

VariantFiltering: filtering of coding and non-coding genetic variants

Dei M. Elurbe^{1,2} and Robert Castelo^{3,4}

September 1, 2017

¹ CIBER de Enfermedades Raras (CIBERER), Barcelona, Spain

² Present address: CMBI, Radboud University Medical Centre, Nijmegen, The Netherlands.

² Department of Experimental and Health Sciences, Universitat Pompeu Fabra, Barcelona, Spain.

³ Research Program on Biomedical Informatics (GRIB), Hospital del Mar Medical Research Institute, Barcelona, Spain.

1 Overview

The aim of this software package is to facilitate the filtering and annotation of coding and non-coding genetic variants from a group of unrelated individuals, or a family of related ones among which at least one of them is affected by a genetic disorder. When working with related individuals, [VariantFiltering](#) can search for variants from the affected individuals that segregate according to a particular inheritance model acting on autosomes (dominant, recessive homozygous or recessive heterozygous -also known as compound heterozygous) or on allosomes (X-linked), or that occur *de novo*. When working with unrelated individuals, no mode of inheritance is used for filtering but it can be used to search for variants shared among individuals affected by a common genetic disorder.

The main input is a multisample Variant Call Format (VCF) file which is parsed using the R/Bioconductor infrastructure, and particularly the functionality from the [VariantAnnotation](#) package for that purpose. This infrastructure also allows [VariantFiltering](#) to stream over large VCF files to reduce the memory footprint and to annotate the input variants with diverse functional information.

A core set of functional data are annotated by [VariantFiltering](#) but this set can be modified and extended using Bioconductor annotation packages such as [MafDb.1Kgenomes.phase3.hs37d5](#), which stores and exposes to the user minor allele frequency (MAF) values frozen from the latest release of the 1000 Genomes project.

The package contains a toy data set to illustrate how it works through this vignette, and it consists of a multisample VCF file with variants from chromosomes 20, 21, 22 and allosomes X, Y from a trio of CEU individuals of the 1000 Genomes project. To further reduce the execution time of this vignette, only the code for the first analysis is actually evaluated and its results reported.

2 Setting up the analysis

To start using [VariantFiltering](#) the user should consider installing the packages listed in the `Suggests:` field from its DESCRIPTION file. After loading [VariantFiltering](#) the first step is to build a parameter object, of class `VariantFilteringParam` which requires at least a character string with the input VCF filename, as follows:

```
> library(VariantFiltering)
> CEUvcf <- file.path(system.file("extdata", package="VariantFiltering"), "CEUtrio.vcf.bgz")
> vfpar <- VariantFilteringParam(CEUvcf)
> class(vfpar)
```

```
[1] "VariantFilteringParam"
attr(,"package")
[1] "VariantFiltering"
```

```
> vfpar
```

VariantFiltering parameter object

```
VCF file(s): CEUtrio.vcf.bgz
Genome version(s): hg19(NCBI)
Number of individuals: 3 (NA12878, NA12891, NA12892)
Genome-centric annotation package: BSgenome.Hsapiens.1000genomes.hs37d5 (1000genomes hs37d5 hs37d5)
Variant-centric annotation package: SNPlocs.Hsapiens.dbSNP144.GRCh37 (dbSNP dbSNP Human BUILD 144)
Transcript-centric annotation package: TxDb.Hsapiens.UCSC.hg19.knownGene
Gene-centric annotation package: org.Hs.eg.db
Radical/Conservative AA changes: AA_chemical_properties_HanadaGojoboriLi2006.tsv
Codon usage table: humanCodonUsage.txt
Regions to annotate: CodingVariants, IntronVariants, FiveSpliceSiteVariants, ThreeSpliceSiteVariants, Pro
Other annotation pkg/obj: MafDb.1Kgenomes.phase1.hs37d5,
                        PolyPhen.Hsapiens.dbSNP131,
                        SIFT.Hsapiens.dbSNP137,
                        phastCons100way.UCSC.hg19,
                        humanGenesPhylostrata

All transcripts: FALSE
```

The display of the *VariantFilteringParam* object indicates a number of default values which can be overridden when calling the construction function. To quickly see all the available arguments and their default values we should type:

```
> args(VariantFilteringParam)
```

```
function (vcfFileNames, pedFilename = NA_character_, bsgenome = "BSgenome.Hsapiens.1000genomes.hs37d5",
  orgdb = "org.Hs.eg.db", txdb = "TxDb.Hsapiens.UCSC.hg19.knownGene",
  snpdb = "SNPlocs.Hsapiens.dbSNP144.GRCh37", weightMatricesFileNames = NA,
  weightMatricesLocations = rep(list(variantLocations()), length(weightMatricesFileNames)),
  weightMatricesStrictLocations = rep(list(FALSE), length(weightMatricesFileNames)),
  radicalAAchangeFilename = file.path(system.file("extdata",
    package = "VariantFiltering"), "AA_chemical_properties_HanadaGojoboriLi2006.tsv"),
  codonusageFilename = file.path(system.file("extdata", package = "VariantFiltering"),
    "humanCodonUsage.txt"), geneticCode = getGeneticCode("SGCO"),
  allTranscripts = FALSE, regionAnnotations = list(CodingVariants(),
    IntronVariants(), FiveSpliceSiteVariants(), ThreeSpliceSiteVariants(),
    PromoterVariants(), FiveUTRVariants(), ThreeUTRVariants()),
  otherAnnotations = c("MafDb.1Kgenomes.phase1.hs37d5", "PolyPhen.Hsapiens.dbSNP131",
    "SIFT.Hsapiens.dbSNP137", "phastCons100way.UCSC.hg19",
    "humanGenesPhylostrata"), geneKeytype = NA_character_,
  yieldSize = NA_integer_)
NULL
```

The manual page of *VariantFilteringParam* contains more information about these arguments and their default values.

3 Annotating variants

After setting up the parameters object, the next step is to annotate variants. This can be done using upfront an inheritance model that will substantially filter and reduce the number of variants and annotations or, as we illustrate here below, calling the function *unrelatedIndividuals()* that just annotates the variants without filtering out any of them:

```
> uind <- unrelatedIndividuals(vfpar)
> class(uind)
```

```
[1] "VariantFilteringResults"
attr(,"package")
[1] "VariantFiltering"
```

```
> uind
```

VariantFiltering results object

```
Genome version(s): "hg19"(NCBI) "hg19"(Ensembl)
Number of individuals: 3 (NA12878, NA12891, NA12892)
Variants segregate according to a(n) unrelated individuals inheritance model
Quality filters
INDELQual LowQual SNPQual
TRUE TRUE TRUE
Functional annotation filters
dbSNP OMIM variantType aaChangeType S0terms
FALSE FALSE FALSE FALSE FALSE
Populations used for MAF filtering: AFKGp1, AFR_AFKGp1, AMR_AFKGp1, ASN_AFKGp1, EUR_AFKGp1
Include MAF NA values: yes
Maximum MAF: 1.00
```

The resulting object belongs to the *VariantFilteringResults* class of objects, defined within [VariantFiltering](#), whose purpose is to ease the task of filtering and prioritizing the annotated variants. The display of the object just tells us the genome version of the input VCF file, the number of individuals, the inheritance model and what variant filters are activated.

To get a summary of the number of variants annotated to a particular feature we should use the function `summary()`:

```
> summary(uind)
```

	SOID	Description	Nr. Variants	% Variants
1	S0:0002012	start_lost	1	0.27
2	S0:0001631	upstream_gene_variant	52	14.13
3	S0:0001624	3_prime_UTR_variant	16	4.35
4	S0:0001623	5_prime_UTR_variant	13	3.53
5	S0:0001589	frameshift_variant	1	0.27
6	S0:0001587	stop_gained	2	0.54
7	S0:0001583	missense_variant	16	4.35
8	S0:0001575	splice_donor_variant	3	0.82
9	S0:0001574	splice_acceptor_variant	6	1.63
10	S0:0001627	intron_variant	309	83.97
11	S0:0001819	synonymous_variant	16	4.35

The default setting of the `summary()` function is to provide feature annotations in terms of Sequence Ontology (SO) terms. The reported number of variants refer to the number of different variants in the input VCF file annotated to the feature while the percentage of variants refers to the fraction of this number over the total number of different variants in the input VCF file.

We can also obtain a summary based on the Bioconductor feature annotations provided by the functions `locateVariants()` and `predictCoding()` from the [VariantAnnotation](#) package, as follows:

```
> summary(uind, method="bioc")
```

	BIOCID	Nr. Variants	% Variants
2	intron	309	83.97
3	fiveUTR	13	3.53
4	threeUTR	16	4.35
5	coding	35	9.51

7	promoter	52	14.13
8	fiveSpliceSite	3	0.82
9	threeSpliceSite	6	1.63
10	frameshift	1	0.27
11	nonsense	2	0.54
12	nonsynonymous	16	4.35
13	synonymous	16	4.35

Since SO terms are organized hierarchically, we can use this structure to aggregate feature annotations into more coarse-grained SO terms using the argument `method="SOfull"`:

```
> uindSO <- summary(uind, method="SOfull")
> uindSO
```

	SOID	Level	Description	Nr. Variants	% Variants
1	SO:0002012	9	start_lost	1	0.27
2	SO:0001582	8	initiator_codon_variant	1	0.27
3	SO:0001819	8	synonymous_variant	16	4.35
4	SO:0001631	5	upstream_gene_variant	52	14.13
5	SO:0001628	4	intergenic_variant	52	14.13
6	SO:0001624	8	3_prime_UTR_variant	16	4.35
7	SO:0001623	8	5_prime_UTR_variant	13	3.53
8	SO:0001622	7	UTR_variant	29	7.88
9	SO:0001589	9	frameshift_variant	1	0.27
10	SO:0001587	5	stop_gained	2	0.54
11	SO:0001906	4	feature_truncation	2	0.54
12	SO:0001583	11	missense_variant	16	4.35
13	SO:0001992	10	nonsynonymous_variant	18	4.89
14	SO:0001650	9	inframe_variant	18	4.89
15	SO:0001818	8	protein_altering_variant	19	5.16
16	SO:0001580	7	coding_sequence_variant	35	9.51
17	SO:0001968	6	coding_transcript_variant	55	14.95
18	SO:0001791	6	exon_variant	55	14.95
19	SO:0001575	8	splice_donor_variant	3	0.82
20	SO:0001574	8	splice_acceptor_variant	6	1.63
21	SO:0001629	7	splice_site_variant	9	2.45
22	SO:0001627	6	intron_variant	310	84.24
23	SO:0001568	6	splicing_variant	9	2.45
24	SO:0001576	5	transcript_variant	351	95.38
25	SO:0001564	4	gene_variant	351	95.38
26	SO:0001878	3	feature_variant	368	100.00
27	SO:0001537	2	structural_variant	368	100.00
28	SO:0001060	1	sequence_variant	368	100.00
29	SO:0002072	0	sequence_comparison	368	100.00

Here the `Level` column refers to the shortest-path distance to the most general SO term `sequence_variant` within the SO acyclic digraph. We can use this level value to interrogate the annotations on a specific granularity:

```
> uindSO[uindSO$Level == 6, ]
```

	SOID	Level	Description	Nr. Variants	% Variants
17	SO:0001968	6	coding_transcript_variant	55	14.95
18	SO:0001791	6	exon_variant	55	14.95
22	SO:0001627	6	intron_variant	310	84.24
23	SO:0001568	6	splicing_variant	9	2.45

Variants are stored internally in a `VRanges` object. We can retrieve the variants as a `VRanges` object with the function `allVariants()`:

```
> allVariants(uind)
```

```
VRangesList of length 3
names(3): NA12878 NA12891 NA12892
```

This function in fact returns a *VRangesList* object with one element per sample by default. We can change the grouping of variants with the argument `groupBy` specifying the annotation column we want to use to group variants. If the specified column does not exist, then it will return a single *VRanges* object with all annotated variants.

Using the following code we can obtain a graphical display of a variant, including the aligned reads and the running coverage, to have a visual representation of its support. For this purpose we need to have the BAM files used to perform the variant calling. In this case we are using toy BAM files stored along with the *VariantFiltering* package, which for practical reasons only include a tiny subset of the aligned reads.

```
> path2bams <- file.path(system.file("extdata", package="VariantFiltering"),
+                          paste0(samples(uind), ".subset.bam"))
> bv <- BamViews(bamPaths=path2bams,
+                bamSamples=DataFrame(row.names=samples(uind)))
> bamFiles(uind) <- bv
> bamFiles(uind)
```

```
BamViews dim: 0 ranges x 3 samples
names: NA12878 NA12891 NA12892
detail: use bamPaths(), bamSamples(), bamRanges(), ...
```

```
> plot(uind, what="rs6130959", sampleName="NA12892")
```

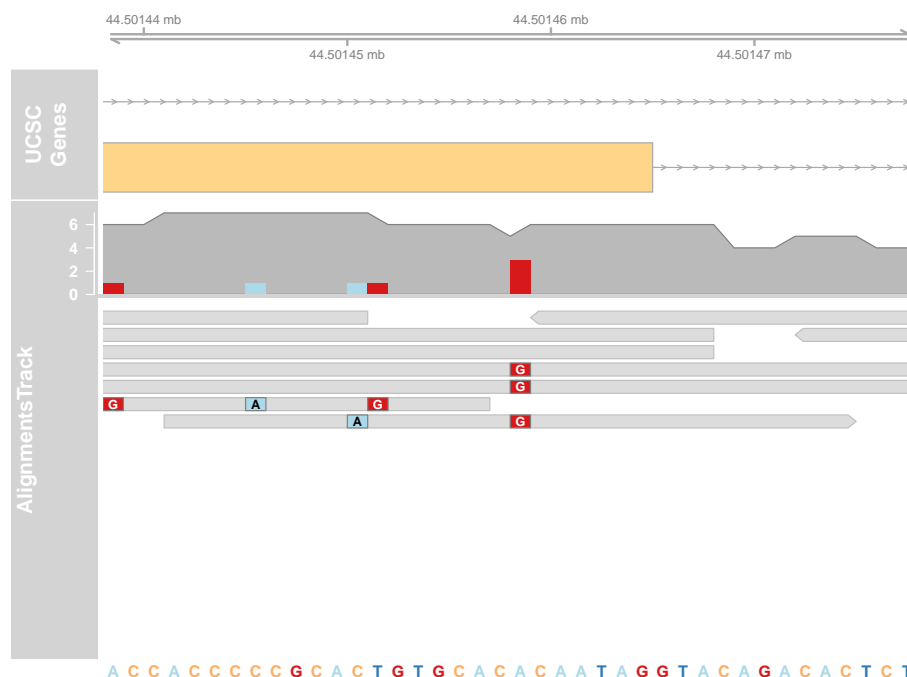


Figure 1: Browser-like display of a variant.

4 Filters and cutoffs

In the case of having run the `unrelatedIndividuals()` annotation function we can filter variants by restricting the samples involved in the analysis, as follows:

```
> samples(uind)
```

```
[1] "NA12878" "NA12891" "NA12892"
```

```
> samples(uind) <- c("NA12891", "NA12892")
```

```
> uind
```

VariantFiltering results object

Genome version(s): "hg19"(NCBI) "hg19"(Ensembl)

Number of individuals: 2 (NA12891, NA12892)

Variants segregate according to a(n) unrelated individuals inheritance model

Quality filters

```
INDELQual  LowQual  SNPQual
      TRUE      TRUE      TRUE
```

Functional annotation filters

```
dbSNP      OMIM  variantType aaChangeType      S0terms
  FALSE      FALSE      FALSE      FALSE      FALSE
```

Populations used for MAF filtering: AFKGp1, AFR_AFKGp1, AMR_AFKGp1, ASN_AFKGp1, EUR_AFKGp1

Include MAF NA values: yes

Maximum MAF: 1.00

```
> uindS02sam <- summary(uind, method="S0full")
```

```
> uindS02sam[uindS02sam$Level == 6, ]
```

	SOID	Level	Description	Nr. Variants	% Variants
17	S0:0001968	6	coding_transcript_variant	50	15.34
18	S0:0001791	6	exon_variant	50	15.34
22	S0:0001627	6	intron_variant	272	83.44
23	S0:0001568	6	splicing_variant	7	2.15

As we can see, restricting the samples for filtering variants results in fewer variants. We can set the original samples back with the function `resetSamples()`:

```
> uind <- resetSamples(uind)
```

```
> uind
```

VariantFiltering results object

Genome version(s): "hg19"(NCBI) "hg19"(Ensembl)

Number of individuals: 3 (NA12878, NA12891, NA12892)

Variants segregate according to a(n) unrelated individuals inheritance model

Quality filters

```
INDELQual  LowQual  SNPQual
      TRUE      TRUE      TRUE
```

Functional annotation filters

```
dbSNP      OMIM  variantType aaChangeType      S0terms
  FALSE      FALSE      FALSE      FALSE      FALSE
```

Populations used for MAF filtering: AFKGp1, AFR_AFKGp1, AMR_AFKGp1, ASN_AFKGp1, EUR_AFKGp1

Include MAF NA values: yes

Maximum MAF: 1.00

The rest of the filtering operations we can perform on a *VariantFilteringResults* objects are implemented through the

FilterRules class which implements a general mechanism for generating logical masks to filter vector-like objects; consult its manual page at the [IRanges](#) package for full technical details.

The [Variantfiltering](#) package provides a number of default filters, which can be extended by the user. To see which are these filters we just have to use the `filters()` function:

```
> filters(uind)
```

```
FilterRules of length 8
```

```
names(8): INDELQual LowQual SNPQual dbSNP OMIM variantType aaChangeType S0terms
```

Filters may be active or inactive. Only active filters will participate in the filtering process when we interrogate for variants. To know what filters are active we should use the `active()` function as follows:

```
> active(filters(uind))
```

INDELQual	LowQual	SNPQual	dbSNP	OMIM	variantType
TRUE	TRUE	TRUE	FALSE	FALSE	FALSE
aaChangeType	S0terms				
FALSE	FALSE				

By default, all filters are always inactive. To activate all of them, we can simply type:

```
> active(filters(uind)) <- TRUE
```

```
> active(filters(uind))
```

INDELQual	LowQual	SNPQual	dbSNP	OMIM	variantType
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
aaChangeType	S0terms				
TRUE	TRUE				

```
> summary(uind)
```

	S0ID	Description	Nr. Variants	% Variants
1	S0:0002012	start_lost	1	0.40
2	S0:0001631	upstream_gene_variant	34	13.49
3	S0:0001624	3_prime_UTR_variant	10	3.97
4	S0:0001623	5_prime_UTR_variant	7	2.78
5	S0:0001587	stop_gained	2	0.79
6	S0:0001583	missense_variant	13	5.16
7	S0:0001575	splice_donor_variant	2	0.79
8	S0:0001574	splice_acceptor_variant	3	1.19
9	S0:0001627	intron_variant	212	84.13
10	S0:0001819	synonymous_variant	12	4.76

To deactivate all of them back and selectively activate one of them, we should use the bracket `[]` notation, as follows:

```
> active(filters(uind)) <- FALSE
```

```
> active(filters(uind))["dbSNP"] <- TRUE
```

```
> summary(uind)
```

	S0ID	Description	Nr. Variants	% Variants
1	S0:0002012	start_lost	1	0.24
2	S0:0001631	upstream_gene_variant	55	13.16
3	S0:0001624	3_prime_UTR_variant	16	3.83
4	S0:0001623	5_prime_UTR_variant	11	2.63
5	S0:0001587	stop_gained	2	0.48
6	S0:0001583	missense_variant	18	4.31
7	S0:0001575	splice_donor_variant	3	0.72
8	S0:0001574	splice_acceptor_variant	6	1.44

```

9  SO:0001627      intron_variant      356      85.17
10 SO:0001819      synonymous_variant      16       3.83

```

The previous filter just selects variants with an annotated dbSNP identifier. However, other filters may require cutoff values to decide what variants pass the filter. To set those values we can use the function `cutoffs()`. For instance, in the case of the `S0terms` filter, we should use set the cutoff values to select variants annotated to specific SO terms. Here we select, for instance, those annotated in UTR regions:

```

> cutoffs(uind)$S0terms <- "UTR_variant"
> active(filters(uind))["S0terms"] <- TRUE
> summary(uind)

[1] S0ID      Description  Nr. Variants % Variants
<0 rows> (or 0-length row.names)

> summary(uind, method="S0full")

      S0ID Level      Description Nr. Variants % Variants
1  SO:0001624      8      3_prime_UTR_variant      16      59
2  SO:0001623      8      5_prime_UTR_variant      11      41
3  SO:0001622      7      UTR_variant      27     100
4  SO:0001968      6 coding_transcript_variant      27     100
5  SO:0001791      6      exon_variant      27     100
6  SO:0001576      5      transcript_variant      27     100
7  SO:0001564      4      gene_variant      27     100
8  SO:0001878      3      feature_variant      27     100
9  SO:0001537      2      structural_variant      27     100
10 SO:0001060      1      sequence_variant      27     100
11 SO:0002072      0      sequence_comparison      27     100

```

Note that the first call to `summary()` did not report any variant since there are no variants annotated to the SO term `UTR_variant`. However, when using the argument `method="S0full"`, all variants annotated to more specific SO terms in the hierarchy will be reported.

The methods `filters()` and `cutoffs()` can be employed to extend the filtering functionality. Here we show a simple example in which we add a filter to detect the loss of the codon that initiates translation. This constitutes already a feature annotated by [VariantFiltering](#) so that we can verify that it works:

```

> startLost <- function(x) {
+   mask <- start(allVariants(x, groupBy="nothing")$CDSLOC) == 1 &
+     as.character(allVariants(x, groupBy="nothing")$REFCODON) == "ATG" &
+     as.character(allVariants(x, groupBy="nothing")$VARCODON) != "ATG"
+   mask
+ }
> filters(uind)$startLost <- startLost
> active(filters(uind)) <- FALSE
> active(filters(uind))["startLost"] <- TRUE
> active(filters(uind))

      INDELQual      LowQual      SNPQual      dbSNP      OMIM      variantType
      FALSE      FALSE      FALSE      FALSE      FALSE      FALSE
aaChangeType      S0terms      startLost
      FALSE      FALSE      TRUE

> summary(uind)

      S0ID      Description Nr. Variants % Variants
1  SO:0002012      start_lost      1      100
2  SO:0001631 upstream_gene_variant      1      100
3  SO:0001623      5_prime_UTR_variant      1      100

```



```

4 S0:0001583      missense_variant      1      100
5 S0:0001575 splice_donor_variant      1      100
6 S0:0001627      intron_variant      1      100

```

As we can see, our filter works as expected and selects the only variant which was annotated with the SO term `start_lost`. Note that there is also an additional annotation indicating this variant belongs to an UTR region resulting from an alternative CDS.

Properly updating cutoff values may be problematic if we do not know how exactly are they employed by the corresponding filters. To facilitate setting the right cutoff values the help page of the *VariantFilteringResults* class contains a list of available accessor methods to update them. Here we illustrate the use of one of them, the one controlling minor allele frequency (MAF) values:

```

> active(filters(uind)) <- FALSE
> MAFmask <- MAFpop(uind)
> MAFmask

      AFKGp1 AFR_AFKGp1 AMR_AFKGp1 ASN_AFKGp1 EUR_AFKGp1
      TRUE      TRUE      TRUE      TRUE      TRUE

> MAFpop(uind) <- !MAFmask
> MAFpop(uind, "AFKGp1") <- TRUE
> MAFpop(uind)

      AFKGp1 AFR_AFKGp1 AMR_AFKGp1 ASN_AFKGp1 EUR_AFKGp1
      TRUE      FALSE      FALSE      FALSE      FALSE

> maxMAF(uind) <- 0.01
> summary(uind)

      SOID      Description Nr. Variants % Variants
1 S0:0001631 upstream_gene_variant      23      15.86
2 S0:0001624 3_prime_UTR_variant       3       2.07
3 S0:0001623 5_prime_UTR_variant       5       3.45
4 S0:0001587 stop_gained                1       0.69
5 S0:0001583 missense_variant          6       4.14
6 S0:0001575 splice_donor_variant       1       0.69
7 S0:0001574 splice_acceptor_variant     3       2.07
8 S0:0001627 intron_variant          125      86.21
9 S0:0001819 synonymous_variant         4       2.76

```

In this case we selected variants with $MAF < 0.01$ in the global population of the 1000 Genomes project. If we are interested in retrieving the actual set of filtered variants, we can do it using the function `filteredVariants()`:

```

> filteredVariants(uind)

VRangesList of length 3
names(3): NA12878 NA12891 NA12892

```

To further understand how to manipulate *Vranges* and *VRangesList* objects, please consult the package [VariantAnnotation](#).

5 Inheritance models

We can filter upfront variants that do not segregate according to a given inheritance model. In such a case, we also need to provide a PED file at the time we build the parameter object, as follows:

```
> CEUped <- file.path(system.file("extdata", package="VariantFiltering"),
+                       "CEUtrio.ped")
> param <- VariantFilteringParam(vcfFileNames=CEUvcf, pedFilename=CEUped)
```

Here we are using a PED file included in the [VariantFiltering](#) package and specifying information about the CEU trio employed in this vignette.

To use an inheritance model we need to replace the previous call to `unrelatedIndividuals()` by one specific to the inheritance model. The [VariantFiltering](#) package offers 5 possible ones:

- **Autosomal recessive inheritance analysis: Homozygous.**
Homozygous variants responsible for a recessive trait in the affected individuals can be identified calling the `autosomalRecessiveHomozygous()` function. This function selects homozygous variants that are present in all the affected individuals and occur in heterozygosity in the unaffected ones.
- **Autosomal recessive inheritance analysis: Heterozygous.**
To filter by this mode of inheritance, also known as compound heterozygous, we need two unaffected parents/ancestors and at least one affected descendant. Variants are filtered in five steps: 1. select heterozygous variants in one of the parents and homozygous in the other; 2. discard previously selected variants that are common between the two parents; 3. group variants by gene; 4. select those genes, and the variants that occur within them, which have two or more variants and there is at least one from each parent; 5. from the previously selected variants, discard those that do not occur in the affected descendants. This is implemented in the function `autosomalRecessiveHeterozygous()`.
- **Autosomal dominant inheritance analysis.**
The function `autosomalDominant()` identifies variants present in all the affected individual(s) discarding the ones that also occur in at least one of the unaffected subjects.
- **X-Linked inheritance analysis.**
The function `xLinked()` identifies variants that appear only in the X chromosome of the unaffected females as heterozygous, don't appear in the unaffected males analyzed and finally are present (as homozygous) in the affected male(s). This function is currently restricted to affected males, and therefore, it cannot search for X-linked segregating variants affecting daughters.
- **De Novo variants analysis**
The function `deNovo()` searches for *de novo* variants which are present in one descendant and present in both parents/ancestors. It is currently restricted to a trio of individuals.

6 Create a report from the filtered variants

The function `reportVariants()` allows us to easily create a report from the filtered variants into a CSV or a TSV file as follows:

```
> reportVariants(uind, type="csv", file="uind.csv")
```

The default value on the `type` argument (`"shiny"`) starts a shiny web app which allows one to interactively filter the variants, obtaining an updated *VariantFilteringResults* object and downloading the filtered variants and the corresponding full reproducible R code, if necessary.

7 Using the package with parallel execution

Functions in [VariantFiltering](#) to annotate and filter variants leverage the functionality of the Bioconductor package [BiocParallel](#) to perform in parallel some of the tasks and calculations and reduce the overall execution time. These functions have an argument called `BPPARAM` that allows the user to control how this parallelism is exploited. In particular the user must give as value to this argument the result from a call to the function `bpparam()`, which actually is its default behavior. Here below we modify that behavior to force a call being executed without parallelism. The interested reader should consult the help page of `bpparam()` and the vignette of the [BiocParallel](#) for further information.

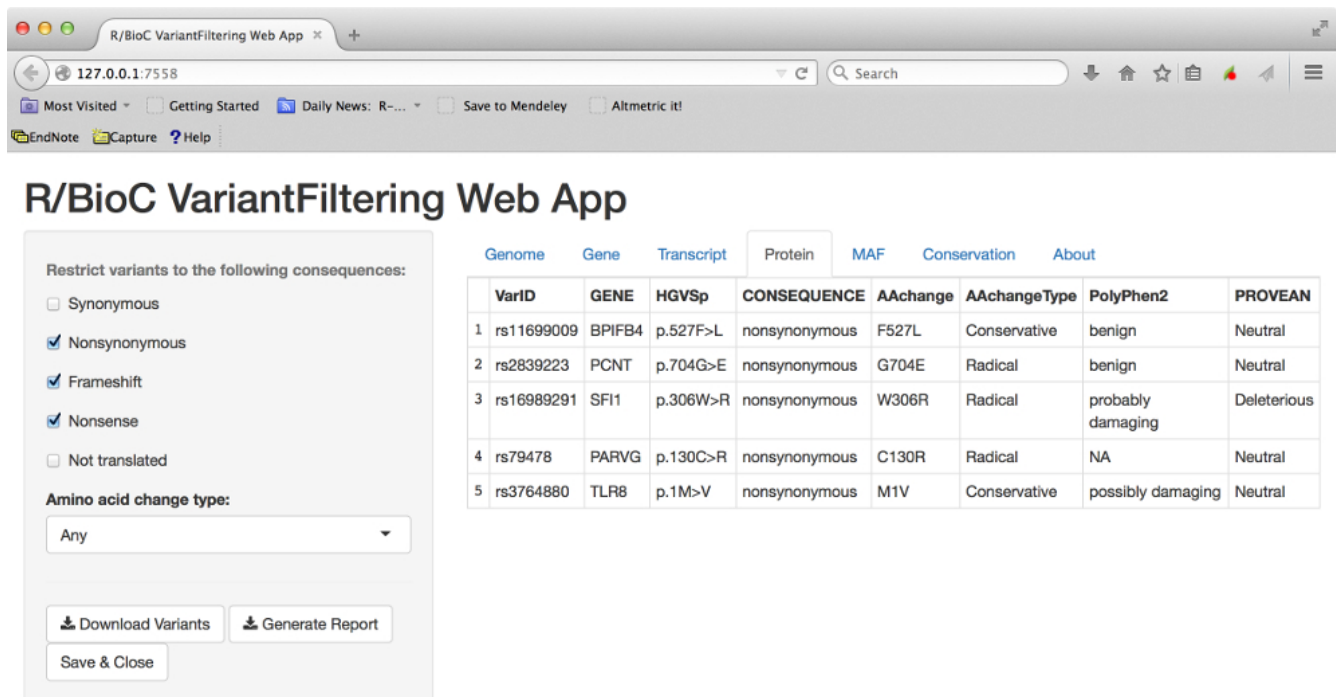


Figure 2: Snapshot of the shiny web app run from VariantFiltering with the function `reportVariants()`. Some of the parameters has been filled for illustrative purposes.

8 Session information

```
> toLatex(sessionInfo())
```

- R version 3.4.1 (2017-06-30), x86_64-apple-darwin15.6.0
- Locale: C/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
- Running under: OS X El Capitan 10.11.6
- Matrix products: default
- BLAS: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRblas.0.dylib
- LAPACK: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRlapack.dylib
- Base packages: base, datasets, grDevices, graphics, methods, parallel, stats, stats4, utils
- Other packages: AnnotationDbi 1.38.2, BSgenome 1.44.1, BSgenome.Hsapiens.1000genomes.hs37d5 0.99.1, Biobase 2.36.2, BiocGenerics 0.22.0, Biostings 2.44.2, DelayedArray 0.2.7, GenomInfoDb 1.12.2, GenomicFeatures 1.28.4, GenomicRanges 1.28.4, GenomicScores 1.0.2, IRanges 2.10.3, MafDb.1Kgenomes.phase1.hs37d5 3.5.0, PolyPhen.Hsapiens.dbSNP131 1.0.2, RSQLite 2.0, Rsamtools 1.28.0, S4Vectors 0.14.3, SIFT.Hsapiens.dbSNP137 1.0.0, SNPlocs.Hsapiens.dbSNP144.GRCh37 0.99.20, SummarizedExperiment 1.6.3, TxDb.Hsapiens.UCSC.hg19.knownGene 3.2.2, VariantAnnotation 1.22.3, VariantFiltering 1.12.2, XVector 0.16.0, matrixStats 0.52.2, org.Hs.eg.db 3.4.1, phastCons100way.UCSC.hg19 3.5.0, rtracklayer 1.36.4
- Loaded via a namespace (and not attached): AnnotationFilter 1.0.0, AnnotationHub 2.8.2, BiocInstaller 1.26.1, BiocParallel 1.10.1, BiocStyle 2.4.1, DBI 0.7, Formula 1.2-2, GenomInfoDbData 0.99.0, GenomicAlignments 1.12.2, Gviz 1.20.0, Hmisc 4.0-3, Matrix 1.2-11, ProtGenerics 1.8.0, R6 2.2.2, RBGL 1.52.0, RColorBrewer 1.1-2, RCurl 1.95-4.8, Rcpp 0.12.12, XML 3.98-1.9, acepack 1.4.1, backports 1.1.0, base64enc 0.1-3, biomaRt 2.32.1, biovizBase 1.24.0, bit 1.1-12, bit64 0.9-7, bitops 1.0-6, blob 1.1.0, checkmate 1.8.3, cluster 2.0.6, colorspace 1.3-2, compiler 3.4.1, curl 2.8.1, data.table 1.10.4, dichromat 2.0-0, digest 0.6.12, ensemblDb 2.0.4, evaluate 0.10.1, foreign 0.8-69, ggplot2 2.2.1, graph 1.54.0, grid 3.4.1, gridExtra 2.2.1, gtable 0.2.0, htmlTable 1.9, htmltools 0.3.6, htmlwidgets 0.9, httpuv 1.3.5, httr 1.3.1,

interactiveDisplayBase 1.14.0, knitr 1.17, lattice 0.20-35, latticeExtra 0.6-28, lazyeval 0.2.0, magrittr 1.5, memoise 1.1.0, mime 0.5, munsell 0.4.3, nnet 7.3-12, pkgconfig 2.0.1, plyr 1.8.4, rlang 0.1.2, rmarkdown 1.6, rpart 4.1-11, rprojroot 1.2, scales 0.5.0, shiny 1.0.5, splines 3.4.1, stringi 1.1.5, stringr 1.2.0, survival 2.41-3, tibble 1.3.4, tools 3.4.1, xtable 1.8-2, yaml 2.1.14, zlibbioc 1.22.0