MV-Test Documentation

Release 1.0.0

Todd Edwards, Chun Li and Eric Torstenson

October 09, 2015

Mean-Variance Test (MVTest) is an analysis tool for use with GWAS data.

Contents:

CHAPTER

ONE

INSTALLATION

MVTest requires python 2.7 or later (but 3.0 and later) as well as the following packages:

- NumPy (version 1.7.2 or later) www.numpy.org
- SciPY (version 0.13.2 or later) www.scipy.org

If these aren't already installed, and you don't have root access to the machine, please see the section, *Miniconda* or *Virtual Env* for easy instructions on different ways of installing tools as a restricted user. MVTest's installer can install these for you, however, it assumes that you have write access to your python library, which will not be the case by default on shared systems.

Download the package at: TODO: URL

To install the software, run the setup script as shown below: $ *python setup.py install*

If no errors are reported, it should be installed and ready to use.

1.1 System Requirements

Aside from the library dependencies, MVTest's requirements depend largely on the number of SNPs and individuals being analyzed as well as the data format being used. In general, GWAS sized datasets will require several gigabytes of memory when using the traditional pedigree format, however, even 10s of thousands of subjects can be analyzed with less than 1 gigabyte of RAM when the data is formatted as transposed pedigree or PLINK's default bed format.

Otherwise, it is recommended that the system be run on a unix-like system such as Linux or OS X, but it should work under windows as well (this is not a fully supported platform).

1.2 Running Unit Tests

MVTest comes with a unit test suite which can be run prior to installation. To run the tests, simply run the following command from within the root directory of the extracted archive's contents:

$ *python setup.py test*

If no errors are reported, then mvtest should run correctly on your system.

1.3 Virtual Env

Virtual ENV is a powerful too for python programers and users alike, as it allows for users to deploy different versions of python applications without the need for root access to the machine.

Because MVTest requires version 2.7, you'll need to ensure that your machine's python version is in compliance. Virtual Env basically uses the the system version of python, but creates a user owned environment wrapper allowing users to install libraries easily without administrative rights to the machine.

For a helpful introduction to VirtualEnv, please have a look at the tutorial: http://www.simononsoftware.com/virtualenv-tutorial/

1.4 Miniconda

Miniconda is a minimal version of the package manager used by the Anaconda python distribution. It makes it easy to create local installations of python with the latest versions of the common scientific libraries for users who don't have root access to their target machines.

Firstly, download and install the appropriate version of miniconda at the project website. Please be sure to choose the Python 2 version: http://conda.pydata.org/miniconda.html

While it is doing the installation, please allow it to update your PATH information. Also, be sure to follow directions such as starting a new shell to allow those changes to take effect.

Once those changes have taken effect, install setuptools and scipy: $ *conda install pip scipy*

Installing SciPy will also force the installation of NumPy, which is also required for running mvtest. (setuptools includes easy_install).

Once that has been completed successfully, you should be ready to follow the standard instructions for installing mvtest.

CHAPTER

TWO

WHAT IS MVTEST?

TODO: Write some background information about the application and it's scientific basis.

2.1 Documentation

Documentation for mvtest is still under construction. However, the application provides reasonable inline help using standard unix help arguments:

> *mvtest.py -h*

or

> *mvtest.py --help*

In general, overlapping functionality should mimic that of PLINK.

Due to automatic conversion of two dashes, "--", into an emdash (single long dash) when writing to PDF, you may need to

2.2 Command-Line Arguments

Command line arguments used by mvtest often mimick those used by PLINK, except where there is no matching functionality (or the functionality differs significantly.)

For the parameters listed below, when a parameter requires a value, the value must follow the argument with a single space separating the two (no '=' signs.) For flags with no specified value, passing the flag indicates that condition is to be "activated".

2.2.1 Getting help

Flag(s)	Type	Description
-h, --help		show this help message and exit
-v		Print version number

2.2.2 Input Data

MVTest attempts to mimic the interface for PLINK where appropriate.

All input files should be whitespace delimited. For text based allelic annotations, 1|2 and A|C|G|T annotation is sufficient. All data must be expressed as alleles, not as genotypes (except for IMPUTE output, which is a specialized format that is very different from the other forms).

For Pedigree, Transposed Pedigree and PLINK binary pedigree files, the using the prefix arguments is sufficient and recommended if your files follow the standard naming conventions.

Pedigree Data

Pedigree data is fully supported, however it is not recommended. When loading pedigree data, mvtest must load the entire dataset into memory prior to analysis, which can result in a substantial amount of memory overhead that is unnecessary.

Flags like --no-pheno and --no-sex can be used in any combination creating MAP files with highly flexible header structures.

Flag(s)	Type	Description
--file FILE	file prefix	Prefix for .ped and .map files
--ped PED	filename	PLINK compatible .ped file
--map MAP	filename	PLINK compatible .map file
--map3		MAP file has only 3 columns
--no-sex		Pedigree file doesn't have column 5 (sex)
--no-parents		Pedigree file doesn't have columns 3 and 4 (parents)
--no-fid		Pedigree file doesn't have column 1 (family ID)
--no-pheno		Pedigree file doesn't have column 6 (phenotype
--liability		Pedigree file has column 7 (liability)

PLINK Binary Pedigree

This format represents the most efficient storage for large GWAS datasets, and can be used directly by mvtest. In addition to a minimal overhead, plink style bed files will also run very quickly, due to the efficient disk layout.

Flag(s)	Type	Description
--bfile FILE	file prefix	Prefix for .bed, .bim and .fam files
--bed BED	filename	Binary Ped file (.bed)
--bim MAP	filename	Binary ped marker file (.bim)
--fam FAM	filename	Binary ped family file (.fam)

Transposed Pedigree Data

Transposed Pedigree data is similar to standard pedigree except that the data is arranged such that the data is organized as SNPs as rows, instead of individuals. This allows mvtest to run it's analysis without loading the entire dataset into memory.

Flag(s)	Type	Description
--tfile FILE	file prefix	Prefix for .tped and .tfam files
--tped BED	filename	Transposed Pedigree file (.tped)
--tfim MAP	filename	Transposed pedigree Family file (.tfam)

Pedigree/Transposed Pedigree Common Flags

By default, Pedigree and Transposed Pedigree data is assumed to be uncompressed. However, mvtest can directly use gzipped data files if they have the extension .tgz with the addition of the --compressed argument.

Flag(s)	Type	Description
--compressed	Ped/TPed	compressed with gzip (named .ped.tgz or .tped.tgz)

IMPUTE output

MVTest doesn't call genotypes when performing analysis, and allows users to define which model to use when analyzing the data. Due to the fact that there is no specific location for chromosome within the input files, mvtest requires that users provide chromosome, impute input file and the corresponding .info file for each imputed output.

Due to the huge number of expected loci, mvtest allows users to specify an offset and file count for analysis. This is to allow users to run multiple jobs simultaneously on a cluster and work individually on separate impute region files. Users can segment those regions even further using standard mvtest region selection as well.

By default, all imputed data is assumed to be compressed using gzip.

Default naming convention is for impute data files to end in .gen.gz and the info files to have the same name except for the end being replaced by .info.

Flag(s)	Type	Description
--impute IMPUTE	file-name	File containing list of impute output for analysis
--impute-fam IMPUTE_FAM	file-name	File containing family details for impute data
--impute-offset IMPUTE_OFFSET	int	Impute file index (1 based) to begin analysis
--impute-count IMPUTE_COUNT	int	Number of impute files to process (for this node). Defaults to all remaining.
--impute-uncompressed		Indicate that the impute input is not gzipped, but plain text
--impute-encoding {additive,dominant,recessive}	selec-tion	Genetic model to be used when analyzing imputed data.
--impute-info-ext IMPUTE_INFO_EXT	file prefix	Portion of filename denotes info filename
--impute-gen-ext IMPUTE_GEN_EXT	file suffix	Portion of filename that denotes gen file
--impute-info-thresh IMPUTE_INFO_THRESH	float	Threshold for filtering imputed SNPs with poor 'info' values

Phenotype/Covariate Data

Phenotypes and Covariate data can be found inside either the standard pedigree headers or within special PLINK style covariate files. Users can specify phenotypes and covariates using either header names (if a header exists in the file) or by 1 based column indices.

Flag(s)	Type	Description
--pheno PHENO	file-name	File containing phenotypes. Unless --all-pheno is present, user must provide either index(s) or label(s) of the phenotypes to be analyzed.
--mphenos MPHENOS	num-bers	Column number(s) for phenotype to be analyzed if number of columns > 1. Comma separated list if more than one is to be used.
--pheno-names PHENO_NAMES	string	Name for phenotype(s) to be analyzed (must be in --pheno file). Comma separated list if more than one is to be used.
--covar COVAR	file-name	File containing covariates
--covar-numbers COVAR_NUMBERS	num-bers	Comma-separated list of covariate indices
--covar-names COVAR_NAMES		Comma-separated list of covariate names
--sex		Use sex from the pedigree file as a covariate
--missing-phenotype MISSING_PHENOTYPE	char-acter	Encoding for missing phenotypes as can be found in the data.
--all-pheno		When present, mv-test will run each phenotypes found inside the phenotype file.

2.2.3 Restricting regions for analysis

When specifying a range of positions for analysis, a chromosome must be present. If a chromosome is specified but is not accompanied by a range, the entire chromosome will be used. Only one range can be specified per run.

Flag(s)	Type	Description
--snps SNPS	string	Comma-delimited list of SNP(s): rs1,rs2,rs3-rs6
--chr N	int	Select Chromosome. If not selected, all chromosomes are to be analyzed.
--from-bp START	int	SNP range start
--to-bp END	int	SNP range end
--from-kb START	int	SNP range start
--to-kb END	int	SNP range end
--from-mb START	int	SNP range start
--to-mb END	int	SNP range end
--exclude EXCLUDE	string	Comma-delimited list of rsids to be excluded
--remove REMOVE	string	Comma-delimited list of individuals to be removed from analysis. This must be in the form of family_id:individual_id
--maf MAF	float	Minimum MAF allowed for analysis
--max-maf MAX_MAF	float	MAX MAF allowed for analysis
--geno GENO	int	MAX per-SNP missing for analysis
--mind MIND	int	MAX per-person missing
--verbose		Output additional data details

CHAPTER

THREE

MV-TEST AUTHORS

MVTest is written and maintained by Eric Torstenson <eric.s.torstenson@vanderbilt.edu> based on the algorithm developed by Todd Edwards <todd.l.edwards@vanderbilt> and Chun Li <cxl791@case.edu>.

Much of the command line interface is mimicks that of PLINK http://pngu.mgh.harvard.edu/~purcell/plink/ in order to make it easy for researchers to be able to quickly integrate MVTest into their workflow.

CHAPTER

FOUR

MEANVAR

The following represents the API functionality associated with the meanvar application which includes a single interface for extracting data from each of the supported file types (pygwas). The contents below are only of interest for those who wish to extend MV-Test or utilize PyGWAS in their own GWAS analysis programs.

4.1 meanvar package

4.1.1 Submodules

4.1.2 meanvar.mv_esteq module

meanvar.mv_esteq.**MeanVarEstEQ**(*y*, *x*, *covariates*, *tol=1e-08*)

Perform the mean var calculation using estimated equestions

Parameters

- **y** -- Outcomes
- **x** -- [genotypes, cov1, ..., covN]
- **tol** -- convergence criterion

meanvar.mv_esteq.**RunAnalysis**(*dataset*, *pheno_covar*)

Run the actual analysis on all valid loci for each phenotype

Parameters

- **dataset** -- GWAS parser object
- **pheno_covar** -- holds all of the variables

This acts as a standard iterator, returning a single MVResult for each locus/phenotype combination.

Missing is evaluated as anything missing in any of the phenotype, covariate(s) or genotype

meanvar.mv_esteq.**RunMeanVar**(*pheno*, *geno*, *covar=[]*)

Setup and execute the mean var calculation.

Parameters

- **pheno** -- Phenotype data (one phenotype at a time)
- **geno** -- SNP data (might be genotypes, or dosages, etc)
- **covar** -- List of covariate data

It is possible that the optimization will fail to converge. Such cases are stripped of data, but are still reported to alert the user that there were problems with the data.

4.1.3 meanvar.mvresult module

class meanvar.mvresult.**MVResult**(*chr*, *pos*, *rsid*, *maj*, *min*, *eff_alcount*, *non_miss_count*, *p_mvtest*, *ph_label*, *beta_values*, *pvalues*, *stderrors*, *maf*, *covar_labels=[]*, *lm=-1*, *runtime=-1*)

Bases: object

Result associated with a single locus/phenotype execution

beta_pvalues = None
list of beta pvalues

beta_stderr = None
list of std errors

betas = None
list of beta values

chr = None
Chromosome

covar_labels = None
Covariate labels used for analysis

eff_alcount = None
Total count of effect alleles

lmpv = None
LM

maf = None
minor allele frequency

maj_allele = None
Major allele (A,C,G,T, etc)

min_allele = None
Minor allele

non_miss = None
non missing count

p_mvtest = None
mvtest's pvalue

p_variance

ph_label = None
current phenotype label

pos = None
BP position

print_header(*f=<open file '<stdout>', mode 'w'>*, *verbose=False*)
Prints header to f (will write header based on verbose)

Parameters

- **f** -- stream to print output
- **verbose** -- print all data or only the most important parts?

print_result(*f=<open file '<stdout>', mode 'w'>*, *verbose=False*)
Print result to f

Parameters

- **f** -- stream to print output
- **verbose** -- print all data or only the most important parts?

rsid = None
RSID

runtime = None
number of seconds analysis took to complete

stringify(*value*)

4.1.4 meanvar.mvstandardizer module

class meanvar.mvstandardizer.**Standardizer**(*pc*)
Bases: *pygwas.standardizer.StandardizedVariable*

Optional plugin object that can be used to standardize covariate and phenotype data.

Many algorithms require that input be standardized in some way in order to work properly, however, rescaling the results is algorithm specific. In order to facilitate this situation, application authors can write up application specific Standardization objects for use with the data parsers.

destandardize(*estimates*, *se*, ***kwargs*)
Revert the betas and variance components back to the original scale.

standardize()
Standardize the variables within a range [-1.0 and 1.0]

This replaces the local copies of this data. When it's time to scale back, use destandardize from the datasource for that.

4.1.5 meanvar.simple_timer module

class meanvar.simple_timer.**SimpleTimer**
Simple abstraction to allow for basic timing.

report(*msg*, *do_reset=False*, *file=<open file '<stdout>', mode 'w'>*)
Print to stdout msg followed by the runtime.

When true, do_reset will result in a reset of start time.

reset()
Reset start time

result(*msg*, *do_reset=False*)
Return log message containing ellapsed time as a string.

When true, do_reset will result in a reset of start time.

runtime()
Return ellapsed time and reset start.

4.1.6 Module contents

4.2 pygwas package

4.2.1 Submodules

4.2.2 pygwas.bed_parser module

class pygwas.bed_parser.**Parser**(*fam*, *bim*, *bed*)

Bases: *pygwas.transposed_pedigree_parser.Parser*

ReportConfiguration(*file*)

Report configuration for logging purposes.

Parameters **file** -- Destination for report details

Returns None

alleles = None

Alleles for each locus

bed_file = None

Filename associated with the binary allele information (in variant major format only)

bim_file = None

filename for marker info in PLINK .bim format

extract_genotypes(*bytes*)

Extracts encoded genotype data from binary formatted file.

Parameters **bytes** -- array of bytes pulled from the .bed file

Returns standard python list containing the genotype data

Only ind_count genotypes will be returned (even if there are a handful of extra pairs present).

fam_file = None

Filename associated with the pedigree data (first 6 columns from standard pedigree: fid, iid, fid, mid, sex, pheno)

families = None

Pedigree information for reporting

filter_missing()

Filter out individuals and SNPs that have too many missing to be considered

Returns None

This must be run prior to actually parsing the genotypes because it initializes the following instance members:

•ind_mask

•total_locus_count

•locus_count

•data_parser.boundary (adds loci with too much missingness)

geno_conversions = None

Genotype conversion

genotype_file = None
Actual pedigree file being parsed (file object)

ind_count = None
Number of valid individuals

ind_mask = None
Mask indicating valid samples

init_genotype_file()
Resets the bed file and preps it for starting at the start of the genotype data

Returns to beginning of file and reads the version so that it points to first marker's info

Returns None

load_bim(*map3=False*)
Basic marker details loading.

(chr, rsid, gen. dist, pos, allelel 1, allele2)

Parameters **map3** -- When true, ignore the genetic distance column

Returns None

load_fam(*pheno_covar*)
Load contents from the .fam file, updating the pheno_covar with family ids found.

Parameters **pheno_covar** -- Phenotype/covariate object

Returns None

load_genotypes()
Prepares the file for genotype parsing.

Returns None

markers = None
Valid loci to be used for analysis

populate_iteration(*iteration*)
Parse genotypes from the file and iteration with relevant marker details.

Parameters **iteration** -- ParseLocus object which is returned per iteration

Returns True indicates current locus is valid.

StopIteration is thrown if the marker reaches the end of the file or the valid genomic region for analysis.

4.2.3 pygwas.boundary module

class pygwas.boundary.**BoundaryCheck**(*bp=(None, None)*, *kb=(None, None)*, *mb=(None, None)*)
Bases: object

Record boundary specifications from user to control traversal.

Default boundaries are specified in numerical positions along a single chromosome. Users are permitted to provide boundaries in 3 forms: Bases, Kilobasees and Megabases. All are recorded as single base offsets from the beginning of the chromosome (starting at 1).

The valid setting doesn't mean the boundary object is invalid, only that no actual boundary ranges have been provided. This is done to allow the user interface code to be a little simpler (i.e. if the user didn't provide bounds using numerical boundaries, it can try instantiating a SnpBoundary and pass the relevant arguments to that object.

If none are valid, then either can be used, at which point both act as chromosome boundaries or simple SNP filters)

If chrom is specfied, all SNPs and boundaries are expected to reside on that chromosome.

LoadExclusions(*snps*)
Load locus exclusions.

Parameters **snps** -- Can either be a list of rsids or a file containing rsids.

Returns None

If snps is a file, the file must only contain RSIDs separated by whitespace (tabs, spaces and return characters).

LoadSNPs(*snps=[]*)
Define the SNP inclusions (by RSID). This overrides true boundary definition.

Parameters **snps** -- array of RSIDs

Returns None

This doesn't define RSID ranges, so it throws InvalidBoundarySpec if it encounters what appears to be a range (SNP contains a "-")

NoExclusions()
Determine that there is no exclusion criterion in play

Returns True if there is no real boundary specification of any kind.

This is used to avoid having to unnecessarily deal with missingmissness at the SNP level, when there isn't any to begin with.

ReportConfiguration(*f*)
Report the boundary configuration details

Parameters **f** -- File (or standard out/err)

Returns None

TestBoundary(*chr*, *pos*, *rsid*)
Test if locus is within the boundaries and not to be ignored.

Parameters

- **chr** -- Chromosome of locus
- **pos** -- BP position of locus
- **rsid** -- RSID (used to check for exclusions)

Returns True if locus isn't to be ignored

beyond_upper_bound = None
Is set once the upper limit has been exceeded

bounds = None
Actual boundary details in BP

chrom = -1

dropped_snps = None
Indices of loci that are to be dropped {chr=>[pos1, pos2, ..., posN]}

ignored_rs = None
List of RS Numbers to be ignored

target_rs = None
List of RS Numbers to be targeted (ignors all but those listed)

valid = None
True if boundary conditions remain true

4.2.4 pygwas.data_parser module

class pygwas.data_parser.**DataParser**
Bases: object

Abstract representation of all dataset parsers

boundary = <pygwas.boundary.BoundaryCheck object>
Boundary object specifying valid region for analysis

compressed_pedigree = False
When true, assume that standard pedigree and transposed pedigree are compressed with gzip

get_effa_freq(*genotypes*)

get_loci()

has_fid = True
When false, pedigree header expects no family id column

has_liability = False
When false, pedigree header expects no liability column

has_parents = True
When false, pedigree header expects no parents columns

has_pheno = True
When false, pedigree header expects no phenotype column

has_sex = True
When false, pedigree header expects no sex column

ind_exclusions = []
Filter out specific individuals by individual ID

ind_inclusions = []
Filter in specific individuals by individual ID

ind_miss_tol = 1.0
Filter individuals with too many missing

max_maf = 1.0
filter out if a minor allele frequency exceeds this value

min_maf = 0.0
this can be used to filter out loci with too few minor alleles

missing_representation = '0'
External representation of missingness

missing_storage = -1

snp_miss_tol = 1.0
Filter SNPs with too many missing

static valid_indid(*indid*)

pygwas.data_parser.**check_inclusions**(*item*, *included=[]*, *excluded=[]*)
Everything passes if both are empty, otherwise, we have to check if empty or is present.

4.2.5 pygwas.exceptions module

exception pygwas.exceptions.**InvalidBoundarySpec**(*malformed_boundary*)
Bases: *pygwas.exceptions.ReportableException*

Indicate boundary specification was malformed or non-sensical

exception pygwas.exceptions.**InvalidSelection**(*msg*)
Bases: *pygwas.exceptions.MalformedInputFile*

Indicate that the user provided input that is meaningless.

This is likely a situation where the user provided an invalid name for a phenotype or covariate. Probably a misspelling.

exception pygwas.exceptions.**InvariantVar**(*msg=''*)
Bases: *pygwas.exceptions.ReportableException*

No minor allele found

exception pygwas.exceptions.**MalformedInputFile**(*msg*)
Bases: *pygwas.exceptions.ReportableException*

Error encountered in data from an input file

exception pygwas.exceptions.**NanInResult**(*msg=''*)
Bases: *pygwas.exceptions.ReportableException*

NaN found in result

exception pygwas.exceptions.**NoMatchedPhenoCovars**(*msg=''*)
Bases: *pygwas.exceptions.ReportableException*

No ids matched between pheno or covar and the family data

exception pygwas.exceptions.**ReportableException**(*msg*)
Bases: exceptions.Exception

Simple exeception with message

exception pygwas.exceptions.**TooFewAlleles**(*chr=None*, *rsid=None*, *pos=None*, *alleles=None*, *index=None*)
Bases: *pygwas.exceptions.TooManyAlleles*

Indicate fixed allele was found

exception pygwas.exceptions.**TooManyAlleles**(*chr=None*, *rsid=None*, *pos=None*, *alleles=None*, *index=None*, *prefix='Too many alleles: '*)
Bases: *pygwas.exceptions.ReportableException*

Indicate locus found with more than 2 alleles

alleles = None
Allele 1 and 2

chr = None
Chromosome

index = None
Index of the locus within the file

pos = None
BP Position

rsid = None
RSID

exception pygwas.exceptions.**UnsolvedLocus**(*msg*)
Bases: *pygwas.exceptions.ReportableException*

4.2.6 pygwas.impute_parser module

class pygwas.impute_parser.**Encoding**
Bases: object

Simple enumeration for various model encodings

Additive = 0

Dominant = 1

Genotype = 3

Raw = 4

Recessive = 2

class pygwas.impute_parser.**Parser**(*fam_details*, *archive_list*, *chroms*, *info_files=[]*)
Bases: *pygwas.data_parser.DataParser*

Parse IMPUTE style output.

ReportConfiguration(*file*)

Parameters **file** -- Destination for report details

Returns None

archives = None
This is only the list of files to be processed

chroms = None
List of chroms to match files listed in archives

current_chrom = None
This will be used to record the chromosome of the current file

current_file = None
This will be used to record the opened file used for parsing

current_info = None
This will be used to record the info file associated with quality of SNPs

fam_details = None
single file containing the subject details (similar to plink's .fam)

gen_ext = 'gen.gz'
The genotype file suffix (of not following convention)

get_effa_freq(*genotypes*)
Returns the effect allele's frequency

get_next_line()
If we reach the end of the file, we simply open the next, until we run out of archives to process

info_ext = 'info'
the extension associated with the .info files if not using conventions

info_files = None
array of .info files

info_threshold = 0.4
The threshold associated with the .info info column

load_family_details(*pheno_covar*)
Load family data updating the pheno_covar with family ids found.

Parameters **pheno_covar** -- Phenotype/covariate object

Returns None

load_genotypes()
Prepares the files for genotype parsing.

Returns None

populate_iteration(*iteration*)
Parse genotypes from the file and iteration with relevant marker details.

Parameters **iteration** -- ParseLocus object which is returned per iteration

Returns True indicates current locus is valid.

StopIteration is thrown if the marker reaches the end of the file or the valid genomic region for analysis.

pygwas.impute_parser.**SetEncoding**(*sval*)
Sets the encoding variable according to the text passed

Parameters **sval** -- text specification for the desired model

4.2.7 pygwas.locus module

class pygwas.locus.**Locus**(*other=None*)
Bases: object

alleles = None
List of alleles present

chr = None
Chromosome

exp_hetero_freq
Returns the estimated frequency of heterozygotes

flip()
This will switch major/minor around, regardless of frequency truth.

This is intended for forcing one of two populations to relate correctly to the same genotype definitions. When flipped, Ps and Qs will be backward, and the maf will no longer relate to the "minor" allele frequency. However, it does allow clients to use the same calls for each population without having to perform checks during those calculations.

hetero_count = None
total count of heterozygotes observed

hetero_freq
Returns the frequency of observed heterozygotes (not available with all parsers)

maf
Returns the MAF. This is valid for all parsers

maj_allele_count = None
total number of major alleles observed

major_allele
Sets/Returns the encoding for the major allele (A, C, G, T, etc)

min_allele_count = None
total number of minor alleles observed

minor_allele
Sets/Returns the encoding for minor allele

missing_allele_count = None
total number of missing alleles were observed

p
Frequency for first allele

pos = None
BP Position

q
Frequency for second allele

rsid = None
RSID

sample_size
Returns to total sample size

total_allele_count
Returns the total number of alleles

4.2.8 pygwas.mach_parser module

class pygwas.mach_parser.**Encoding**
Bases: object

Dosage = 0
Currently there is only one way to interpret these values

class pygwas.mach_parser.**Parser**(*archive_list*, *info_files=[]*)
Bases: *pygwas.data_parser.DataParser*

Parse IMPUTE style output.

Due to the nature of the mach data format, we must load the data first into member before we can begin analyzing it. Due to the massive amount of data, SNPs are loaded in in chunks.

ISSUES:

- Currently, we will not be filtering on individuals except by explicit removal
- **We are assuming that each gzip archive contains all data associated with the loci contained within (i.e. there wor** be separate files with different subjects inside) ((Todd email jan-9-2015))
- **There is no reason to process regions in any order. I'm thinking we'll have a master file and then indices into tha** file and task count to facilitate "parallel" execution

- There is no place to store RSID from the output that I've seen (Minimac output generated by Ben Zhang)

ReportConfiguration(*file*)
Report the configuration details for logging purposes.

Parameters **file** -- Destination for report details

Returns None

chunk_stride = 50000
Number of loci to parse at a time (larger stride requires more memory)

dosage_ext = 'dose.gz'
Extension for the dosage file

get_effa_freq(*genotypes*)
Returns the frequency of the effect allele

info_ext = 'info.gz'
Extension for the info file

load_family_details(*pheno_covar*)
Load contents from the .fam file, updating the pheno_covar with family ids found.

Parameters **pheno_covar** -- Phenotype/covariate object

Returns None

load_genotypes()
Actually loads the first chunk of genotype data into memory due to the individual oriented format of MACH data.

Due to the fragmented approach to data loading necessary to avoid running out of RAM, this function will initialize the data structures with the first chunk of loci and prepare it for otherwise normal iteration.

Also, because the parser can be assigned more than one .gen file to read from, it will automatically move to the next file when the first is exhausted.

min_rsquared = 0.3
rsquared threshold for analysis (obtained from the mach output itself)

openfile(*filename*)

parse_genotypes(*lb*, *ub*)
Extracts a fraction of the file (current chunk of loci) loading the genotypes into memoery.

Parameters

- **lb** -- Lower bound of the current chunk
- **ub** -- Upper bound of the current chunk

Returns Dosage dosages for current chunk

populate_iteration(*iteration*)
Parse genotypes from the file and iteration with relevant marker details.

Parameters **iteration** -- ParseLocus object which is returned per iteration

Returns True indicates current locus is valid.

StopIteration is thrown if the marker reaches the end of the file or the valid genomic region for analysis.

This function will force a load of the next chunk when necessary.

4.2.9 pygwas.parsed_locus module

class pygwas.parsed_locus.**ParsedLocus**(*datasource*, *index=-1*)
Bases: *pygwas.locus.Locus*

Locus data representing current iteration from a dataset

Provide an iterator interface for all dataset types.

cur_idx = None
Index within the list of loci being analyzed

genotype_data = None
Actual genotype data for this locus

next()
Move to the next valid locus.

Will only return valid loci or exit via StopIteration exception

4.2.10 pygwas.pedigree_parser module

class pygwas.pedigree_parser.**Parser**(*mapfile*, *datasource*)
Bases: *pygwas.data_parser.DataParser*

Parse standard pedigree dataset.

Data should follow standard format for pedigree data, except alleles be either numerical (1 and 2) or as bases (A, C, T and G). All loci must have 2 alleles to be returned.

Attributes initialized to None are only available after load_genotypes() has been called.

Issues:

- Pedigree files are currently loaded in their entirety, but we could load them in according to chunks like we are doing in mach input.
- There are a bunch of legacy lists which should be reduced to a single list of Locus objects.

ReportConfiguration(*file*)
Report configuration for logging purposes.

Parameters **file** -- Destination for report details

Returns None

alleles = None
List of both alleles for each valid locus

datasource = None
Filename for the actual pedigree information

genotypes = None
Matrix of genotype data

get_loci()

individual_mask = None
Mask used to remove excluded and filtered calls from the genotype data (each position represents an individual)

invalid_loci = None
Loci that are being ignored due to filtration

load_genotypes (*pheno_covar*)
Load all data into memory and propagate valid individuals to pheno_covar.

Parameters **pheno_covar** -- Phenotype/covariate object is updated with subject

information :return: None

load_mapfile (*map3=False*)
Load the marker data

Parameters **map3** -- When true, ignore the gen. distance column

Builds up the marker list according to the boundary configuration

locus_count = None
Number of valid loci

mapfile = None
Filename for the marker information

markers = None
List of valid Locus Objects

markers_maf = None
List of MAF at each locus

populate_iteration (*iteration*)
Parse genotypes from the file and iteration with relevant marker details.

Parameters **iteration** -- ParseLocus object which is returned per iteration

Returns True indicates current locus is valid.

StopIteration is thrown if the marker reaches the end of the file or the valid genomic region for analysis.

rsids = None
List of all SNP names for valid loci

4.2.11 pygwas.pheno_covar module

class pygwas.pheno_covar.**PhenoCovar**
Bases: object

Store both phenotype and covariate data in a single object.

Provide iterable interface to allow evaluation of multiple phenotypes easily. Covariates do not change during iteration. Missing is updated according to the missing content within the phenotype (and covariates as well).

add_subject (*ind_id*, *sex=None*, *phenotype=None*)
Add new subject to study, with optional sex and phenotype

Throws MalformedInputFile if sex is can't be converted to int

covariate_data = None
All covariate data [[cov1],[cov2],etc]

covariate_labels = None
List of covariate names from header, if provided SEX is implied, if sex_as_covariate is true. Covariates loaded without header are simply named Cov-N

destandardize_variables (*tv*, *blin*, *bvar*, *errBeta*, *nonmissing*)
Destandardize betas and other components.

do_standardize_variables = None
Allows you to turn off standardization

freeze_subjects()
Converts variable data into numpy arrays.

This is required after all subjects have been added via the add_subject function, since we don't know ahead of time who is participating in the analysis due to various filtering possibilities.

individual_mask = None
True indicates an individual is to be excluded

load_covarfile(*file*, *indices=[]*, *names=[]*, *sample_file=False*)
Load covariate data from file.

Unlike phenofiles, if we already have data, we keep it (that would be the sex covariate)

load_phenofile(*file*, *indices=[]*, *names=[]*, *sample_file=False*)
Load phenotype data from phenotype file

Whitespace delimited, FAMID, INDID, VAR1, [VAR2], etc

Users can specify phenotypes of interest via indices and names. Indices are 1 based and start with the first variable. names must match name specified in the header (case is ignored).

missing_encoding = -9
Internal encoding for missingness

pedigree_data = None
Pedigree information {FAMID:INDID => index, etc}

phenotype_data = None
Raw phenotype data with every possible phenotype [[ph1],[ph2],etc]

phenotype_names = None
List of phenotype names from header, if provided. If no header is found, the phenotype is simply named Pheno-N

prep_testvars()
Make sure that the data is in the right form and standardized as expected.

sex_as_covariate = False
Do we use sex as a covariate?

test_variables = None
finalized data ready for analysis

4.2.12 pygwas.snp_boundary_check module

class pygwas.snp_boundary_check.**SnpBoundaryCheck**(*snps=[]*)
Bases: *pygwas.boundary.BoundaryCheck*

RS (or other name) based boundary checking.

Same rules apply as those for BoundaryCheck, except users can provide multiple RS boundary regions. Though, all boundary groups must reside on a single chromosome.

Class members (these are not intended for public consumption):

- start_bounds bp location for boundary starts Currently, only one boundary is permitted. This is to remain consistant with plink
- end_bounds bp location for boundary end (inclusive)

- ignored_rs List of RS numbers to be ignored
- target_rs List of RS numbers to be targeted
- **dropped_snps indices of loci that are to be dropped** {chr=>[pos1, pos2, ...]}
- **end_rs This is used during iteration to identify when to** turn "off" the current boundary group

NoExclusions()
Determine that there is no exclusion criterion in play

Returns True if there is no real boundary specification of any kind.

This is used to avoid having to unnecessarily deal with missingmissness at the SNP level, when there isn't any to begin with.

ReportConfiguration(*f*)
Report the boundary configuration details

Parameters **f** -- File (or standard out/err)

Returns None

TestBoundary(*chr*, *pos*, *rsid*)
Test if locus is within the boundaries and not to be ignored.

Parameters

- **chr** -- Chromosome of locus
- **pos** -- BP position of locus
- **rsid** -- RSID (used to check for exclusions)

Returns True if locus isn't to be ignored

4.2.13 pygwas.standardizer module

class pygwas.standardizer.**NoStandardization**(*pc*)
Bases: *pygwas.standardizer.StandardizedVariable*

This is mostly a placeholder for standardizers. Each application will probably have a specific approach to standardizing/destandardizing the input/output.

destandardize(*estimates*, *se*, ***kwargs*)
When the pheno/covar data has been standardized, this can be used to rescale the betas back to a meaningful value using the original data.

For the "Un-standardized" data, we do no conversion.

standardize()
Standardize the variables within a range [-1.0 and 1.0]

This replaces the local copies of this data. When it's time to scale back, use destandardize from the datasource for that.

class pygwas.standardizer.**StandardizedVariable**(*pc*)
Bases: object

Optional plugin object that can be used to standardize covariate and phenotype data.

Many algorithms require that input be standardized in some way in order to work properly, however, rescaling the results is algorithm specific. In order to facilitate this situation, application authors can write up application specific Standardization objects for use with the data parsers.

covar_count = None
number of covars

covariates = None
Standardized covariate data

datasource = None
Reference back to the pheno_covar object for access to raw data

destandardize()
Stub for the appropriate destandardizer function.

Each object type will do it's own thing here.

get_covariate_name(*idx*)
Return label for a specific covariate

Parameters **idx** -- which covariate?

Returns string label

get_covariate_names()
Return all covariate labels as a list

Returns list of covariate names

get_phenotype_name()
Returns current phenotype name

get_variables(*missing_in_geno=None*)
Extract the complete set of data based on missingness over all for the current locus.

Parameters **missing_in_geno** -- mask associated with missingness in genotype

Returns (phenotypes, covariates, nonmissing used for this set of vars)

idx = None
index of the current phenotype

missing = None
mask representing missingness (1 indicates missing)

pheno_count = None
number of phenotypes

phenotypes = None
standardized phenotype data

standardize()
Stub for the appropriate standardizer function

Each Standardizer object will do it's own thing here.

pygwas.standardizer.**get_standardizer**()

pygwas.standardizer.**set_standardizer**(*std*)

4.2.14 pygwas.transposed_pedigree_parser module

class pygwas.transposed_pedigree_parser.**Parser**(*tfam*, *tped*)
Bases: *pygwas.data_parser.DataParser*

Parse transposed pedigree dataset

Class Members: tfam_file filename associated with the pedigree information tped_file Filename associated with the genotype data families Pedigree information for reporting genotype_file Actual pedigree file begin parsed (file object)

ReportConfiguration(*file*)

filter_missing()
Filter out individuals and SNPs that have too many missing to be considered

load_genotypes()
This really just intializes the file by opening it up.

load_tfam(*pheno_covar*)
Load the pedigree portion of the data and sort out exclusions

populate_iteration(*iteration*)
Pour the current data into the iteration object

process_genotypes(*data*)
Parse pedigree line and remove excluded individuals from geno

Translates alleles into numerical genotypes (0, 1, 2) counting number of minor alleles.

Throws exceptions if an there are not 2 distinct alleles

4.2.15 Module contents

pygwas.**BuildReportLine**(*key*, *value*)
Prepare key/value for reporting in configuration report

Parameters

- **key** -- configuration 'keyword'
- **value** -- value reported to be associated with keyword

Returns formatted line starting with a comment

pygwas.**Exit**(*msg*, *code=1*)
Exit execution with return code and message :param msg: Message displayed prior to exit :param code: code returned upon exiting

pygwas.**ExitIf**(*msg*, *do_exit*, *code=1*)
Exit if do_exit is true

Parameters

- **msg** -- Message displayed prior to exit
- **do_exit** -- exit when true
- **code** -- application's return code upon exit

pygwas.**sys_call**(*cmd*)
Execute cmd and capture stdout and stderr

Parameters **cmd** -- command to be executed

Returns (stdout, stderr)

CHAPTER

FIVE

CHANGE LOG

mvtest.py: 1.0.0 released

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

m

p

G

H

I

L

M

N

O

P

Q

R

S

T

U

V