 0 and 1)

Return type float

abtools._compare.**bc_similarity**(*s1*, *s2*)
Calculates the Bray-Curtis similarity for two samples.

Parameters

- **s1** – probability distribution for sample 1
- **s2** – probability distribution for sample 2

Returns Bray-Curtis similarity (between 0 and 1)

Return type float

abtools._compare.**cosine_similarity**(*s1*, *s2*)
Calculates the cosine (angular) similarity for two samples.

Parameters

- **s1** – list of frequencies for sample 1
- **s2** – list of frequencies for sample 2

Returns Cosine similarity (between 0 and 1)

Return type float

abtools._compare.**sd_similarity**(*s1*, *s2*)
Calculates the Brey-Curtis similarity for two samples.

Parameters

- **s1** – list of frequencies for sample 1
- **s2** – list of frequencies for sample 2

Results:

float: Brey-Curtis similarity (between 0 and 1)

abtools._compare.**run**(***kwargs*)
Performs repertoire-level comparison of antibody sequencing datasets.

Currently, the only metric for comparison is V-gene usage frequency. Additional measures are in the works (such as comparisons based on clonality).

Parameters

- **db** (*str*) – MongoDB database name.
- **collection1** (*str*) – Name of the first MongoDB collection to query for comparison. If both `collection1` and `collection2` are provided, `collection1` will be compared only to `collection2`. If neither `collection1` nor `collection2` are provided, all collections in `db` will be processed iteratively (all pairwise comparisons will be made). If `collection1` is provided but `collection2` is not, `collection1` will be iteratively compared to all other collections in `db`.
- **collection2** (*str*) – Name of the second MongoDB collection to query for comparison. If both `collection1` and `collection2` are provided, `collection1` will be

compared only to `collection2`. If neither `collection1` nor `collection2` are provided, all collections in `db` will be processed iteratively (all pairwise comparisons will be made).

- **`collection_prefix`** (*`str`*) – All collections beginning with `collection_prefix` will be iteratively compared (all pairwise comparisons will be made).
- **`ip`** (*`str`*) – IP address of the MongoDB server. Default is `localhost`.
- **`port`** (*`int`*) – Port of the MongoDB server. Default is `27017`.
- **`user`** (*`str`*) – Username with which to connect to the MongoDB database. If either of `user` or `password` is not provided, the connection to the MongoDB database will be attempted without authentication.
- **`password`** (*`str`*) – Password with which to connect to the MongoDB database. If either of `user` or `password` is not provided, the connection to the MongoDB database will be attempted without authentication.
- **`chunksize`** (*`int`*) – Number of sequences for each iteration. Default is 100,000.
- **`iterations`** (*`int`*) – Number of iterations to perform on each pair of samples. Default is 10,000
- **`method`** (*`str`*) – Similarity/divergence method to used for comparison. Default is `marisita-horn`. Options are:
  - `marisita-horn`
  - `kullback-leibler`
  - `jensen-shannon`
  - `jaccard`
  - `bray-curtis`
  - `renkonen`
  - `cosine`
- **`control_similarity`** (*`bool`*) – If `True`, control similarity/divergence will be calculated, in which each sample is also compared to itself. Default is `False`.
- **`chain`** (*`str`*) – Antibody chain to be used for comparison. Options are `heavy`, `kappa` and `lambda`. Default is `heavy`.

`abtools.correct`: PCR and sequencing error correction

`abtools.finder`: Mine NGS datasets for similarity to known mAbs

`abtools._finder.`**`chunker`**(*l*, *n*)

Generator that produces n-length chunks from iterable l.

`abtools._finder.`**`run`**(***kwargs*)

Mines NGS datasets for identity to known antibody sequences.

All of `db`, `output`, `temp` and `standard` are required.

Parameters

- **`db`** (*`str`*) – Name of a MongoDB database to query.

- **collection** (*str*) – Name of a MongoDB collection. If not provided, all collections in `db` will be processed iteratively.
- **output_dir** (*str*) – Path to the output directory, into which identity/divergence figures will be deposited.
- **temp_dir** (*str*) – Path to a temporary directory.
- **log** (*str*) – Path to a log file. If not provided, log information will not be retained.
- **ip** (*str*) – IP address of the MongoDB server. Default is `localhost`.
- **port** (*str*) – Port of the MongoDB server. Default is `27017`.
- **user** (*str*) – Username with which to connect to the MongoDB database. If either of `user` or `password` is not provided, the connection to the MongoDB database will be attempted without authentication.
- **password** (*str*) – Password with which to connect to the MongoDB database. If either of `user` or `password` is not provided, the connection to the MongoDB database will be attempted without authentication.
- **standard** (*path*) – Path to a FASTA-formatted file containing one or more 'standard' sequences, against which the NGS sequences will be compared.
- **chain** (*str*) – Antibody chain. Choices are 'heavy', 'kappa', 'lambda', and 'light'. Default is 'heavy'. Only NGS sequences matching `chain` (with 'light' covering both 'kappa' and 'lambda') will be compared to the `standard` sequences.
- **update** (*bool*) – If `True`, the MongoDB record for each NGS sequence will be updated with identity information for each standard. If `False`, the updated is skipped. Default is `True`.
- **is_aa** (*bool*) – If `True`, the `standard` sequences are amino acid sequences. If `False`, they are nucleotide seqeunces. Default is `False`.
- **x_min** (*int*) – Minimum x-axis value on identity/divergence plots.
- **x_max** (*int*) – Maximum x-axis value on identity/divergence plots.
- **y_min** (*int*) – Minimum y-axis value on identity/divergence plots.
- **y_max** (*int*) – Maximum y-axis value on identity/divergence plots.
- **gridsize** (*int*) – Relative size of hexbin grids.
- **mincount** (*int*) – Minimum number of sequences in a hexbin for the bin to be colored. Default is 3.
- **colormap** (*str, colormap*) – Colormap to be used for identity/divergence plots. Default is `Blues`.
- **debug** (*bool*) – If `True`, more verbose logging.

`abtools.phylogeny`: Phylogenetic analysis of antibody lineages

`abtools._phylogeny.`**run**(***kwargs*)

Builds a phylogenetic representation of antibody sequences.

`output` is required, as well as one of `input`, `alignment` or `newick`.

Parameters

- **input** (*str*) – Can be one of three things:

1. Path to a FASTA-formatted file containing input sequences.
2. A list of AbTools `Sequence` objects.
3. A list of dictionaries, containing at minimum `name_key` and `seq_key`.

- **output** (*str*) – Path to the output directory, into which tree images and all intermediate files will be deposited.
- **root** (*str*) – Path to a FASTA-formatted file containing a single sequence which will be used to root the tree. If not provided, tree will be unrooted.
- **mabs** (*str*) – Path to a FASTA-formatted file containing mAb sequences. If supplying both mAb sequences and NGS sequences, passing the mAb sequences separately allows you to modify their representation separately (for example, show sequence IDs for just the mAb sequences).
- **alignment** (*str*) – Path to a multiple sequence alignment, in FASTA format. If sequences are already aligned, this will save some computational time since the alignment will not be redone.
- **newick** (*str*) – Path to a tree file, in Newick format. As with `alignment`, this is primarily to save computational time if the tree file has already been generated.
- **name_key** (*str*) – If `input` is a list of Sequence objects or dicts, this key will be used to find the sequence ID. Default is `seq_id`.
- **sequence_key** (*str*) – If `input` is a list of Sequence objects or dicts, this key will be used to find the sequence. Default is `vdj_nt`.
- **timepoints** (*str*) – Path to a Tab-delimited file, of the following format (one per line):

```
TimepointName    TimepointOrder    TimepointColor
```

  `TimepointName` should prepended to the sequences in the input file (separated by `delimiter`).

  `TimepointOrder` is an integer that indicates the order in which the timepoints should be sorted.

  `TimepointColor` is a hex value that will be used to color the phylogenetic tree. If mAb sequences are provided, the 'mab' `TimepointName` will be used to sort/color the mAb sequences. If not provided, colors will be automatically selected and timepoints will be determined by a simple sort of the raw timepoint values parsed from the input file.

- **is_aa** (*bool*) – If `True`, input sequences will be assumed to be amino acid sequences. Default is `False`, which assumes nucleotide sequences.
- **delimiter** (*str*) – The delimiter used in sequence IDs to separate the timepoint from the sequence name. Default is _.
- **scale** (*int*) – Horizontal scale of the phylogeny. Default is `None`, which uses the default `ete2` value.
- **branch_vertical_margin** (*float*) – Vertical scale of the phylogeny. Default is `None`, which uses the default `ete2` value.
- **label_nodes** (*str*) – Type of nodes to be labeled. Options are: `all`, `none`, `no-root`, `mab`, `input`, and `root`.
- **label_fontsize** (*float*) – Font size for the node labels.
- **tree_orientation** (*int*) – If `0`, tree is drawn from left to right. If `1`, tree will be drawn from right to left (mirror). Default is `0`.

CHAPTER 4

About

License

News

CHAPTER 5

Related Projects

CHAPTER 6

Index

- modindex
- search

Python Module Index

a

Index

Q

R

S

T