abstar Documentation

Release 0.3.4

Bryan Briney

Jun 08, 2021

getting started

abstar is a core component of the ab[x] toolkit for antibody sequence analysis. abstar performs V(D)J germline gene assignment and primary sequence annotation and can readily scale from a single sequence to billions of sequences.

CHAPTER 1

getting started

1.1 Overview

With technical breakthroughs in the throughput and read-length of next-generation sequencing platforms, antibody repertoire sequencing is becoming an increasingly important tool for detailed characterization of the immune response to infection and immunization. Accordingly, there is a need for open, scalable software for the genetic analysis of repertoire-scale antibody sequence data.

We built abstar to be a modular component of these analyses. abstar is engineered to be highly flexible, capable of processing a single sequence or billions of sequences and scaling from individual laptops to large clusters of cloud computing instances.

1.1.1 Workflows

In addition to V(D)J germline gene assignment and primary antibody sequence annotation, abstar contains utilities for sequence acquisition, pre-processing, and database import. abstar also exposes a high-level public API to many of the core functions, which allows users to easily construct *custom analysis workflows* using multiple abstar utilities as well as other third-party tools. To ease integration of abstar into currently existing antibody analysis pipelines based on IMGT, abstar can optionally produce output that mimics the IMGT-HighV/Quest Summary output file.

1.1.2 File formats

abstar accepts standard FASTA or FASTQ files and produces, by default, JSON-formatted output. This output format allows us to build the output using data structures that match the way we process data programatically. JSON is also easily importable into NoSQL databases like MongoDB. We have found NoSQL databases to be very well suited for performing downstream analyses of antibody repertoire data, as the flexible schema allows for easy updating of sequence records with additional annotation information. Although additional data can be added to relational databases, querying this data often involves joining tables, which can require significant optimization for very large datasets.

1.1.3 Scalability

Cloud computing has dramatically changed the landscape of high-performance computing (HPC), and has allowed small academic labs to 'rent' access to computational resources that would have been previously far outside their budget. abstar is tightly integrated with abcloud, which provides tools for launching, configuring and managing clusters of compute instances on Amazon's Elastic Compute Cloud (EC2). Using the Celery distributed task queue, jobs are distributed to worker nodes and processed in parallel.

In order to maintain compatability with alternate cloud computing platforms with minimal effort, an abstar Docker image is also provided.

1.2 Install

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

```
$ pip install abstar
```

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.

abstar does not run natively on Windows, but Windows users can run abstar with Docker:

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

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

```
$ git clone https://github.com/briney/abstar
$ cd abstar/
$ 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 abstar. Starting with version 0.5, scikit-bio dropped support for Python 2.7, so install scikit-bio on Python 2.7 with:

```
$ pip install scikit-bio<=0.4.2
```

1.2.1 Requirements

- Python 2.7 or 3.5+
- abutils
- biopython
- celery
- pymongo

- pytest
- scikit bio

1.2.2 Optional dependencies

Several optional abstar components have additional dependencies:

- `abstar.preprocessing` requires FASTQC, cutadapt and sickle
- sequence merging requires PANDAseq
- downloading data from BaseSpace requires the BaseSpace Python SDK
- `batch_mongoimport` requires MongoDB

If using Docker, all of the the optional dependencies are included.

CHAPTER 2

usage

2.1 Commandline Use

Running abstar from the command line is reasonably simple, even for users with minimal experience with command-line applications. In the most basic case, with a single input file of human antibody sequences:

```
$ abstar -i /path/to/mydata.fasta -t /path/to/temp/ -o /path/to/output/
```

abstar will process all sequences contained in `mydata.fasta` and the results will be written to `/path/to/output/mydata.json`. If either (or both) of `/path/to/temp/` or `/path/to/output/` don't exist, they will be created.

If you have a directory of FASTA/Q-formatted files for abstar to process, you can pass a directory via `-i` and all files in the directory will be processed:

```
$ abstar -i /path/to/input/ -t /path/to/temp/ -o /path/to/output/
```

For input directories that contain paired FASTQ files that need to be merged prior to processing, passing the `-m` flag instructs abstar to merge paired files with PANDAseq:

```
$ abstar -i /path/to/input/ -t /path/to/temp/ -o /path/to/output/ -m
```

The merged reads will be deposited into a `merged` directory located in the parent directory of the input directory. By default, abstar will use PANDAseq's `simple_bayesian` merging algorithm, although alternate merging algorithms can be selected with `--pandaseq-algo`.

For data generated with Illumina sequencers, abstar can directly interface with BaseSpace to download raw sequencing data. In order for abstar to connect to BaseSpace, you need BaseSpace access token. The easiest way to do this is to set up a BaseSpace developer account following these instructions. Once you have your credentials, you can generate a BaseSpace credentials file by running:

```
$ make_basespace_credfile
```

and following the instructions.

When downloading data from BaseSpace, you obviously don't have an input directory of data for abstar to process (since that data hasn't been downloaded yet). Instead of providing input, output and temp directories, you can just pass abstar a project directory using `-p` and abstar will create all of the necessary subdirectories within the project directory. Running abstar with the `-b` option indicates that input data should be downloaded from BaseSpace:

```
$ abstar -p /path/to/project_dir/ -b
```

A list of available BaseSpace projects will be displayed and you can select the appropriate project. If downloading data from BaseSpace, `-m` is assumed and paired-end reads will be merged.

abstar uses a human germline database by default, but germline databases are also provided for macaque, mouse and rabbit. To process macaque antibody sequences (from BaseSpace):

```
$ abstar -p /path/to/project_dir/ -b -s macaque
```

2.2 API Examples

abstar and abutils both expose a public API containing many of the core functions. This makes it reasonably straightforward to build custom pipelines that include several abstar/abutils components or integrate these tools with third-party tools. A few simple examples are shown below.

2.2.1 Case #1

Sequencing data consists of an Illumina MiSeq run on human samples, with the raw data stored in BaseSpace (project ID: 123456789). Samples are indexed, so each sample will be downloaded from BaseSpace as a separate pair of read files. We'd like to do several things:

- get a FASTQC report on the raw data
- remove adapters
- quality trim
- get another FASTQC report on the cleaned data
- merge paired reads
- annotate with abstar

```
import os

import abstar
from abstar.utils import basespace, pandaseq

PROJECT_DIR = '/path/to/project'
PROJECT_ID = '123456789'

# download data from BaseSpace
bs_dir = os.path.join(PROJECT_DIR, 'raw_data')
basespace.download(bs_dir, project_id=PROJECT_ID)

# FASTQC on the raw data
fastqc1_dir = os.path.join(PROJECT_DIR, 'fastqc-pre')
abstar.fastqc(bs_dir, output=fastqc1_dir)

# adapter trimming
```

(continues on next page)

(continued from previous page)

```
adapter_dir = os.path.join(PROJECT_DIR, 'adapter_trimed')
adapters = '/path/to/adapters.fasta'
abstar.adapter_trim(bs_dir, output=adapter_dir, adapter_both=adapters)

# quality trimming
quality_dir = os.path.join(PROJECT_DIR, 'quality_trimed')
abstar.quality_trim(adapter_dir, output=quality_dir)

# FASTQC on the cleaned data
fastqc2_dir = os.path.join(PROJECT_DIR, 'fastqc-post')
abstar.fastqc(quality_dir, output=fastqc2_dir)

# read merging
merged_dir = os.path.join(PROJECT_DIR, 'merged')
pandaseq.run(quality_dir, merged_dir)

# run abstar
temp_dir = os.path.join(PROJECT_DIR, 'temp')
json_dir = os.path.join(PROJECT_DIR, 'json')
abstar.run(input=merged_dir,
           temp=temp_dir,
           output=json_dir)
```

2.2.2 Case #2

Sequencing data is a directory of single-read FASTQ files that have already been quality/adapter trimmed. We'd like to do the following:

- get a FASTQC report
- annotate with abstar
- import the JSONs into a MongoDB database named `MyDatabase`

Our FASTQ file names are formatted as: `SampleNumber-SampleName.fastq`, which means the abstar output file name would be `SampleNumber-SampleName.json`. We'd like the corresponding MongoDB collection to just be named `SampleName`.

```
import os

import abstar
from abstar.utils import mongoimport

PROJECT_DIR = '/path/to/project'
FASTQ_DIR = '/path/to/fastqs'

MONGO_IP = '123.45.67.89'
MONGO_PORT = 27017
MONGO_USER = 'MyUsername'
MONGO_PASS = 'Secr3t'

# FASTQC on the input data
fastqc_dir = os.path.join(PROJECT_DIR, 'fastqc')
abstar.fastqc(FASTQ_DIR, output=fastqc_dir)

# run abstar
```

(continues on next page)

(continued from previous page)

```
temp_dir = os.path.join(PROJECT_DIR, 'temp')
json_dir = os.path.join(PROJECT_DIR, 'json')
abstar.run(input=FASTQ_DIR,
           temp=temp_dir,
           output=json_dir)

# import into MongoDB
mongoimport.run(ip=MONGO_IP,
                port=MONGO_PORT
                user=MONGO_USER,
                password=MONGO_PASS,
                input=json_dir,
                db='MyDatabase'
                delim1='-',
                delim2='.')
```

2.2.3 Case #3

Now we'd like to use abstar as part of an analysis script in which sequence annotation isn't the primary output. In the previous examples, we started with raw(ish) sequence data and ended with either a directory of JSON files or a MongoDB database populated with abstar output. In this case, we're going to start with a MongoDB database, query that database for some sequences, and generate the unmutated common ancestor (UCA). We'd like to annotate the UCA sequence inline (as part of the script) so that we can do world-changing things with the annotated UCA later in our script. For simplicity's sake, we're querying a local MongoDB database that doesn't have authentication enabled, although `abutils.utils.mongodb` can work with remote MongoDB servers that require authentication.

```
import abstar

from abutils.utils import mongodb
from abutils.utils.sequence import Sequence

DB_NAME = 'MyDatabase'
COLLECTION_NAME = 'MyCollection'

def get_sequences(db_name, collection_name):
    db = mongodb.get_db(db_name)
    c = db[collection]
    seqs = c.find({'chain': 'heavy'})
    return [Sequence(s) for s in seqs]

def calculate_uca(sequences):
    #
    # code to calculate the UCA sequence, as a string
    #
    return uca

# get sequences, calculate the UCA
sequences = get_sequences(DB_NAME, COLLECTION_NAME)
uca_seq = calculate_uca(sequences)

# run abstar on the UCA, returns an abutils Sequence object
uca = abstar.run(['UCA', uca_seq])

# do amazing, world-changing things with the UCA
```

(continues on next page)

(continued from previous page)

```
# ...
# ...
# ...
```

2.3 API Reference

2.3.1 core

core

abstar.core.abstar.**run**(**args*, ***kwargs*)

Runs abstar.

Input sequences can be provided in several different formats:

1) individual sequences as positional arguments: `run(seq1, seq2, temp=temp, output=output)`
2) a list of sequences, as an argument: `run([seq1, seq2], temp=temp, output=output)`
3) a single FASTA/Q-formatted input file, passed via `input`
4) a directory of FASTA/Q-formatted files, passed via `input`

When passing sequences (not FASTA/Q files), the sequences can be in any format recognized by `abtools.sequence.Sequence`, including:

- a raw nucleotide sequence, as a string (a random sequence ID will be assigned)
- a list/tuple of the format `[sequence_id, sequence]`
- a BioPython SeqRecord object
- an abtools Sequence object

Caution: Supplying a single input sequence in list/tuple format is not supported, as abstar assumes that each element of an iterable is a separate sequence if an iterable is the only argument. Either convert the list/tuple to a `Sequence` object before calling `abstar.run()` or supply a nested list containing the sequence information, for example: `[[sequence_id, sequence], ]`.

Either sequences, `project_dir`, or all of `input`, `output` and `temp` are required.

Examples

If processing a single sequence, you can pass the raw sequence, as a string:

```
import abstar

result = abstar.run('ATGC')
```

or as a `Sequence` object:

```
sequence = Sequence('ATGC', id='seq1')

result = abstar.run(sequence)
```

If you pass just the raw sequence, a random sequence ID will be generated with `uuid.uuid4()`. In either case, when given a single sequence, `abstar.run()` will return a single `Sequence` object. If running multiple sequences, you can either pass each sequence as a positional argument:

```
result_list = run(['seq1', 'ATGC'], ['seq2', 'CGTA'])
```

or you can pass a list of sequences as the first argument, in this case using sequences parsed from a FASTA file using Biopython:

```
from Bio import SeqIO

fasta = open('my_sequences.fasta', 'r')
seqs = [s for s in SeqIO.parse(fasta, 'fasta')]
result_list = abstar.run(seqs)
```

When given multiple sequences, `abstar.run()` will return a list of abtools `Sequence` objects, one per input sequence.

If you'd prefer not to parse the FASTQ/A file into a list (for example, if the input file is extremely large), you can pass the input file path directly, along with a temp directory and output directory:

```
result_files = abstar.run(input='/path/to/my_sequences.fasta',
                          temp='/path/to/temp',
                          output='/path/to/output')
```

Given a file path, `abstar.run()` returns a list of output file paths. In the above case, `result_files` will be a list containing a single output file path: `/path/to/output/json/my_sequences.json`.

If you have a directory containing multiple FASTQ/A files, you can pass the directory path using `input`:

```
result_files = abstar.run(input='/path/to/input',
                          temp='/path/to/temp',
                          output='/path/to/output')
```

As before, `result_files` will contain a list of output file paths.

If your input directory contains paired FASTQ files (gzip compressed or uncompressed) that need to be merged prior to processing with abstar:

```
result_files = abstar.run(input='/path/to/input',
                          temp='/path/to/temp',
                          output='/path/to/output',
                          merge=True)
```

The paired read files in `input` will be merged with PANDAseq prior to processing with abstar. By default, PANDAseq's 'simple bayesian' read merging algorithm is used, although alternate algorithms can be selected with `pandaseq_algo`.

abstar provides several output format options. By default, abstar will produce JSON-formatted output file. abstar's output format options include:

json abstar's default format, in Javascript Object Notation (JSON) format. This format is the most comprehensive. JSON's nesting and inclusion of programmatic objects (such as lists) make this format extremely flexible and well-suited to adaptive immune receptor sequence data, particularly for cases in which addtional fields may be added in the future (such as clonality-related annotations).

airr Tab-delimited format compatible with the Adaptive Immune Receptor Repertoires Community's (AIRR-C) schema guidelines. This format contains all required fields, several "optional" fields, and several fields that are not part of the schema but conform to the naming conventions of existing schema fields (examples include `v_mutations` and `v_mutations_aa`).

tabular Comma-delimited format containing a subset of the fields contained in the default JSON output format. This format was originally conceived for extremely large datasets, for which output size and compatibility with tabular databases (such as MySQL and Apache Spark) were high priorities.

imgt Comma-delimited format that mimics the IMGT Summary file. This output option is provided to minimize the effort needed to convert existing IMGT-based pipelines to abstar.

Multiple output formats can be produced in a single run of abstar, although this is only available when passing an input file or directory; passing individual sequences or a list of sequences (which returns `Sequence` objects) can only return a single output format. To produce AIRR output:

```
result_files = abstar.run(input='/path/to/input',
                          temp='/path/to/temp',
                          output='/path/to/output',
                          output_type='airr')
```

To produce both JSON and AIRR-formatted outputs:

```
result_files = abstar.run(input='/path/to/input',
                          temp='/path/to/temp',
                          output='/path/to/output',
                          output_type=['json', 'airr'])
```

In interactive mode (providing `Sequence` objects rather than an input file or directory), returning AIRR-formated data can be accomplished by:

```
results = abstar.run(sequences, output_type='airr')
```

Parameters

- **project_dir** (*str*) – Path to the project directory. Most useful when directly downloading files from BaseSpace, and all subdirectories will be created by AbStar.
- **input** (*str*) – Path to input directory, containing FASTA/Q files. If performing read merging with PANDAseq, paired FASTQ files may be gzip compressed.
- **output** (*str*) – Path to output directory.
- **temp** (*str*) – Path to temp directory, where intermediate job files will be stored.
- **log** (*str*) – Path to log file. If not provided and `project_dir` is provided, the log will be written to `/path/to/project_dir/abstar.log`. If output is provided, log will be written to `/path/to/output/abstar.log`.
- **germ_db** (*str*) – Germline database to be used. Choices are 'human', 'macaque', 'mouse', 'humouse', and 'rabbit'. The 'humouse' database contains all germline genes from human and mouse databaes, and is designed to process data from humanized mouse models expressing one or more human germline genes as well as mouse germline genes. Default is 'human'.
- **isotype** (*bool*) – If True, the isotype will infered by aligning the sequence region downstream of the J-gene. If False, the isotype will not be determined. Default is True.
- **uid** (*int*) – Length (in nucleotides) of the Unique Molecular ID used to barcode input RNA. A positive integer results in the UMID being parsed from the start of the read (or

merged read), a negative integer results in parsing from the end of the read. Default is 0, which results in no UMID parsing.

- **gzip** (*bool*) – If True, compresses output files with gzip. Default is False.
- **pretty** (*bool*) – If True, formats JSON output files to be more human-readable. If False, JSON output files contain one record per line. Default is False.
- **output_type** (*str*) – Options are 'json' 'airr', 'tabular', or 'imgt'. JSON output is the most detailed. Default is 'json'.
- **merge** (*bool*) – If True, input must be paired-read FASTA files (gzip compressed or uncompressed) which will be merged with PANDAseq prior to processing with AbStar. If `basespace` is True, `merge` is automatically set to True. Default is False.
- **pandaseq_algo** (*str*) – Define merging algorithm to be used by PANDAseq. Options are 'simple_bayesian', 'ea_util', 'flash', 'pear', 'rdp_mle', 'stitch', or 'uparse'. Default is 'simple_bayesian', which is the default PANDAseq algorithm.
- **debug** (*bool*) – If `True`, `abstar.run()` runs in single-threaded mode, the log is much more verbose, and temporary files are not removed. Default is `False`.

Returns

If the input is a single sequence, `run()` returns a single abtools `Sequence` object.

If the input is a list of sequences, `run()` returns a list of abtools `Sequence` objects.

If the input is a file or a directory of files, `run()` returns a list of output files.

2.3.2 assigner

base assigner

class abstar.assigners.assigner.**BaseAssigner**(*db_name*)

`BaseAssigner` provides an abstract base class for custom VDJ Assigners.

Several different tools exist for inferring the germline components of recombined antibody sequences. Due to their differing design goals, these tools often have different priorities – some are optimized for speed with a small (often negligible) accuracy penalty, while others define recombination junctions with high precision at the cost of speed. Because there isn't a single germline assignment tool that is optimal for every use case, the germline assignment component of AbStar has been designed to be modular and replaceable, allowing the relatively straightforward addition of new germline assignment tools and the selection of the desired Assigner at runtime. AbStar's default Assigner (`blastn`) identifies V- and J-genes using BLASTn and identifies D-genes using rapid Smith-Waterman local alignment.

The basic purpose of any custom Assigner class is to accept sequences and produce `VDJ` objects. `VDJ` objects package the input sequence together with `GermlineSegment` objects that contain information about V, D and/or J assignments. Following germline assignment by the Assigner, addtional annotation will be performed by AbStar. The `VDJ` objects provide a common interface by which various Assigners can communicate with the additional sequence annotation components in AbStar. This additional annotation is consistent regardless of the Assigner (as is the output schema), and provides a unified method by which different germline Assigners can be used across different projects while maintaining schematic compatibility with downstream analysis tools.

`BaseAssigner` provides the following attributes:

- **assigned: a list designed to contain VDJ objects for which** successful V(D)J assignment has been completed. These `VDJ` objects will be additionally annotated by AbStar following assignment. Exception/log information in these `VDJ` objects will only be written to the log file if AbStar is run in debug mode.

- **unassigned: a list designed to contain VDJ objects for which** V(D)J assigjment was unsuccessful. These VDJ objects will not be additionally annotated by AbStar and any exception/log information contained in these VDJ objects will be logged (even if AbStar was not run in debug mode).
- **germline_directory: path to the directory containing germline databases.** Germline DBs are heirarchically grouped into separate folders first by species, then by the Assigner for which they were designed (blastn, for example). Within each folder, germline DB files are named as {segment}.fasta, so the Blast V-gene DB for human would be human/blast/v.fasta. Note that this will change as AbStar is updated to annotate TCRs in addition to antibodies
- **binary_directory: path to the directory containing Assigner binaries. Binaries** are named as {binary}_{system}, where system is the lowercase output from platform.system(). For example, the Blastn binary for OSX would be at binary_directory/blastn_darwin. Some assigners require more than a simple binary for execution (Partis, for example). In this case, all required elements will be placed into a folder within the binary directory, named using the same convention as binaries. So the folder containing Linux-compiled Partis components will be binary_directory/partis_linux. The main Partis executable would then be located at binary_directory/partis_linux/bin/partis.

In order to build a custom assignment class, you simply need to subclass BaseAssigner, implement your assigner, and register your assigner in abstar.assigners.registry.ASSIGNERS. Obviously, this is in addition to adding any required binaries and specialized germline database formats. AbStar prefers to incorporate assigner binaries if at all possible (as opposed to relying on system installations) to smooth the process of installing AbStar and eliminate the need for users to install several additional programs before having a fully operational AbStar installation.

There are two critical elements to the design of your assigner class. First, ensure that you call the BaseAssigner's __init__() at the start of your class's __init__(). Second, you must implement the __call__() method.

Your custom assigner class's __init__() must accept a single argument (species, in addition to the required self). You must then call the BaseAssigner's __init__() and pass the species argument. The simplest was to do this is by using super(), like this: super(MyAssigner, self).__init__(species), where MyAssigner is your assigner class.

- __call__() must accept two arguments. The first (sequence_file) is a FASTA- or FASTQ-formatted file of input sequences, depending on the user-provided input. The second argument (file_format) defines the format of sequence_file, either 'fasta' or 'fastq'. If a FASTQ file is required by your assigner, ensure that __call__() raises the appropriate exception (and provides a sufficiently clear description of the problem) if a FASTA-formatted file is provided.
- The result of __call__() should be the generation of a VDJ object for each input sequence (more on this below). The VDJ objects should be appended to either the assigned or unassigned attribute, depending on the outcome of the V(D)J assignment. A high-level overview of how __call__() should work is as follows:

```
def __call__(self, seqs, species):
    for seq in seqs:
        vdj = self.assign_vdj(seq, species)
        if self.is_properly_assigned(vdj):
            self.assigned.append(vdj)
        else:
            self.unassigned.append(vdj)
```

  In this case, the output of assign_vdj() should be a VDJ object.

AbStar defines VDJ objects, which are designed to package the input sequence and GermlineSegment objects that provide information about germline assignment for V, D and/or J genes. VDJ objects must contain the input sequence and may also contain information about V, D and J assignments. While the sequence

is required, the germline assignments are optional, to allow creation of VDJ objects for light chains (which lack a D-gene) and/or for unsuccessful assignments. Additionally, this allows flexibility in designing the __call__() method. Assigners can either instantiate a VDJ object at the start of __call__() and attach GermlineSegment``s as they are computed or they can instantiate ``VDJ objects upon completion of V(D)J assignment using the input sequences and computed GermlineSegment``s as instantiation arguments. Because ``VDJ objects contain built-in logging capability, such objects should be created for each input sequence even in the case of unsuccessful assignments so that assignment issues can be logged and reported.

VDJ objects include a logging method: log(). Using log() is similar to using Python's builtin print() function. Multiple arguments to log() will be space-delimited, and a sep keyword allows you to provide an alternate delimiter. Log messages are typically used to record useful information or to note the occurance of errors/exceptions that are caught and handled by the Assigner. It is also a useful means by which to note the reason a specific input sequence is unable to be fully processed (because it isn't a valid antibody sequence, etc). By convention, AbStar log entries should contain the name of the log event (in all caps) followed by a colon and additional log information. An example log() call in the case of a V(D)J assignment that was aborted because the V-gene alignment score didn't meet the alignment quality threshold:

```
for seq in seqs:
    vdj = VDJ(seq)
    v = assign_v_gene(seq)
    if v.score < threshold:
        vdj.log('V-GENE ASSIGNMENT ERROR:', 'Alignment score ({}) was too low'.
↪format(v.score))
        continue
```

VDJ objects also contain a mechanism for logging exceptions: exception(). This is typically used to log unexpected exceptions (ie, those that are not caught and handled by the Assigner). It is useful to provide some information about when/where in the Assigner the exception occured, as well as a formatted version of the exception traceback. An example of exceptions() being used to log an exception that occurs during V-gene assignment:

```
vdj = VDJ(seq)
try:
    v = assign_v_gene(seq)
except:
    vdj.exception('V-GENE ASSIGNMENT EXCEPTION:', traceback.format_exc())
```

One of the primary benefits of recording log and exception information through the VDJ objects is that it allows for much easier formatting of such events when multiple sequences are being processed in parallel. Using Python's logging utility would result in log events being logged as they occur, which means that multiple log entries for the same input sequence may be far apart in the resulting log file. Packaging log and exception information with each VDJ object alows AbStar to create a much more useful log file in which all information related to a single input sequence is found in a single log location.

GermlineSegment objects contain information about the assigned germline gene segment (V, D or J). Instantiation of a GermlineSegment object requires only the name of the germline gene segment (in IMGT format, like 'IGHV3-23*01') and the database name. Several other optional arguments can be included at instantiation. score is the assignment score, typically an int or float. strand indicates the strand orientation of the input sequence ('+' or '-'). others is a list of additional high scoring GermlineSegment objects. assigner_name is the name of the custom Assigner (as a str), and will be converted to lowercase before recording in AbStar output.

GermlineSegment objects also provide a few additional public attributes that may be desirable for certain custom Assigners, especially those that have been specifically designed to accurately define recombination regions. query_start, query_end, germline_start and germline_end allow the Assigner to define the start and end points of the germline alignment for both the query and germline sequences. Note that

both `query_start` and `query_start` and `query_end` MUST be relative to the complete query sequence in the `'+'` orientation (in other words, V-gene at the 5' end and J-gene at the 3' end). AbStar realigns all assigned germline genes using alignment parameters designed to accurately identify somatic mutations as well as somatic mutation-induced indels. If the additional assignment position attributes are provided by the Assigner, AbStar will force their use during realignment. If not, AbStar will use the assigned germline genes for realignment and will define the start and end points of the germline alignment during realignment.

Example

The following is an example of a custom Assigner class, named MyAssigner. It uses BioPython's SeqIO to parse the input file, and because BioPython is a dependency of AbStar, you can assume that BioPython will be present on any system running AbStar. All custom Assigners should be located in the `assigners` directory:

```
import os
import platform
import traceback

from abstar.assigners.assigner import BaseAssigner
from abstar.core.vdj import VDJ
from abstar.core.germline import GermlineSegment

from Bio import SeqIO

class MyAssigner(BaseAssigner):

    def __init__(self, db_name):
        super(MyAssigner, self).__init__(db_name)
        self.binary = os.path.join(self.binary_directory, 'mybinary_{}'.
↪format(platform.system()))

    def __call__(self, sequence_file, file_format):
        for sequence in SeqIO.parse(open(sequence_file, 'r'), file_format):
            seq = Sequence(sequence)
            vdj = VDJ(seq)

            # V-gene assignment
            # The VDJ object (rather than just the sequence) is passed to the␣
↪assignment
            # function for three reasons:
            #    1) it contains the sequence, so we don't need to pass it␣
↪separately
            #    2) it allows logging during germline assignment, via vdj.log()
            #    3) subsequent assignments will have access to information about
            #       previous assignments (J-gene assignment operations have access
            #       to information about the V-gene assignment, etc.)
            try:
                vdj.v = self.assign_germline(vdj, 'V')
            except:
                vdj.exception('V-ASSIGNMENT:', traceback.format_exc())

            # J-gene assignment
            try:
                vdj.j = self.assign_germline(vdj, 'J')
            except:
                vdj.exception('J-ASSIGNMENT:', traceback.format_exc())
```

(continues on next page)

(continued from previous page)

```
            # D-gene assignment
            try:
                vdj.d = self.assign_germline(vdj, 'D')
            except:
                vdj.exception('D-ASSIGNMENT:', traceback.format_exc())

            return vdj

    def assign_germline(self, vdj, segment):
        germ_db = os.path.join(self.germline_directory, '{}/ungapped/{}.fasta'.
↪format(self.db_name,
                                                                                 ␣
↪    segment.lower()))

        # do stuff to assign the germline gene, using the species-appropriate␣
↪germline DB
        # ...
        # ...
        # ...
        # germs is a list of germline genes, ordered by score (highest first)

        if germs[0].score < threshold:
            vdj.log('{}-ASSIGNMENT ERROR:'.format(segment),
                    'Score ({}) is too low'.format(germs[0].score))
            return None
        others = [GermlineSegment(germ.name, self.db_name, score=germ.score) for␣
↪germ in germs[1:6]]
        return GermlineSegment(germs[0].name, self.db_name, score=germs[0].score,␣
↪others=others)
```

BLASTn assigner

class abstar.assigners.blastn.**Blastn**(*db_name*)

docstring for Blastn

blast(*seq_file*, *segment*)

Runs BLASTn against an antibody germline database.

seq_file (str): Path to a FASTA-formatted file of input sequences.

segment (str): Germline segment to query. Options are `V` and `J`.

2.3.3 preprocess

preprocess

abstar.preprocess.**quality_trim**(*input_directory=None*, *output_directory=None*, *quality_cutoff=20*, *length_cutoff=50*, *quality_type=u'sanger'*, *compress_output=True*, *file_pairs=None*, *singles_directory=None*, *nextseq=False*, *paired_reads=True*, *allow_5prime_trimming=False*, *print_debug=False*)

Performs quality trimming with sickle.

Parameters

- **input_directory** (*str*) – Path to a directory of files to be quality trimmed. If the directory contains paired reads, they should follow the Illumina MiSeq naming scheme. If you have paired reads that do not follow the MiSeq naming scheme, you can group the paired read files yourself and pass them to `--file-pairs`.
- **output_directory** (*str*) – Path to the output directory, into which quality- trimmed read files will be deposited. If not provided, a directory will be created in the parent directory of `input_directory`. Required if using `file_pairs` instead of `input_directory`.
- **quality_cutoff** (*int*) – Quality score at which to truncate reads. Default is `20`.
- **length_cutoff** (*int*) – Reads will be discarded if, after quality trimming, the length is shorter than this cutoff. Default is `50`.
- **quality_type** (*str*) – Quality score type. Options are `solexa`, `illumina`, and `sanger`. `illumina` is equivalent to Casava 1.3-1.7 and `sanger` is Casava >= 1.8. Default is `sanger`.
- **compress_output** (*bool*) – If `True`, output files will be gzip compressed. Default is `True`.
- **file_pairs** (*list*) – If input files are paired-end reads that don't follow Illumina's MiSeq naming scheme, you can pass a list of lists/tuples, with each list/tuple containing a pair of read file paths.
- **singles_directory** (*str*) – Path to singles output directory. If processing paired reads and one read of the pair passes quality/length filters and the other doesn't, the single passing read will be written to this file. Default is `None`, which results in the single sequences being discarded and not written to file.
- **nextseq** (*bool*) – Set to `True` if the sequencing data comes from a NextSeq run. The file naming scheme for NextSeq runs is different that MiSeq runs, and setting this option will allow NextSeq paired read files to be processed appropriately. Default is `False`.
- **paired_reads** (*bool*) – If `True`, reads will be processed as paired reads. If `False`, each read will be processed separately. It is not advisable to process paired reads with `paired_reads` set to `False` because if paired read files are processed separately and one read passes filters while the paired read doesn't, this may cause problems with downstream processes (like read merging).
- **allow_5prime_trimming** (*bool*) – If `True`, quality trimming will be performed on the 5' end of the reads as well as the 3' end. Default is `False`.

Returns Path to the output directory

Return type str

abstar.preprocess.**adapter_trim**(*input_directory*, *output_directory=None*, *adapter_5prime=None*, *adapter_3prime=None*, *adapter_5prime_anchored=None*, *adapter_3prime_anchored=None*, *adapter_both=None*, *compress_output=True*)

Trims adapters with cutadapt.

Parameters

- **input_directory** (*str*) – Path to a directory of FASTQ files to be adapter trimmed. Required.
- **output_directory** (*str*) – Path to the output directory. If not provided, a directory will be created in the parent directory of `input_directory`.

- **adapter_5prime** (*str*) – Path to a FASTA-formatted file of adapters to be trimmed from the 5' end of reads.
- **adapter_3prime** (*str*) – Path to a FASTA-formatted file of adapters to be trimmed from the 3' end of reads.
- **adapter_5prime_anchored** (*str*) – Path to a FASTA-formatted file of adapters to be trimmed from the 5' end of reads. More strictly requires the read to be anchored to the 5' end of the read than when using `adapter_5prime`.
- **adapter_3prime_anchored** (*str*) – Path to a FASTA-formatted file of adapters to be trimmed from the 3' end of reads. More strictly requires the read to be anchored to the 3' end of the read than when using `adapter_3prime`.
- **adapter_both** (*str*) – Path to a FASTA-formatted file of adapters that will be trimmed from either end of the reads.
- **compress_output** (*bool*) – If `True`, output files will be gzip compressed. Default is `True`.

Returns Path to the output directory

Return type str

abstar.preprocess.**fastqc**(*input_directory*, *output_directory=None*, *threads=-1*)

Performs FASTQC analysis on raw NGS data.

Parameters

- **input_directory** (*str*) – Path to the input directory, containing one or more FASTQ files (either gzip compressed or uncompressed).
- **output_directory** (*str*) – Path to the output directory, where the FASTQC results will be deposited. If not provided, a directory named 'fastqc_reports' will be created in the parent directory of `input_directory`
- **threads** (*int*) – Number of threads to be used (passed to the `-t` flag when running `fastqc`). Default is -1, which uses all cores.

Returns path to the output directory

Return type str

2.3.4 helper utilities

helper utilities

abstar.utils.basespace

abstar.utils.basespace.**download**(*download_directory*, *project_id=None*, *project_name=None*)

Downloads sequencing data from BaseSpace (Illumina's cloud storage platform).

Before accessing BaseSpace through the AbStar API, you need to set up a credentials file:

1. You need a BaseSpace access token. The easiest way to do this is to set up a BaseSpace developer account following these instructions
2. Make a BaseSpace credentials file using your developer credentials:

```
$ make_basespace_credfile
```

and follow the instructions.

Examples

If you know the name of the project you'd like to download:

```
from abstar.utils import basespace

basespace.download('/path/to/download_directory', project_name='MyProject')
```

If you know the ID of the project you'd like to download:

```
basespace.download('/path/to/download_directory', project_id='ABC123')
```

If neither `project_id` nor `project_name` is provided, a list of your available BaseSpace projects will be provided and you can select a project from that list:

```
basespace.download('/path/to/download_directory')
```

Parameters

- **download_directory** (*str*) – Directory into which the raw sequences files should be downloaded. If the directory does not exist, it will be created.
- **project_id** (*str*) – ID of the project to be downloaded.
- **project_name** (*str*) – Name of the project to be downloaded.

Returns The number of sequence files downloaded.

Return type int

abstar.utils.mongoimport

abstar.utils.mongoimport.**run**(***kwargs*)

Imports one or more JSON files into a MongoDB database.

Examples

To import a single JSON file into MyDatabase on a local MongoDB database:

```
from abstar.utils import mongoimport

mongoimport.run(input='/path/to/MySequences.json', db='MyDatabase')
```

This will result in a collection named 'MySequences.json' being created in MyDatabase on your local MongoDB instance (if it doesn't already exist) and the data from MySequences.json being imported into that collection.

Doing the same thing, but with a remote MongoDB server running on port 27017:

```
mongoimport.run(ip='123.45.67.89',
                user='my_username',
                password='Secr3t',
                input='/path/to/MySequences.json',
                db='MyDatabase')
```

But what if we want the collection name to be different than the file name? We can truncate the filename at the first occurance of any given pattern with `delim1`:

```
mongoimport.run(input='/path/to/MySequences.json,
                db='MyDatabase',
                delim1='.')
```

In this case, the collection name is created by truncating the input file name at the first occurance of `.`, so the collection name would be MySequences. We can also truncate the filename at the Nth occurance of any given pattern by using `delim1` with `split1_pos`:

```
mongoimport.run(input='/path/to/my_sequences_2016-01-01.json,
                db='MyDatabase',
                delim1='_',
                split1_pos=2)
```

which results in a collection name of my_sequences.

If we have more complex filenames, we can use `delim1` in combination with `delim2`. When `delim1` and `delim2` are used together, `delim1` becomes the pattern used to cut the filename on the left and `delim2` is used to cut the filename on the right. For example, if our filename is `plate-2_SampleName-01_redo.json` and we want the collection to be named `SampleName`, we would set `delim1` to _ and `delim2` to -. We also need to specify that we want to cut at the second occurance of `delim2`, which we can do with `split2_pos`:

```
mongoimport.run(input='/path/to/plate-2_SampleName-01_redo.json,
                db='MyDatabase',
                delim1='_',
                delim2='-',
                split2_pos=2)
```

Trimming filenames this way is nice, but it becomes much more useful if you're importing more than one file at a time. `mongoimport.run()` will accept a list of file names, and will generate separate collection names for each input file:

```
files = ['/path/to/A01-Sample01_2016-01-01',
         '/path/to/A02-Sample02_2016-01-01',
         '/path/to/A03-Sample03_2016-01-01]

mongoimport.run(input=files,
                db='MyDatabase',
                delim1='-',
                delim2='_')
```

The three input files will be imported into collections `Sample01`, `Sample02` and `Sample03`, respectively. Finally, you can pass the path to a directory containing oen or more JSON files, and all the JSON files will be imported:

```
mongoimport.run(input='/path/to/output/directory',
                db='MyDatabase',
                delim1='-',
                delim2='_')
```

Parameters

- **input** (*str, list*) – Input is required and may be one of three things:

  1) A list/tuple of JSON file paths

2) A path to a single JSON file

3) A path to a directory containing one or more JSON files.

- **ip** (*str*) – The IP address of the MongoDB server. Default is 'localhost'.
- **port** (*int*) – MongoDB port. Default is 27017.
- **user** (*str*) – Username with which to connect to the MongoDB database. If either of `user` or `password` is not provided, `mongoimport.run()` will attempt to connect to the MongoDB database without authentication.
- **password** (*str*) – Password with which to connect to the MongoDB database. If either of `user` or `password` is not provided, `mongoimport.run()` will attempt to connect to the MongoDB database without authentication.
- **db** (*str*) – Name of the MongoDB database for import. Required.
- **log** (*str*) – Path to a logfile. If not provided log information will be written to stdout.
- **delim1** (*str*) – Pattern on which to split the input file to generate the collection name. Default is `None`, which results in the file name being used as the collection name.
- **split1_pos** (*int*) – Occurance of `delim1` on which to split the input file name. Default is 1.
- **delim2** (*str*) – Second pattern on which to split the input file name to generate the collection name. Default is None, which results in only `delim1` being used.
- **split2_pos** (*int*) – Occurance of `delim2` on which to split the input file name. Default is 1.

abstar.utils.pandaseq

`abstar.utils.pandaseq.`**run**(*input*, *output*, *algorithm=u'simple_bayesian'*, *nextseq=False*)

Merge paired-end FASTQ files with PANDAseq.

Examples

To merge a directory of raw (gzip compressed) files from a MiSeq run:

```
merged_files = run('/path/to/input', '/path/to/output')
```

Same as above, but using the Pear read merging algorithm:

```
merged_files = run('/path/to/input', '/path/to/output', algorithm='pear')
```

To merge a list of file pairs:

```
file_pairs = [(sample1_R1.fastq, sample1_R2.fastq),
              (sample2_R1.fastq.gz, sample2_R2.fastq.gz),
              (sample3_R1.fastq, sample3_R2.fastq)]
merged_files = run(file_pairs, '/path/to/output')
```

Parameters

- **input** (*str, list*) – Input can be one of three things:

1. path to a directory of paired FASTQ files

2. a list of paired FASTQ files
3. a list of read pairs, with each read pair being a list/tuple containing paths to two paired read files

Regardless of what input type is provided, paired FASTQ files can be either gzip compressed or uncompressed.

When providing a list of files or a directory of files, it is assumed that all files follow Illumina naming conventions. If your file names aren't Illumina-like, submit your files as a list of read pairs to ensure that the proper pairs of files are merged.

- **output** (*str*) – Path to an output directory, into which merged FASTQ files will be deposited. To determine the filename for the merged file, the R1 file (or the first file in the read pair) is split at the first occurance of the '_' character. Therefore, the read pair `['my-sequences_R1.fastq', 'my-sequences_R2.fastq']` would be merged into `my-sequences.fasta`.
- **algorithm** (*str*) – PANDAseq algorithm to be used for merging reads. Choices are: 'simple_bayesian', 'ea_util', 'flash', 'pear', 'rdp_mle', 'stitch', or 'uparse'. Default is 'simple_bayesian', which is the default PANDAseq algorithm.
- **nextseq** (*bool*) – Set to `True` if the sequencing data was generated on a NextSeq. Needed because the naming conventions for NextSeq output files differs from MiSeq output.

Returns a list of merged file paths

Return type list

CHAPTER 3

about

3.1 License

3.2 News

CHAPTER 4

related projects

CHAPTER 5

Index

- modindex
- search

Python Module Index

a

Index