VAPr Documentation

Release 3.0

Amanda Birmingham, Adam Mark, Carlo Mazzaferro, Guorong Xu,

Aug 20, 2018

Contents:

CHAPTER 1

Introduction

This package is aimed at providing a way of retrieving variant information using ANNOVAR and myvariant.info. In particular, it is suited for bioinformaticians interested in aggregating variant information into a single NoSQL database (MongoDB solely at the moment).

1.1 Installation

1.1.1 Ancillary Libraries

VAPr relies on a variety of packages to function correctly. Below are packages and dependencies required to ensure that VAPr works correctly.

Note: Jupyter, Pandas, and other ancillary libraries are not installed with VAPr and must be installed separately. These can be conveniently install using Anaconda:

```
$ conda install python=3 pandas mongodb pymongo jupyter notebook
```

1.1.2 MongoDB

VAPr is written in Python and stores variant annotations in NoSQL database, using a locally-installed instance of MongoDB. Installation instructions

1.1.3 BCFtools

BCFtools will be used for VCF file merging between samples. To download and install:

```
$ wget https://github.com/samtools/bcftools/releases/download/1.6/bcftools-1.6.tar.bz2
$ tar -vxjf bcftools-1.6.tar.bz2
$ cd bcftools-1.6
$ make
$ make install
$ export PATH=/where/to/install/bin:$PATH
```

Refer here for installation debugging.

1.1.4 Tabix

Tabix and **bgzip** binaries are available through the HTSlib project:

```
$ wget https://github.com/samtools/htslib/releases/download/1.6/htslib-1.6.tar.bz2
$ tar -vxjf htslib-1.6.tar.bz2
$ cd htslib-1.6
$ make
$ make install
$ export PATH=/where/to/install/bin:$PATH
```

Refer here for installation debugging.

1.1.5 ANNOVAR

(It is possible to proceed without installing ANNOVAR. Variants will only be annotated with MyVariant.info. In that case, users can skip the next steps and go straight to the section Known Variant Annotation and Storage)

Users who wish to annotate novel variants will also need to have a local installation of the popular command-line software ANNOVAR(1), which VAPr wraps with a Python interface. If you use ANNOVAR's functionality through VAPr, please remember to cite the ANNOVAR publication (see #1 in Citations)!

The base ANNOVAR program must be installed by each user individually, since its license agreement does not permit redistribution. Please visit the ANNOVAR download form here, ensure that you meet the requirements for a free license, and fill out the required form. You will then receive an email providing a link to the latest ANNOVAR release file. Download this file (which will usually have a name like annovar.latest.tar.gz) and place it in the location on your machine in which you would like the ANNOVAR program and its data to be installed–the entire disk size of the databases will be around 25 GB, so make sure you have such space available!

1.1.6 VAPr

VAPr is compatible with Python 2.7 or later, but it is preferred to use Python 3.5 or later to take full advantage of all functionality. The simplest way to install VAPr is from PyPI with pip, Python's preferred package installer.

```
$ pip install VAPr
```

1.2 Annotation Quickstart using ANNOVAR

An annotation project can be started by providing the API with a small set of information and then running the core methods provided to spawn annotation jobs. This is done in the following manner:

```
# Import core module
from VAPr import vapr_core
import os

# Start by specifying the project information
IN_PATH = "/path/to/vcf"
OUT_PATH = "/path/to/out"
ANNOVAR_PATH = "/path/to/annovar"
MONGODB = 'VariantDatabase'
COLLECTION = 'Cancer'

annotator = vapr_core.VaprAnnotator(input_dir=IN_PATH,
                                    output_dir=OUT_PATH,
                                    mongo_db_name=MONGODB,
                                    mongo_collection_name=COLLECTION,
                                    build_ver='hg19',
                                    vcfs_gzipped=False,
                                    annovar_install_path=ANNOVAR_PATH)

annotator.download_databases()
dataset = annotator.annotate(num_processes=8)
```

1.2.1 Downloading the ANNOVAR databases

If you plan to use Annovar, the command below will download the necessary Annovar databases. The code above includes this step. When Annovar is first installed, it does not install databases by default. The vapr_core has a method download_annovar_databases() that will download the necessary annovar databases. If you do not plan on using Annovar, you should not run this command. Note: this command only needs to be run once, the first time you use VAPr.

```
annotator.download_databases()
```

This will download the required databases from ANNOVAR for annotation and will kickstart the annotation process, storing the variants in MongoDB.

CHAPTER 2

Downstream Analysis

For notes on how to implement these features, refer to the *Tutorial* and the API Reference

2.1 Filtering Variants

Four different pre-made filters that allow for the retrieval of specific variants have been implemented. These allow the user to query in an easy and efficient manner variants of interest

2.1.1 1. Rare Deleterious Variants

- criteria 1: 1000 Genomes (ALL) allele frequency (Annovar) < 0.05 or info not available
- criteria 2: ESP6500 allele frequency (MyVariant.info - CADD) < 0.05 or info not available
- criteria 3: cosmic70 (MyVariant.info) information is present
- criteria 4: Func_knownGene (Annovar) is exonic, splicing, or both
- criteria 5: ExonicFunc_knownGene (Annovar) is not "synonymous SNV"

2.1.2 2. Known Disease Variants

- criteria: cosmic70 (MyVariant.info) information is present or ClinVar data is present and clinical significance is not Benign or Likely Benign

2.1.3 3. Deleterious Compound Heterozygous Variants

- criteria 1: genotype_subclass_by_class (VAPr) is compound heterozygous
- criteria 2: CADD phred score (MyVariant.info - CADD) > 10

2.1.4 4. De novo Variants

- criteria 1: Variant present in proband
- criteria 2: Variant not present in either ancestor-1 or ancestor-2

2.1.5 Create your own filter

As long as you have a MongoDB instance running and an annotation job ran successfully, filtering can be performed through pymongo as shown by the code below. Running the query will return a `cursor` object, which can be iterated upon.

If instead a list is intended to be created from it, simply add: *filter2 = list(filter2)*.

Warning: If the number of variants in the database is large and the filtering is not set up correctly, returning a list will be probably crash your computer since lists are kept in memory. Iterating over the cursor object perform *lazy evaluations* (i.e., one item is returned at a time instead of in bulk) which are much more memory efficient.

Further, if you'd like to customize your filters, a good idea would be to look at the available fields to be filtered. Looking at the myvariant.info documentation, you can see what are all the fields available and can be used for filtering.

```
from pymongo import MongoClient

client = MongoClient()
db = getattr(client, mongodb_name)
collection = getattr(db, mongo_collection_name)

filtered = collection.find({"$and": [
                                    {"$or": [{"func_knowngene": "exonic"},
                                             {"func_knowngene": "splicing"}]},
                                    {"cosmic70": {"$exists": True}},
                                    {"1000g2015aug_all": {"$lt": 0.05}}
                            ]})

# filtered = list(filtered) Uncomment this if you'd like to return them as a list
for var in filtered:
    print(var)
```

2.2 Output Files

Although iterating over variants can be interesting for cursory analyses, we provide functionality to retrieve as well csv files for downstream analysis. A few options are available:

2.2.1 Unfiltered Variants CSV

`write_unfiltered_annotated_csv(out_file_path)`

- All variants will be written to a CSV file.

2.2.2 Filtered Variants CSV

```
write_filtered_annotated_csv(variant_list, out_file_path)
```

- A list of filtered variants will be written to a CSV file.

2.2.3 Unfiltered Variants VCF

```
write_unfiltered_annotated_vcf(vcf_out_path)
```

- All variants will be written to a VCF file.

2.2.4 Filtered Variants VCF

```
write_filtered_annotated_vcf(variant_list, vcf_out_path)
```

- A List of filtered variants will be written to a VCF file.

CHAPTER 3

Core Methods

See the API Reference for *VAPr.vapr_core module* for a detailed functionality of the core methods and classes of this package.

CHAPTER 4

Tutorial

A brief, although comprehensive tour of the functionality offered by this package can be found in this Jupyter Notebook. To run it interactively, download the github repo (or just the Notebook), install the required dependencies (see *Installation*)

CHAPTER 5

VAPr package

5.1 Submodules

5.2 VAPr.annovar_output_parsing module

class VAPr.annovar_output_parsing.**AnnovarAnnotatedVariant**
Bases: object

ALLELE_DEPTH_KEY = 'AD'

FILTER_PASSING_READS_COUNT_KEY = 'filter_passing_reads_count'

GENOTYPE_KEY = 'genotype'

GENOTYPE_LIKELIHOODS_KEY = 'genotype_likelihoods'

GENOTYPE_SUBCLASS_BY_CLASS_KEY = 'genotype_subclass_by_class'

HGVS_ID_KEY = 'hgvs_id'

SAMPLES_KEY = 'samples'

SAMPLE_ID_KEY = 'sample_id'

classmethod make_per_variant_annotation_dict(*fields_by_annovar_header*, *hgvs_id*, *format_string*, *genotype_field_strings_by_sample_name*)

class VAPr.annovar_output_parsing.**AnnovarTxtParser**
Bases: object

Class that processes an Annovar-created tab-delimited text file.

ALT_HEADER = 'alt'

CHR_HEADER = 'chr'

CYTOBAND_HEADER = 'cytoband'

END_HEADER = 'end'

ESP6500_ALL_HEADER = 'esp6500siv2_all'

EXONICFUNC_KNOWNGENE_HEADER = 'exonicfunc_knowngene'

FUNC_KNOWNGENE_HEADER = 'func_knowngene'

GENEDETAIL_KNOWNGENE_HEADER = 'genedetail_knowngene'

GENE_KNOWNGENE_HEADER = 'gene_knowngene'

GENOMIC_SUPERDUPS_HEADER = 'genomicsuperdups'

NCI60_HEADER = 'nci60'

OTHERINFO_HEADER = 'otherinfo'

RAW_CHR_MT_SUFFIX_VAL = 'M'

RAW_CHR_MT_VAL = 'chrM'

REF_HEADER = 'ref'

SCORE_KEY = 'Score'

STANDARDIZED_CHR_MT_SUFFIX_VAL = 'MT'

STANDARDIZED_CHR_MT_VAL = 'chrMT'

START_HEADER = 'start'

TFBS_CONS_SITES_HEADER = 'tfbsconssites'

THOU_G_2015_ALL_HEADER = '1000g2015aug_all'

classmethod read_chunk_of_annotations_to_dicts_list(*annovar_txt_file_like_obj*, *sample_names_list*, *chunk_index*, *chunk_size*)

5.3 VAPr.annovar_running module

class VAPr.annovar_running.**AnnovarWrapper**(*annovar_install_path*, *genome_build_version*, *custom_annovar_dbs_to_use=None*)

Bases: object

Wrapper around ANNOVAR download and annotation functions

download_databases()

hg_19_databases = {'1000g2015aug': 'f', 'knownGene': 'g'}

hg_38_databases = {'1000g2015aug': 'f', 'knownGene': 'g'}

run_annotation(*single_vcf_path*, *output_basename*, *output_dir*)

5.4 VAPr.chunk_processing module

class VAPr.chunk_processing.**AnnotationJobParamsIndices**

CHUNK_INDEX_INDEX = 0

CHUNK_SIZE_INDEX = 2

COLLECTION_NAME_INDEX = 4

DB_NAME_INDEX = 3

FILE_PATH_INDEX = 1

GENOME_BUILD_VERSION_INDEX = 5

SAMPLE_LIST_INDEX = 7

VERBOSE_LEVEL_INDEX = 6

classmethod get_num_possible_indices()

VAPr.chunk_processing.**collect_chunk_annotations_and_store**(*job_params_tuple*)

5.5 VAPr.filtering module

VAPr.filtering.**get_any_of_sample_ids_filter**(*sample_names_list*)

VAPr.filtering.**get_sample_id_filter**(*sample_name*)

VAPr.filtering.**make_de_novo_variants_filter**(*proband*, *ancestor1*, *ancestor2*)
Function for de novo variant analysis. Can be performed on multisample files or or on data coming from a collection of files. In the former case, every sample contains the same variants, although they have differences in their allele frequency and read values. A de novo variant is defined as a variant that occurs only in the specified sample (sample1) and not on the other two (sample2, sample3). Occurrence is defined as having allele frequencies greater than [0, 0] ([REF, ALT]).

VAPr.filtering.**make_deleterious_compound_heterozygous_variants_filter**(*sample_ids_list=None*)

VAPr.filtering.**make_known_disease_variants_filter**(*sample_ids_list=None*)
Function for retrieving known disease variants by presence in Clinvar and Cosmic.

VAPr.filtering.**make_rare_deleterious_variants_filter**(*sample_ids_list=None*)
Function for retrieving rare, deleterious variants

5.6 VAPr.validation module

This module exposes utility functions to validate user inputs

By convention, validation functions in this module raise an appropriate Error if validation is unsuccessful. If it is successful, they return either nothing or the appropriately converted input value.

VAPr.validation.**convert_to_nonneg_int**(*input_val*, *nullable=False*)
For non-null input_val, cast to a non-negative integer and return result; for null input_val, return None.

Parameters

- **input_val** (*Any*) – The value to attempt to convert to either a non-negative integer or a None (if nullable). The recognized null values are '.', None, '', and 'NULL'
- **nullable** (*Optional[bool]*) – True if the input value may be null, false otherwise. Defaults to False.

Returns None if nullable=True and the input is a null value. The appropriately cast non-negative integer if input is not null and the cast is successful.

Raises ValueError – if the input cannot be successfully converted to a non-negative integer or, if allowed, None

VAPr.validation.**convert_to_nullable**(*input_val*, *cast_function*)

For non-null input_val, apply cast_function and return result if successful; for null input_val, return None.

Parameters

- **input_val** (*Any*) – The value to attempt to convert to either a None or the type specified by cast_function. The recognized null values are '.', None, '', and 'NULL'
- **cast_function** (*Callable[[Any], Any]*) – A function to cast the input_val to some specified type; should raise an error if this cast fails.

Returns None if input is the null value. An appropriately cast value if input is not null and the cast is successful.

Raises Error – whatever error is provided by cast_function if the cast fails.

5.7 VAPr.vapr_core module

class VAPr.vapr_core.**VaprAnnotator**(*input_dir*, *output_dir*, *mongo_db_name*, *mongo_collection_name*, *annovar_install_path=None*, *design_file=None*, *build_ver=None*, *vcfs_gzipped=False*)

Bases: object

Class in charge of gathering requirements, finding files, downloading databases required to run the annotation

Parameters

- **input_dir** (*str*) – Input directory to vcf files
- **output_dir** (*str*) – Output directory to annotated vcf files
- **mongo_db_name** (*str*) – Name of the database to which you'll store the collection of variants
- **mongo_collection_name** (*str*) – Name of the collection to which you'd store the annotated variants
- **annovar_install_path** (*str*) – Path to locally installed annovar scripts
- **design_file** (*str*) – path to csv design file
- **build_ver** (*str*) – genome build version to which annotation will be done against. Either *hg19* or *hg38*
- **vcfs_gzipped** (*bool*) – if the vcf files are gzipped, set to True

Returns:

DEFAULT_GENOME_VERSION = 'hg19'

HG19_VERSION = 'hg19'

HG38_VERSION = 'hg38'

SAMPLE_NAMES_KEY = 'Sample_Names'

SUPPORTED_GENOME_BUILD_VERSIONS = ['hg19', 'hg38']

annotate(*num_processes=4*, *chunk_size=2000*, *verbose_level=1*, *allow_adds=False*)

This is the main function of the package. It will run Annovar beforehand, and will kick-start the full annotation functionality. Namely, it will collect all the variant data from Annovar annotations, combine it with data coming from MyVariant.info, and parse it to MongoDB, in the database and collection specified in project_data.

It will return the class *VaprDataset*, which can then be used for downstream filtering and analysis.

Parameters

- **num_processes** (*int, optional*) – number of parallel processes. Defaults to 8
- **chunk_size** (*int, optional*) – int number of variants to be processed at once. Defaults to 2000
- **verbose_level** (*int, optional*) – int higher verbosity will give more feedback, raise to 2 or 3 when debugging. Defaults to 1
- **allow_adds** (*bool, optional*) – bool Allow adding new variants to a pre-existing Mongo collection, or overwrite it (Default value = False)

Returns class:*~VAPr.vapr_core.VaprDataset*

Return type class

annotate_lite(*num_processes=8*, *chunk_size=2000*, *verbose_level=1*, *allow_adds=False*)

'Lite' Annotation: it will query myvariant.info only, without generating annotations from Annovar. It requires solely VAPr to be installed. The execution will grab the HGVS ids from the vcf files and query the variant data from MyVariant.info.

and inability to run native VAPr queries on the data.

It will return the class *VaprDataset*, which can then be used for downstream filtering and analysis.

Parameters

- **num_processes** (*int, optional*) – number of parallel processes. Defaults to 8
- **chunk_size** (*int, optional*) – int number of variants to be processed at once. Defaults to 2000
- **verbose_level** (*int, optional*) – int higher verbosity will give more feedback, raise to 2 or 3 when debugging. Defaults to 1
- **allow_adds** (*bool, optional*) – bool Allow adding new variants to a pre-existing Mongo collection, or overwrite it (Default value = False)

Returns *~VAPr.vapr_core.VaprDataset*

Return type class

download_annovar_databases()

Needed for ANNOVAR to run, it will download the required databases

Args:

Returns:

class VAPr.vapr_core.**VaprDataset**(*mongo_db_name*, *mongo_collection_name*, *merged_vcf_path=None*)

Bases: object

full_name

Full name of database and collection

Args:

Returns Full name of database and collection

Return type str

get_all_variants()
Self-explanatory

Args:

Returns list of variants

Return type list

get_custom_filtered_variants(*filter_dictionary*)
See *Create your own filter* for more information on how to implement

Parameters **filter_dictionary(dictionary** – dict): mongodb custom filter

Returns list of variants

Return type list

get_de_novo_variants(*proband*, *ancestor1*, *ancestor2*)
See *4. De novo Variants* for more information on how this is implemented

Parameters

- **proband** (*str*) – proband variant
- **ancestor1** (*str*) – ancestor #1 variant
- **ancestor2** (*str*) – ancestor #2 variant

Returns list of variants

Return type list

get_deleterious_compound_heterozygous_variants(*sample_names_list=None*)
See *3. Deleterious Compound Heterozygous Variants* for more information on how this is implemented

Parameters **sample_names_list(list** – list, optional): list of samples to draw variants from (Default value = None)

Returns list of variants

Return type list

get_distinct_sample_ids()
Self-explanatory

Args:

Returns list of sample ids

Return type list

get_known_disease_variants(*sample_names_list=None*)
See *2. Known Disease Variants* for more information on how this is implemented

Parameters **sample_names_list(list** – list, optional): list of samples to draw variants from (Default value = None)

Returns list of variants

Return type list

get_rare_deleterious_variants(*sample_names_list=None*)
See *1. Rare Deleterious Variants* for more information on how this is implemented

Parameters **sample_names_list(list** – list, optional): list of samples to draw variants from (Default value = None)

Returns list of variants

Return type list

get_variants_as_dataframe(*filtered_variants=None*)
Utility to get a dataframe from variants, either all of them or a filtered subset

Parameters **filtered_variants** – a list of variants (Default value = None)

Returns pandas.DataFrame

get_variants_for_sample(*sample_name*)
Return variants for a specific sample

Parameters **sample_name** (*str*) – name of sample

Returns list of variants

Return type list

get_variants_for_samples(*specific_sample_names*)
Return variants from multiple samples

Parameters **specific_sample_names** (*list*) – name of samples

Returns list of variants

Return type list

is_empty
If there are no records in the collection, returns True

Args:

Returns if there are no records in the collection, returns True

Return type bool

num_records
Number of records in MongoDB collection

Args:

Returns Number of records in MongoDB collection

Return type int

write_filtered_annotated_csv(*filtered_variants*, *output_fp*)
Filtered csv file containing annotations from a list passed to it, coming from MongoDB

Parameters

- **filtered_variants** (*list*) – variants coming from MongoDB
- **output_fp** (*str*) – Output file path

Returns None

write_filtered_annotated_vcf(*filtered_variants*, *vcf_output_path*, *info_out=True*)

Parameters

- **filtered_variants** (*list*) – variants coming from MongoDB
- **vcf_output_path** (*str*) – Output file path
- **info_out** – if True, extra annotation information will be written to the vcf file (Default value = True)

- **info_out** – bool (Default value = True)

Returns None

write_unfiltered_annotated_csv(*output_fp*)
Full csv file containing annotations from both annovar and myvariant.info

Parameters **output_fp** (*str*) – Output file path

Returns None

write_unfiltered_annotated_csvs_per_sample(*output_dir*)

Parameters **output_dir** – return: None

Returns None

write_unfiltered_annotated_vcf(*vcf_output_path*, *info_out=True*)
Filtered vcf file containing annotations from a list passed to it, coming from MongoDB

Parameters

- **vcf_output_path** (*str*) – Output file path
- **info_out** – if True, extra annotation information will be written to the vcf file (Default value = True)
- **info_out** – bool (Default value = True)

Returns None

5.8 VAPr.vcf_genotype_fields_parsing module

class VAPr.vcf_genotype_fields_parsing.**Allele**(*unfiltered_read_counts=None*)
Bases: object

Store unfiltered read counts, if any, for a particular allele.

unfiltered_read_counts
int or None – Number of unfiltered reads counts for this sample at this site, from AD field.

class VAPr.vcf_genotype_fields_parsing.**GenotypeLikelihood**(*allele1_number*, *allele2_number*, *likelihood_neg_exponent*)
Bases: object

Store parsed info from VCF genotype likelihood field for a single sample.

allele1_number
int – The allele identifier for the left-hand allele inferred for this genotype likelihood.

allele2_number
int – The allele identifier for the right-hand allele inferred for this genotype likelihood.

likelihood_neg_exponent
float – The "normalized" Phred-scaled likelihood of the genotype represented by allele1 and allele2.

class VAPr.vcf_genotype_fields_parsing.**VCFGenotypeInfo**(*raw_string*)
Bases: object

Store parsed info from VCF genotype fields for a single sample.

_raw_string
str – The genotype fields values string from a VCF file (e.g., '0/1:173,141:282:99:255,0,255').

genotype
Optional[*str*] – The type of each of the sample's two alleles, such as 0/0, 0/1, etc.

alleles
List[Allele] – One Allele object for each allele detected for this variant (this can be across samples, so there can be more than 2 alleles).

genotype_likelihoods
List[GenotypeLikelihood] – The GenotypeLikelihood object for each allele.

unprocessed_info
Dict[str, Any] – Dictionary of field tag and value(s) for any fields not stored in dedicated attributes of VCFGenotypeInfo. Values are parsed to lists and/or floats if possible.

genotype_subclass_by_class
Dict[str, str] – Genotype subclass (reference, alt, compound) keyed by genotype class (homozygous/heterozygous).

filter_passing_reads_count
int or None – Filtered depth of coverage of this sample at this site from the DP field.

genotype_confidence
str – Genotype quality (confidence) of this sample at this site, from the GQ field.

class VAPr.vcf_genotype_fields_parsing.**VCFGenotypeParser**
Bases: object

Mine format string and genotype fields string to create a filled VCFGenotypeInfo object.

FILTERED_ALLELE_DEPTH_TAG = 'DP'

GENOTYPE_QUALITY_TAG = 'GQ'

GENOTYPE_TAG = 'GT'

NORMALIZED_SCALED_LIKELIHOODS_TAG = 'PL'

UNFILTERED_ALLELE_DEPTH_TAG = 'AD'

static is_valid_genotype_fields_string(*genotype_fields_string*)
Return true if input has any real genotype fields content, false if is just periods, zeroes, and delimiters.

Parameters **genotype_fields_string** (*str*) – A VCF-style genotype fields string, such as 1/1:0,2:2:6:89,6,0 or ./.:.:.:.:.

Returns bool: true if input has any real genotype fields content, false if is just periods, zeroes, and delimiters.

classmethod parse(*format_key_string*, *format_value_string*)
Parse the input format string and genotype fields string into a filled VCFGenotypeInfo object.

Parameters

- **format_key_string** (*str*) – The VCF format string (e.g., 'GT:AD:DP:GQ:PL') for this sample at this site.
- **format_value_string** (*str*) – The VCF genotype fields values string (e.g., '1/1:0,34:34:99:1187.2,101,0') corresponding to the format_key_string for this sample at this site.

Returns

A filled VCFGenotypeInfo for this sample at this site unless an error was encountered, in which case None is returned. encountered, in which case None is returned.

Return type *VCFGenotypeInfo* or None

5.9 VAPr.vcf_merging module

VAPr.vcf_merging.**bgzip_and_index_vcf**(*vcf_path*)
bgzip and index each vcf so it can be merged with bcftools.

VAPr.vcf_merging.**merge_vcfs**(*input_dir*, *output_dir*, *project_name*, *raw_vcf_path_list=None*, *vcfs_gzipped=False*)
Merge vcf files into single multisample vcf, bgzip and index merged vcf file.

5.10 Module contents

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

V

Index

Symbols

A

B

C

D

E

F

N

O

P

R

S

T

U

V

W