R documentation

of all in 'man/'

October 25, 2011

R topics documented:

`GWASBinTests-package`
p-values and FDRs for genomic regions in genome-wide association studies

---

Description

Bin-based analysis of genome-wide association studies

Details

Given an a priori partitioning of the genome into regions termed **bins**, GWASBinTests can compute p-values (and FDRs) of several association tests. Some are likelihood-based and use genotyping error models that take into account genotyping errors and missing data, some can take into account the correlation pattern of the neighbor markers, and some are based on sum-scores.

To be more precise, the available tests are:

univariate Univariate Genotypic Likelihood, corresponding to the **L3** score in the article. It is computed on genotype contigency tables, and it does not take into account the correlation between markers of the same **bin**.

divariate Divariate Genotypic Likelihood, corresponding to the **L2** score in the article. It is computed on genotype contigency tables, and use the correlation between consecutive pairs of markers.

allelic Allelic SumScore test. It is based on the sum of the allelic (i.e. computed on the allelic contingency tables) pearson scores of all the markers of a bin. It should be very close to the set-based test implemented in **PLINK** when using the parameters `--set-max 99999 --set-p 1 --set-r2 1`

genotypic Genotypic SumScore test. Like the allelic one but computed on genotypic tables.

usage: GWASBinTests data are based on `GenABEL` datasets, and are called **gws**. The genotypes are stored using the gwaa-dataclass and saved in **raw** files, and the phenotype data in a data frame stored in a tabulated file. GWASBinTests C++ engine can either be run directly on those data files, or on R data.

The first step is to get and prepare some data. You can use samples from GenABEL (`srdta`) or directly from GWASBinTests (`ms1`). You can convert data sets from PLINK format using `readPlinkTransposedData`, or convert genotypes using the functions provided by `GenABEL`. If you use a dataset from `GenABEL`, you have to adapt it, basically to set some phenotypic variables that are used by GWASBinTests (see `asGws`).

Then you need a bins description file (see `readBins`). There are is one available from the data/ directory of GWASBinTests: `system.file("data/ms1.bins", package = "GWASBinTests")`

Now you can run the GWASBinTests analysis, using `processFiles` or `processGws`. Please take a look at the parameters (see `parameters`), that can be fined tuned for performance and accuracy.

In the end you obtain a dataframe with all the results.

You may also *play* with hand-made tables to test the different p-values using `processTable`

Implementation:

GWASBinTests is developped for reliability and efficiency. To achieve these goals we developed two implementations of the tests. The first is optimized for simplicity, readability and conciseness and coded purely in R. It serves as a reference implementation, exhaustively tested with the included unit tests. The second implementation is optimized for performance and thus coded in C++ and integrated in R via \textitRcpp.

The pure-R implementation is too slow to be used on a big dataset, but can be used to validate the c++ optimized implementation. There is also a standalone C++ executable not distributed for now but that available on request. The integration with R has been greatly facilitated thanks to `Rcpp`. We make use of the **BOOST** C++ libraries, especially of the Binomial Distribution implementation of confidence intervals.

We put a lot of efforts into the computational speed of the analysis,:

1. by carefully optimizing the bottlenecks of the code, especially the Random Number Generator and the computation of the contingency tables
2. by using multiple **threads**. We used **OpenMP** to implement the multihreading, because it is easy and allow to write code that is both sequential and multithreaded. OpenMP should be available with GCC $\geq$ 4.2. The R package should detect automatically if OpenMP is available and compile the code accordingly. See the `threads` parameter in `parameters` to control the number of threads that are used.
3. by implementing **heuristics** on the number of permutations to use to avoid useless rounds of computation. See `parameters` for a description of these heuristics.

\subsubsectionRandom Number Generator

One of the bottlenecks was the Random Number Generator, both for speed and for the cycle length because we may use a lot of permutations. Moreover we wanted to have reproducible results, even when multithreading. All of these objectives have been met by using a modified version of the **CRandomSFMT0** generator of the **Randomc** c++ library, which is "an improvement of the Mersenne Twister with better randomness and higher speed, designed specifically for processors with Single-Instruction-Multiple-Data (SIMD) capabilities, such as the SSE2 and later instruction set". We further modified it by inlining some functions to achieve yet more speed. The cycle length is $\geq 2^{11213} - 1$ that allows a lot of independent permutations. An unusual feature is that we get the very exact same results (at a given fixed seed) in sequential or multithreaded modes, and thus obtain reproducible results, using multiple seeds thanks to *Randomc*.

The result is that now we can compute in minutes what took days before.

Author(s)

Nicolas Omont, Jerome Wojcik, Karl Forner

References

N.B: An application note for this package is in preparation.

Articles presenting the method:

Long version: http://videolectures.net/site/normal_dl/tag=8590/article10.pdf

Short version: http://www.biomedcentral.com/1753-6561/2/S4/S6

Additional resources: http://videolectures.net/msht07_omont_gbba/

See Also

PLINK http://pngu.mgh.harvard.edu/purcell/plink/, *Purcell S, Neale B, Todd-Brown K, Thomas L, Ferreira MAR, Bender D, Maller J, Sklar P, de Bakker PIW, Daly MJ & Sham PC (2007) PLINK: a toolset for whole-genome association and population-based linkage analysis. American Journal of Human Genetics, 81.* Dirk Eddelbuettel, Romain Francois, with contributions by Simon

BOOST free peer-reviewed portable C++ source libraries: http://www.boost.org/

Randoma package of random number generators: http://www.agner.org/random/ran-instructions.pdf

OpenMP API specification for parallel programming: http://openmp.org

---

0_Synopsis *Synopsis*

Description

a sample session

Details

data: Use readGws, readPlinkTransposedData and readBins to load/import your data. See the **GenABEL** documentation if you have data in other formats. Once you have a GenABEL dataset you can easily adapt it using asGws. Save your data with saveGws.

A sample session to get you started. Be sure to understand the parameters, especially those related to the heuristics, which can save you hours of computing time (cf **parameters**)

Examples

```
## Not run:

gws <- readGws("path/to/your_dataset")
bins <- readBins("path/to/your_bins_file.bins")

params <- parameters(verbosity=1, nb_permutations=1000000, min_pvalue=0.01, max_relative_
results <- processGws(gws, bins, params=params, covariables=your_covariables)

## End(Not run)
```

asGws — *Convert a gwaa dataset to a GWS dataset and perform some sanity checks...*

Description

Convert a gwaa dataset to a GWS dataset and perform some sanity checks

Usage

```
asGws(gwaa, assignPopNATo)
```

Arguments

gwaa	The gwaa.data-class object to convert
assignPopNATo	if defined, the NAs in the pop variable (or from the *bt* GenABEL variable if pop is not present) will be assigned to this value.

Details

GWASBinTests datasets are GenABEL datasets with some special variables in the phenotypic variable data, that we call GWS datasets. The main variables are:

pop the Population index - It is the phenotype, for example case or control. You can currently have up to 256 different populations. This has to be coded as integers and be ≤ 255

Some functions such as processGws use this variable as the default phenotype, and require the all the phenotypes have to be defined, i.e. no NA is allowed. If you have NAs in your data you may use the assignPopNATo parameter to assign those NAs either to an existing phenotype or to a new one. Otherwise you may subset your Gws dataset to only consider the individuas with defined phenotypes.

gws the Genome Wide Study index - It is used to perform *meta-analysis* of several studies, meaning that the permutations of the labels will be performed intra studies, i.e. a permutation will swap labels from samples inside a same study. See mergeGws for merging studies in a GWS dataset. This index has to be an integer.

Value

A gwaa.data-class object converted. In fact only the phenodata **sex** and **gws** might be modified.

Examples

```
data(srdta)
# make a GWS dataset by setting the pop to 0
gws <- asGws( srdta, assignPopNATo=0)
```

`concept#allelic.sum.score`
The allelic Sum of scores bin statistic

Description

The allelic Sum Of Scores statistic.

Details

It is based on the Pearson test-statistic on a contingency table (a.k.a chi-squared test) X^2.

For a bin b, a marker j, a given set of variables values $(v)_m$ selects a subset of the individuals I. On this subset I, we note the contingency table cell values counting the number of alleles per phenotype value $O(a, p)$ where $O(a, p)$ is the number of allele a for phenotype p.

Then the allelic score (Pearson score) for this table is:

$$X^2_{all}(b, j, (v)_m) = \sum_{\{a \in \{a,A\}, p \in P, E(a,p) \neq 0\}} \frac{(O(a, p) - E(a, p))^2}{E(a, p)}$$

so that

$$X^2_{all}(Bin\ b) = \sum_{marker_j \in b} \sum_{(v)_m \in (V)_m} X^2_{all}(b, j, (v)_m)$$

`concept#bin` *Definition of bins*

Description

A bin is a genomic region, defined by its chromosome and is start and end positions. It corresponds to a contiguous set of SNPs.

Details

Notations:

B is the number of bins. A given bin contains J SNPs.

`concept#divariate.likelihood`

The two-marker sliding windows (aka divariate) Likelihood function

---

Description

The two-marker sliding windows (aka divariate) likelihood is a one of the possible likelihood functions used to assess the association of bins using likelihood ratio (cf **concept#likelihood.ratio**)

Details

It is written L2 in the article.

For a given bin, a given set of variables (V)m, we define this likelihood function on a patient i as follows:

$$L_2(Patient_i) = \prod_{2 \leq j \leq J} \sum_{(g_1,g_2) \in [aa,Aa,AA]^2} p(o^j(i)|g_2).p(G_{(j-1,j)} = (g_1,g_2)|(v(i))_m).p(o^{j-1}(i)|g_1)$$

with $o^j(i)$ the observed genotype for marker j and patient i, $(v(i))_m$ the set of variable values of patient i and $p(G_{(j-1,j)} = (g_1,g_2)|(v(i))_m)$ the probability of having the two genotypes for markers j-1 and j.

If you include the phenotype in the set of variables you end up with $L2_{H_1}$ the likelihood under H1, and otherwise with $L2_{H_0}$ under H0.

As all patients are considered to be independently chosen, the likelihood of the set of patients available is:

$$L2(Bin_b) = \prod_{all\ patients\ i} L2(Patient_i) = \prod_{all\ patients\ i} \prod_{2 \leq j \leq J} \sum_{(g_1,g_2) \in [aa,Aa,AA]^2} p(o^j(i)|g_2).p(G_{(j-1,j)} = (g_1,g_2)|(v($$

We can swap the two products:

$$L2(Bin_b) = \prod_{2 \leq j \leq J} \prod_{all\ patients\ i} \sum_{(g_1,g_2) \in [aa,Aa,AA]^2} p(o^j(i)|g_2).p(G_{(j-1,j)} = (g_1,g_2)|(v(i))_m).p(o^{j-1}(i)|g_1)$$

So that if we define the likelihood L2 for a pair of consecutive markers (j-1, j):

$$L2(j-1,j) := \prod_{all\ patients\ i} \sum_{(g_1,g_2) \in [aa,Aa,AA]^2} p(o^j(i)|g_2).p(G_{(j-1,j)} = (g_1,g_2)|(v(i))_m).p(o^{j-1}(i)|g_1)$$

We can now compute likelihood L2 successively for each pair of consecutive markers, and still compute the likelihood of the bin:

$$L2(Bin_b) = \prod_{2 \leq j \leq J} L2(j-1,j)$$

`concept#genotype.error.model`

Modelization of genotyping errors

Description

An error model is introduced with observed genotypes $\mathcal{O}^j$ (with $\mathcal{O}^j \in \{aa, Aa, AA, \emptyset\}$, where $\emptyset$ means that the genotype is missing):

$$P(\mathcal{O}^j|(\mathcal{G}^l)_{l\in[1,J]}) = P(\mathcal{O}^j|\mathcal{G}^j)$$

Details

Indeed, the technology is the same for all determinations of the same marker and there is no correlation between the genomic order of SNPs and the localization of their probes on the genotyping chips, so that there is no reason why the observed genotype should depend on something else than the real genotype.

So basically, the error model for a given marker j is the set of conditional probabilities

$$P(\mathcal{O}^j\backslash\mathcal{G}^j) = \left(\begin{array}{r|c|c|c} \mathcal{O}^j\backslash\mathcal{G}^j & aa & Aa & AA \\ \hline aa & & & \\ \hline Aa & & & \\ \hline AA & & & \\ \hline \emptyset & & & \end{array}\right)$$

`concept#genotypic.sum.score`

The genotypic Sum of scores bin statistic

Description

The genotypic Sum Of Scores statistic.

Details

It is the equivalent of the concept#allelic.sum.score but computed on genotypic tables instead of allelic ones.

It is based on the Pearson test-statistic on a contingency table (a.k.a chi-squared test) X^2.

For a bin b, a marker j, a given set of variables values $(v)_m$ selects a subset of the individuals I. On this subset I, we note the contingency table cell values counting the number of genotypes per phenotype value $O(g, p)$ where $O(g, p)$ is the number of genotypes g for phenotype p.

Then the genotypic score (Pearson score) for this table is:

$$X^2_{gen}(b, j, (v)_m) = \sum_{\{g\in\{aa,Aa,AA\},p\in P,E(g,p)\neq 0\}} \frac{(O(g, p) - E(g, p))^2}{E(g, p)}$$

so that

$$X^2_{gen}(Bin\ b) = \sum_{marker_j\in b} \sum_{(v)_m\in(V)_m} X^2_{gen}(b, j, (v)_m)$$

`concept#likelihood.ratio`

The likelihood ratio

Description

A explained in the article, among the scores (or statistics) we consider and use to assess the association of a bin are the likelihood ratios.

Details

We use several different likelihood functions, that are defined on the patients, i.e on their observed genotypes for the SNPs of the bin (cf **concept#bin**) and their different covariable values.

These likelihood functions rely upon the probability distributions of genotypes $P((G)j|S,(V)m)$. Different hypotheses lead to different probability distributions hence to different likelihood values.

To compare hypothesis, the likelihood function will be evaluated for two different hypothesis: the H1 hypothesis considering the phenotype and the true probability distribution $P((G)j|S,(V)m)$, and the null hypothesis H0 that considers that the genotypes are independent from the phenotype S: $P_{H_0}((G)j|(V)m)$.

So we have two likelihoods L_{H_1} and L_{H_0} in function of the hypothesis/distribution considered, that define the likelihood ratio for a given patient i:

$$LR(Patient_i) = \frac{L_{H_1}(Patient_i)}{L_{H_0}(Patient_i)}$$

As all patients are considered to be independently chosen, the likelihood of the set of patients available is:

$$LR(Bin_b) = \prod_{all\ patients\ i} LR(Patient_i))$$

`concept#naive.likelihood`

The Naive (aka univariate) Likelihood function

Description

The naive or univariate likelihood is a one of the possible likelihood functions used to assess the association of bins using likelihood ratio (cf **concept#likelihood.ratio**)

Details

It is written L3 in the article, and named thereafter "naive likelihood" because it corresponds to a naive Bayesian model. It does not model the linkage disequilibrium considering markers as independents.

For a given bin, a given set of variables (V)m, we define this likelihood function on a patient i as follows:

$$L_3(Patient_i) = \prod_{all\ markers\ j} \sum_{g \in aa,Aa,AA} p(o^j(i)|g).p(g|(v(i))_m)$$

with $o^j(i)$ the observed genotype for marker j and patient i, and $(v(i))_m$ the set of variable values of patient i.

If you include the phenotype in the set of variables you end up with $L3_{H_1}$ the likelihood under H1, and otherwise with $L3_{H_0}$ under H0.

As all patients are considered to be independently chosen, the likelihood of the set of patients available is:

$$L3(Bin_b) = \prod_{all\ patients\ i} L3(Patient_i) = \prod_{all\ patients\ i} \prod_{all\ markers\ j} \sum_{g \in aa,Aa,AA} p(o^j(i)|g).p(g|(v(i))_m)$$

We can swap the two products:

$$L3(Bin_b) = \prod_{all\ markers\ j} \prod_{all\ patients\ i} \sum_{g \in aa,Aa,AA} p(o^j(i)|g).p(g|(v(i))_m)$$

So that if we define the likelihood L3 for a given marker j:

$$L3(Marker_j) := \prod_{all\ patients\ i} \sum_{g \in \{aa,Aa,AA\}} p(o^j(i)|g).p(g|(v(i))_m$$

We can now compute likelihood L3 marker by marker, and still compute the likelihood of the bin:

$$L3(Bin_b) = \prod_{all\ markers\ j} L3(Marker_j)$$

`concept#regularization.of.table.estimates`

Regularizations of the estimates from contingency tables

Description

To obtain more regular estimates, a constant is added to all cell counts. It is a Dirichlet prior on parameters. This constant is can be chosen to be $C = \alpha.\bar{n}$, where α is the chosen error rate and $\bar{n}$ is the mean number of individuals per cell. This constant means that uncertainty on low cell counts is high, not only because of randomness, but also because of genotyping errors.

`convertChromosomeNames`

convert GenABEL chromosome names to integer...

Description

convert GenABEL chromosome names to integer

Usage

```
convertChromosomeNames(chr_names)
```

Arguments

chr_names	a vector of chromosome names or integers

Details

Take a vector of chromosome names such as those use in a GenABEL dataset (chromosome and convert them to integer.

The non trivial conversions are: X=23, Y=24, XY=25, MT=26, NOTON(non localized)=0, MULTI(multi-localized)=-1 , UN(Unmapped)=-2, PAR(Pseudo Autosomal Region)=25, BAD(other error)=-100

N.B: In case of unknown name, the function will stop with an error.

Value

a vector of chromosome integer codes

Examples

```
chrs <- convertChromosomeNames(c("1", "6", "x", "Y", "xY"))
```

---

convertGenotypes *convert and check the genotypes to our simple pure-R implementation conventions...*

Description

convert and check the genotypes to our simple pure-R implementation conventions

Usage

```
convertGenotypes(genotypes)
```

Arguments

genotypes	a vector (or matrix) of integers with values meaning: **homozygous1 (AA)** 0 **heterozygous (AB)** 1 **homozygous2 (BB)** 2 **missing (∅)** NA or -1 . There is no assumption about which allele is minor.

Details

The genotypes must be an integer vector. The code for missing data is -1, so all NAs will be converted into -1

All other values will raise an error.

This format of genotypes is expected by all the simple_* functions. To extract genotypes from a Gws, rather use the fetchGenotypes and fetchGenotypesAsList functions that will call this function anyway.

Value

the cleaned and checked integer vector of genotypes, with values in -1:2

Examples

```
data(srdta)
gs <- convertGenotypes( as.integer( as.double(srdta[,1:20]) ) )
```

---

data#ms1 *A sample data set*

Description

This is the first 1000 SNPs of an internal and private dataset.

Details

The sample names have been anonymized.

The Bins are defined on DNA from protein genes as defined in the version 35:35 of EnsEMBL Birney et al. (2006) of the human DNA sequence. The basic region of a gene lies from the beginning of its first exon to the end of its last exon. Overlapping genes are clustered in the same bin. If two consecutive genes or clusters of overlapping genes are separated by less than 200 kbp, the bin limit is fixed in the middle of the interval. Otherwise, the limit of the upstream bin is set 50 kbp downstream its last exon, the limit of the downstream bin is set 50 kbp upstream its first exon, and a special bin corresponding to a desert is created in between the two bins. With these rules, desert bins have a minimum length of 100 kbp

It consists of:

- ms1.gag the genotypes
- ms1.gap the phenotypes
- ms1.bins the bins definitions

It can be loaded via the data() function: `data("ms1")`, and provides:

`ms1_gws` a *GWSified* `gwaa.data-class` dataset

`ms1_bins` a bins description dataframe, as returned by `readBins`

---

data#ms1_bins *See data#ms1...*

Description

See data#ms1

`data#ms1_gws` *See data#ms1...*

Description

See data#ms1

`data#plink_small` *A sample data set*

Description

This is a subset of an example dataset available from the Plink website

Details

It has been downloaded and converted from the the files wgas1.ped and wgas1.map from the example.zip file.

The command used to generate it is:

```
plink --chr 21 --from-kb 5000 --to-kb 20000 --recode --transpose --file wgas1 --out plink_small
```

It consists of:

- plink_small.tped the transposed genotypes
- plink.tfam the phenotypes

References

urlhttp://pngu.mgh.harvard.edu/~purcell/plink/res.shtml#teach

`fetchGenotypes` *fetch the genotypes from a GenABEL dataset in our format...*

Description

fetch the genotypes from a **GenABEL** dataset in our format

Usage

```
fetchGenotypes(gws, index)
```

Arguments

`gws`	the **GenABEL** dataset
`index`	the index of the SNP. If NULL (the default), will fetch genotypes for all the SNPs

Value

the genotypes, an integer matrix with values meaning:

homozygous1 (aa) 0

heterozygous (Aa) 1

homozygous2 (AA) 2

missing (∅) -1

Examples

```
data(srdta)
# one index
g <- fetchGenotypes(srdta,1)

# index range
gs <- fetchGenotypes(srdta, 1:10)

# names
gs <- fetchGenotypes(srdta, c("rs150", "rs179"))

# all
gs <- fetchGenotypes(srdta)
```

---

fetchGenotypesAsList

fetch the genotypes from a GenABEL dataset in our format as list of vectors...

Description

fetch the genotypes from a **GenABEL** dataset in our format as list of vectors

Usage

```
fetchGenotypesAsList(gws, index)
```

Arguments

gws	the **GenABEL** dataset
index	the index of the SNP(s). If NULL (the default), will fetch genotypes for all the SNPs

Details

See fetchGenotypes

Value

the genotypes, a list of genotypes integer vectors

gwsForBin *create a subset of a gws for a given bin...*

Description

create a subset of a gws for a given bin

Usage

```
gwsForBin(gws, chr, start, end)
```

Arguments

gws	the dataset as a gwaa.data-class object
chr	the chromosome of the bin
start	the start of the bin
end	the end of the bin

Value

a (sub) dataset as a gwaa.data-class object, or NULL

makeBinsForSnps *make a bin for every SNP of the dataset...*

Description

make a bin for every SNP of the dataset

Usage

```
makeBinsForSnps(gws)
```

Arguments

gws	the dataset as a gwaa.data-class object

Value

a sorted dataframe of bins (chr_name, start, end, bin_index)

mergeGws *Merge several genome wide association studies into a single dataset...*

Description

Merge several genome wide association studies into a single dataset

Usage

```
mergeGws(gwsA, gwsB, intersected_snps_only=TRUE)
```

Arguments

gwsA	First genome wide association study as a gwaa.data-class object.
gwsB	Second genome wide association study as a gwaa.data-class object.
intersected_snps_only	merge only SNPs shared by gwsA and gwsB

Details

Merge two association studies datasets as gwaa.data-class objects into a single one. The 'gws' column in phenodata keeps track of the sample study of origin.

N.B the sample ids have to be distinct !!!

N.B When using intersected_snps_only=FALSE, genotypes for samples for SNPs not in dataset will be set to NA.

When doing a GWASBinTests analysis, the method takes into account the study of origin via the **gws** variable by permuting the phenotypes of the labels intra-study.

N.B: the function is named mergeGws and not merge.gws to avoid R CMD check warnings about *S3 generic/method consistency* and the generic merge S3 function.

Value

A genome wide association dataset as a gwaa.data-class object. Only the following fields are merged:

pop	Case/Control status;
sex	Sex;
gws	Genome wide study identifier.

If the gws field is absent in one of the studies, it is added with a value different from the ones found in the other study.

missingRateFromGenotypes *compute the missing rate from the genotypes of a marker...*

Description

compute the missing rate from the genotypes of a marker

Usage

```
missingRateFromGenotypes(genotypes)
```

Arguments

genotypes	a vector of integers: see convertGenotypes

Details

The missing rate (β) is the proportion of missing data (i.e. NA) among the genotypes

Value

the missing rate

Examples

```
data(srdta)
beta <- missingRateFromGenotypes( fetchGenotypes(srdta, 1) )
```

parameters *make a set of parameters for GWASBinTests...*

Description

make a set of parameters for GWASBinTests

Usage

```
parameters(types, nb_permutations=1000, verbosity=0, seed=as.integer(Sys.time())
    max_error_rate=0.05, use_affymetrix_model=FALSE, min_pvalue=-1,
    confidence=0.999, max_relative_error=0, regularizeEstimators=FALSE,
    excludeX=FALSE, excludeMalesonXChr=FALSE, threads=0)
```

Arguments

`types`	the list of pvalue types to compute. Currently the possible values are "univariate", "divariate", "allelic", "genotypic" (default=all)
`nb_permutations`	the (maximum) number of permutations to use to compute the pvalues.
`verbosity`	the amount of verbosity: 0=none, 1=verbose
`seed`	the seed for the NNBC internal Random Number Generator. Useful to reproduce results.
`max_error_rate`	The maximum genotyping error rate: α For a given marker, the **error rate** is the probability of having an incorrect (but not missing) observed genotype, i.e. $P(\mathcal{O} \neq \mathcal{G} \mid \mathcal{O} \neq \emptyset)$ Hence the maximum error rate α is a higher bound of the error rates of all markers, i.e. $$\forall j, P(\mathcal{O}^j \neq \mathcal{G}^j \mid \mathcal{O}^j \neq \emptyset) \leq \alpha$$ This **maximum error rate**, sometimes called simply **error rate** is estimated during external comparison of genotyping technologies. We often use 0.05 as a default value.#'
`use_affymetrix_model`	Use the *Affymetrix* genotyping error model.
`min_pvalue`	The minimum "interesting" pvalue. Pvalues above this threshold (with confidence `confidence`) may be computed with less permutations.
`confidence`	value that controls the threshold on probability that a given pvalue computed with a given number of permutations is above the `min_pvalue` threshold, and so do not need to be computed with more permutations
`max_relative_error`	stop the permutations as soon as the relative error on the palues is below that threshold at `confidence` level. More info in the *Details* section below.
`regularizeEstimators`	Add a regularization constant to contingency tables.
`excludeX`	If set, X chromosome is excluded from the study.
`excludeMalesonXChr`	If set, male patients are excluded for analysis of bins on the X chromosome.
`threads`	the number of threads to use (if **OPENMP** is supported) to speed up the computation (defaults to the number of cpus/cores detected on the system).

Details

It may be used to set some parameter values, and use the defaults values for the others. Moreover some checks are performed and some internal settings computed afterwards. So one should never change a set of parameters after its construction by this function.

parameters for heuristics:

There are two heuristics implemented, that share the parameter `confidence` :

the min_pvalue heuristic The goal of this strategy is to avoid speeding permutations on getting a good accuracy on p-values that are of no interest. Let's say for instance we are computing p-values for 10^6 SNPs, we for sure are not interested in p-values > 0.1 . The estimated p-value follows a binomial distribution, so that we can compute its lower bound for a given probability, controlled by the parameter `confidence`. So if the lower bound of the p-value is greater than `min_pvalue` we stop the computation, and report the current estimation of the p-value along with the actual number of permutations used to compute it.

the "max_relative_error" heuristic Basically the principle of this heuristic is that the lower the p-value, the higher the number of permutations it needs to get a good accuracy, and most often we are greatly interested in those small p-values. What precision do we need ? We could set a maximum value on the radius of the confidence interval of the estimated p-value, e.g. 10^6. But what if the real p-value is 0.01 ? Then this amount of precision is a waste. And what if the real p-value is 10^-7 ? Then knowing that its confidence interval radius is < 10^-6 is of not great use.
So instead the idea is to control the relative error on the approximated p-value Given the parameters **max_relative_error** and **confidence**, the computation will stop when the relative error at a thegiven confidence level is lower than **max_relative_error**

Value

a named list with all parameters defined to their default values

Examples

```
params <- parameters(
seed = 0,
verbosity = 0,
nb_permutations = 100)
```

`processFiles` *run the c++ analysis engine on data files...*

Description

run the c++ analysis engine on data files

Usage

```
processFiles(basename="", geno_file="", pheno_file="", bins_file="",
    params=parameters(), covariables)
```

Arguments

`basename`	the basename of the files. ".gag", ".gap" and ".bins" suffixes will respectively be appended to the basename to form the data file names. Purely for convenience.
`geno_file`	the GenABEL genotypes file. See `load.gwaa.data` for a description of the format.
`pheno_file`	the GenABEL phenotypes data file, also refer to `load.gwaa.data`.
`bins_file`	the bins definition file. See `readBins` for more information about the format.
`params`	the NNBC parameters, see `parameters`
`covariables`	an optional dataframe of covariables. If no covariables are given, and that the dataset stored in the files contain a phenotypic column "gws" it will be used by default as covariable

Details

This function does not load the data into R, it just passes the parameters to the C++ code. It means that you do not need to load any data in R, everything happens in the NNBC C++ engine.

Value

a data frame with the following columns:

`bin`	the bin index
`chr`	the chr code (as integer, e.g. 23 for chromosome X)
`start`	the start position of the bin
`end`	the end position of the bin
`nb_snps`	the number of SNPs in this bin
`nb_permutations`	the actual number of permutations used to compute the pvalues of this bin
`score_type`	the score for the given **type**. For likelihoods it is not the ratio but the likelihood under the alternative hypothesis
`pv_type`	the pvalue for the given **type**
`fdr_type`	the FDR for the given **type**

See Also

`processTable`, `processGws`

Examples

```
## Not run:
data_path <- system.file("extdata", package = "GWASBinTests")
basename <- paste(data_path, "/ms1", sep="")
res <- processFiles(basename)

## End(Not run)
```

`processGws` *run the c++ analysis engine on a GenABEL gwaa dataset...*

Description

run the c++ analysis engine on a GenABEL gwaa dataset

Usage

```
processGws(gws, bins, params=parameters(), phenotypes=phdata(gws)$pop, covariabl
```

Arguments

`gws`	a GWSified dataset of class `gwaa.data-class`
`bins`	a data frame, as returned by `readBins`. If no bins are given the function will use a bin for each SNP to simulate a marker by marker scan (see `makeBinsForSnps`
`params`	the GWASBinTests parameters, see `parameters`
`phenotypes`	an optional vector of phenotypes. It will be converted to integer and must not contain a value higher than 255
`covariables`	an optional dataframe of covariables. If no covariables are given, and that the dataset stored in the files contain a phenotypic column **gws** it will be used by default as covariable

Details

The GenABEL dataset needs to be *GWSified* in order to be processed, see `asGws`

Value

a data frame (cf `processFiles`)

Examples

```
data("ms1")
res <- processGws(ms1_gws, ms1_bins,
parameters(nb_permutations=1000000, max_relative_error=0.1)
)
```

---

processTable *run the c++ analysis engine on a univariate or divariate table...*

Description

run the c++ analysis engine on a univariate or divariate table

Usage

```
processTable(table, params=parameters())
```

Arguments

`table`	a matrix of dimension **nb_phenotype***4 or **nb_phenotype***4*4 where table[i,j] (resp table[i,j,k] is the number of samples for phenotype i and genotype j (resp phenotype i and genotypes (j,k). The genotypes values are: 1. homozygous1 2. heterozygous 3. homozygous2 4. missing
`params`	the GWASBinTests parameters, see `parameters`

Details

Useful to study the different types of pvalues on some example tables.

Value

a list :

`nb_permutations`	the actual number of permutations used to compute the pvalues of the table
`pvalues`	a named vector of the pvalues computed for this table

See Also

`processFiles`, `processGws`

Examples

```
table2x4 <- matrix(c(100, 0, 20, 1,  80, 10, 10, 0), nrow=2,ncol=4, byrow=TRUE)
res <- processTable(table2x4)

table2x4x4 <- c(277,250,5,2,0,0,0,1,
107,106,3,3,0,0,0,0,
8, 12, 1, 1,0,0,0,0,
1,1,0,0,0,0,0,0)
dim(table2x4x4) <- c(2,4,4)
res <- processTable(table2x4x4)
```

---

readBins *Read a file containing defintions of genomic bins...*

Description

Read a file containing defintions of genomic bins

Usage

```
readBins(filename)
```

Arguments

filename	The path to the bins file.

Details

read a bins text file (TAB separated) into a dataframe, and sort the dataframe by genomic position.

Bins are just genomic intervals, i.e. fully qualified by a chromosome and an interval [start-end'].

Bins can of course be restricted to a single location by defining start=end=position. You could do this to ensure that some of the markers are alone in a bin.

FORMAT: This file must be TAB separated, with a header line, and contain the following fields:

- `chr_name` name of chromosome (1-22, X, Y, ...);
- `start` start position of bin
- `end` end position of bin.

Value

The sorted bins dataframe, with an additional column **bin_index**, useful to identify the bin

`readGws` *Read a genome wide association study dataset (genotypes + phenotypes)...*

Description

Read a genome wide association study dataset (genotypes + phenotypes)

Usage

```
readGws(basename, verbose=FALSE)
```

Arguments

`basename`	Full path to files without extensions. The function adds successively `.gag` and `.gap` to the path in order to have the full paths to files.
`verbose`	if TRUE, suppress the output of the GenABEL `load.gwaa.data` function.

Details

Reads a genome wide association study dataset in the internal GenABEL format. It is just a proxy for the `load.gwaa.data` function of `GenABEL`.

The dataset should be stored in two files named `basename`**.gag** and `basename`**.gap** for the genotypes and the phenotypes.

Value

A genome wide association dataset as a `gwaa.data-class` object.

`readPlinkTransposedData` *Read a genome wide association study dataset in PLINK transposed-ped format...*

Description

Read a genome wide association study dataset in PLINK transposed-ped format

Usage

```
readPlinkTransposedData(basename, verbose=FALSE)
```

Arguments

`basename`	Full path to files without extensions. The function adds successively `.tped` and `.tfam` to the path in order to have the full paths to files.
`verbose`	if TRUE, suppress the output of the GenABEL `load.gwaa.data` function.

Details

This function is a quite easy way to read and convert a PLINK dataset. It still requires that you convert the PLINK dataset to transposed-ped format (.tped and .tfam).

If you have the PLINK software installed, you just have to use the `--recode` and `--transpose` options. For example, if you have a binary PLINK dataset named foo (foo.bed, foo.bim, foo.fam), you can convert it into the bar transposed dataset:

```
plink --recode --transpose --bfile foo --out bar
```

The dataset should be stored in two files named `basename`**.gag** and `basename`**.gap** for the genotypes and the phenotypes.

Value

A genome wide association dataset as a `gwaa.data-class` object.

---

saveGws *Save a genome wide association study dataset (genotypes + phenotypes)...*

Description

Save a genome wide association study dataset (genotypes + phenotypes)

Usage

```
saveGws(gws, basename, verbose=FALSE)
```

Arguments

`gws`	A genome wide association dataset as a `gwaa.data-class` object.
`basename`	Full path to files without extensions. The function adds successively `.gag` and `.gap` to the path in order to have the full paths to files.
`verbose`	if TRUE, suppress the output of the GenABEL `save.gwaa.data` function.

Details

Save a genome wide association study dataset in the internal GenABEL format. It is just a proxy for the `save.gwaa.data` function of `GenABEL`.

The dataset will be stored in two files named `basename`**.gag** and `basename`**.gap** for the genotypes and the phenotypes.

Value

No value returned

`simpleAllelicSumScores` *compute the sum of allelic pearson score on a dataset...*

Description

compute the sum of allelic pearson score on a dataset

Usage

```
simpleAllelicSumScores(genotypes_list, phenotypes, covariables=data.frame(), reg
```

Arguments

`genotypes_list`	a list of vector of integers: see `fetchGenotypesAsList`
`phenotypes`	a vector of phenotypes of same length as the elements of genotypes_list
`covariables`	a data frame of cvariables: $(\mathcal{V})_m$
`regCoeff`	regularization rate: will be used to compute the constant to add in each cell of the contingency table to regularize estimators. This constant will be = rate*total/nb_of_cells. See **concept#regularization.of.table.estimates**.

Details

cf concept#allelic.sum.score This is a reference implementation, very naive on purpose, optimized for clarity.

Value

the score for the set of markers

`simpleAllelicSumScoresPvalue` *compute the sum of allelic pearson score pvalue on a dataset using permutations...*

Description

compute the sum of allelic pearson score pvalue on a dataset using permutations

Usage

```
simpleAllelicSumScoresPvalue(nb_permutations, genotypes_list, phenotypes, covari
    regCoeff=0)
```

Arguments

nb_permutations	the number of permutations to do
genotypes_list	a list of vector of integers: see fetchGenotypesAsList
phenotypes	a vector of phenotypes of same length as the elements of genotypes_list
covariables	a data frame of cvariables: $(\mathcal{V})_m$
regCoeff	regularization rate: will be used to compute the constant to add in each cell of the contingency table to regularize estimators. This constant will be = rate*total/nb_of_cells. See **concept#regularization.of.table.estimates**.

Details

See simpleAllelicSumScores This is a reference implementation, very naive on purpose, optimized for clarity.

Value

a list (pvalue=, observed_score=, nb_permutations=)

---

simpleDivariateLogLikelihood

compute the divariate log likelihood of a dataset...

Description

compute the divariate log likelihood of a dataset

Usage

```
simpleDivariateLogLikelihood(genotypes_list, variables, error_models, regCoeff=0
```

Arguments

genotypes_list	a list of vector of integers: see fetchGenotypes
variables	a data frame of variables, possibly empty
error_models	list of genotyping error model for each marker, see concept#genotype.error.model, simpleGenotypeErrorModel if it is not a list, it must be a matrix and this error model will be used for all markers in the dataset
regCoeff	see simpleDivariateLogLikelihoodForPair

Details

This is a reference implementation, very naive on purpose, optimized for clarity.

If there is only one SNP or less, it stops with an error message!

Value

the log-likelihood for the set of markers

simpleDivariateLogLikelihoodForPair

compute the divariate log likelihood for a given pair of markers...

Description

compute the divariate log likelihood for a given pair of markers

Usage

```
simpleDivariateLogLikelihoodForPair(g1, e1, g2, e2, variables, regCoeff=0)
```

Arguments

g1	the genotypes of the first marker as a vector of integers: see convertGenotypes
e1	the genotyping error model for the first marker, see concept#genotype.error.model, simpleGenotypeErrorModel
g2	the genotypes of the second marker
e2	the genotyping error model for the second marker
variables	a data frame of variables, possibly empty
regCoeff	regularization rate: will be used to compute the constant to add in each cell of the contingency table to regularize estimators. This constant will be = rate*total/nb_of_cells. See **concept#regularization.of.table.estimates**.

Details

This is the a very simple implementation, self-contained and easy to check but not performant. It is used as illustration or to check other implementations.

Value

the log likelihood numeric value. If there is no genotype for a category, the likelihood will not be defined and so it will return NaN

simpleDivariatePvalue

compute the divariate likelihood Ratio pvalue for a dataset/bin using permutations...

Description

compute the divariate likelihood Ratio pvalue for a dataset/bin using permutations

Usage

```
simpleDivariatePvalue(nb_permutations, genotypes_list, phenotypes, error_models,
    covariables=data.frame(), regCoeff=0)
```

Arguments

nb_permutations	the number of permutations to do
genotypes_list	a list of vector of integers: see fetchGenotypes
phenotypes	a vector of phenotypes of same length as the elements of genotypes_list
error_models	list of genotyping error model for each marker, see concept#genotype.error.model, simpleGenotypeErrorModel if it is not a list, it must be a matrix and this error model will be used for all markers in the dataset
covariables	a data frame of cvariables: $(\mathcal{V})_m$
regCoeff	see simpleDivariateLogLikelihoodForPair

Details

See simpleDivariateLogLikelihood This is a reference implementation, very naive on purpose, optimized for clarity.

If there is only one SNP or less, it stops with an error message!

Value

a list (pvalue=, observed_score=, nb_permutations=)

---

simpleGenotypeErrorModel

compute the simple genotype error model for a marker (SNP)...

Description

compute the *simple* genotype error model for a marker (SNP)

Usage

```
simpleGenotypeErrorModel(alpha, beta)
```

Arguments

alpha	the error_rate α. See **parameters**
beta	the missing value rate β

Details

See concept#genotype.error.model.

This model is much simpler than the affymetrix one, in that in does not assume that there is a bias on the heterozygous genotypes.

This will generate the following genotype error model:

$$P(\mathcal{O}^j \backslash \mathcal{G}^j) = \left(\begin{array}{c|c|c|c} \mathcal{O}^j \backslash \mathcal{G}^j & aa & Aa & AA \\ \hline aa & (1-\beta)*(1-\alpha) & (1-\beta)*\alpha/2 & (1-\beta)*\alpha/2 \\ \hline Aa & (1-\beta)*\alpha/2 & (1-\beta)*(1-\alpha) & (1-\beta)*\alpha/2 \\ \hline AA & 1-\beta)*\alpha/2 & 1-\beta)*\alpha/2 & (1-\beta)*(1-\alpha) \\ \hline \emptyset & \beta & \beta & \beta \end{array} \right)$$

Some comments:

1. $(1-\beta)$ is the probability for an observed genotype not to be missing. That is why this term is included in all non-missing observed genotype rows.
2. $(1-\alpha)$ is the probability of not doing an error
3. the probability of not doing any mistake (the diagonal) for a given genotype is $(1-\beta)*(1-\alpha)$, which simply means that this is not missing AND there is no error.
4. the error is equally split among the two other genotypes

Value

return error model a double

Examples

```
m <- simpleGenotypeErrorModel(0.05, 0.01)

# error model for first SNP of srdta
# by computing missing rate on genotypes
data(srdta)
beta <- missingRateFromGenotypes( fetchGenotypes(srdta,1) )
m <- simpleGenotypeErrorModel(0.05, 0.01)

prob <- m["aa","Aa"]

# error model for no errors !
no_errors <- simpleGenotypeErrorModel(0,0)
```

---

simpleGenotypicSumScores

compute the sum of genotypic pearson score on a dataset...

Description

compute the sum of genotypic pearson score on a dataset

Usage

```
simpleGenotypicSumScores(genotypes_list, phenotypes, covariables=data.frame(), r
```

Arguments

genotypes_list	a list of vector of integers: see fetchGenotypesAsList
phenotypes	a vector of phenotypes of same length as the elements of genotypes_list
covariables	a data frame of cvariables: $(\mathcal{V})_m$
regCoeff	regularization rate: will be used to compute the constant to add in each cell of the contingency table to regularize estimators. This constant will be = rate*total/nb_of_cells. See **concept#regularization.of.table.estimates**.

Details

cf concept#genotypic.sum.score This is a reference implementation, very naive on purpose, optimized for clarity.

Value

the score for the set of markers

---

simpleGenotypicSumScoresPvalue

compute the sum of genotypic pearson score pvalue on a dataset using permutations...

Description

compute the sum of genotypic pearson score pvalue on a dataset using permutations

Usage

```
simpleGenotypicSumScoresPvalue(nb_permutations, genotypes_list, phenotypes, cova
    regCoeff=0)
```

Arguments

nb_permutations	the number of permutations to do
genotypes_list	a list of vector of integers: see fetchGenotypesAsList
phenotypes	a vector of phenotypes of same length as the elements of genotypes_list
covariables	a data frame of cvariables: $(\mathcal{V})_m$
regCoeff	regularization rate: will be used to compute the constant to add in each cell of the contingency table to regularize estimators. This constant will be = rate*total/nb_of_cells. See **concept#regularization.of.table.estimates**.

Details

See concept#genotypic.sum.score and simpleGenotypicSumScores This is a reference implementation, very naive on purpose, optimized for clarity.

Value

a list (pvalue=, observed_score=, nb_permutations=)

`simpleUnivariateLogLikelihood`
compute the univariate log likelihood of a dataset...

Description

compute the univariate log likelihood of a dataset

Usage

```
simpleUnivariateLogLikelihood(genotypes_list, variables, error_models, regCoeff=
```

Arguments

`genotypes_list`	a list of vector of integers: see `fetchGenotypes`
`variables`	a data frame of variables, possibly empty
`error_models`	list of genotyping error model for each marker, see concept#genotype.error.model, `simpleGenotypeErrorModel` if it is not a list, it must be a matrix and this error model will be used for all markers in the dataset
`regCoeff`	see `simpleUnivariateLogLikelihoodForMarker`

Details

This is a reference implementation, very naive on purpose, optimized for clarity.

Value

the log-likelihood for the set of markers

`simpleUnivariateLogLikelihoodForMarker`
compute the univariate log likelihood for a given marker...

Description

compute the univariate log likelihood for a given marker

Usage

```
simpleUnivariateLogLikelihoodForMarker(genotypes, variables, error_model, regCoe
```

Arguments

`genotypes`	a vector of integers: see `convertGenotypes`
`variables`	a data frame of variables, possibly empty
`error_model`	a genotyping error model, see concept#genotype.error.model, `simpleGenotypeErrorModel`
`regCoeff`	regularization rate: will be used to compute the constant to add in each cell of the contingency table to regularize estimators. This constant will be = rate*total/nb_of_cells. See **concept#regularization.of.table.estimates**.

Details

This is the a very simple implementation, self-contained and easy to check but not performant. It is used as illustration or to check other implementations.

Value

the log likelihood numeric value. If there is no genotype for a category, the likelihood will not be defined and so it will return NaN

`simpleUnivariateLogLikelihoodRatio` *compute the univariate log likelihood Ratio for a dataset/bin...*

Description

compute the univariate log likelihood Ratio for a dataset/bin

Usage

```
simpleUnivariateLogLikelihoodRatio(genotypes_list, phenotypes, error_models, cov
    regCoeff=0)
```

Arguments

`genotypes_list`	a list of vector of integers: see `fetchGenotypes`
`phenotypes`	a vector of phenotypes of same length as the elements of genotypes_list
`error_models`	list of genotyping error model for each marker, see concept#genotype.error.model, `simpleGenotypeErrorModel` if it is not a list, it must be a matrix and this error model will be used for all markers in the dataset
`covariables`	a data frame of cvariables: $(\mathcal{V})_m$
`regCoeff`	see `simpleUnivariateLogLikelihoodForMarker`

Value

the log-likelihood ratio

`simpleUnivariatePvalue` *compute the univariate likelihood Ratio pvalue for a dataset/bin using permutations...*

Description

compute the univariate likelihood Ratio pvalue for a dataset/bin using permutations

Usage

```
simpleUnivariatePvalue(nb_permutations, genotypes_list, phenotypes, error_models
    covariables=data.frame(), regCoeff=0)
```

Arguments

`nb_permutations`	the number of permutations to do
`genotypes_list`	a list of vector of integers: see `fetchGenotypes`
`phenotypes`	a vector of phenotypes of same length as the elements of genotypes_list
`error_models`	list of genotyping error model for each marker, see concept#genotype.error.model, `simpleGenotypeErrorModel` if it is not a list, it must be a matrix and this error model will be used for all markers in the dataset
`covariables`	a data frame of cvariables: $(\mathcal{V})_m$
`regCoeff`	see `simpleUnivariateLogLikelihoodForMarker`

Details

See `simpleUnivariateLogLikelihoodRatio` This is a reference implementation, very naive on purpose, optimized for clarity.

Value

a list (pvalue=, observed_score=, nb_permutations=)

Index