

VariantFiltering: filtering of coding and non-coding genetic variants

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

May 25, 2015

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 or 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.

[VariantFiltering](#) exploits the R/Bioconductor infrastructure to read and stream over input files and to annotate the input variants with diverse functional information. A core set of functional annotations are generated by [VariantFiltering](#) but this set can be modified and extended using Bioconductor annotation packages such as [MafDb.ALL.wgs.phase3.release.v5a.20130502](#), which stores and exposes to the user minimum allele frequency (MAF) values frozen from the latest release of the 1000 Genomes project.

The main input is a multisample Variant Call Format (VCF) file which is parsed using the functionality from the [VariantAnnotation](#) package for that purpose. This functionality allows also [VariantFiltering](#) to stream over large VCF files to reduce the memory footprint.

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(vcfFileNames=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.UCSC.hg19 (UCSC hg19 Genome Reference Consortium GRCh37)
Variant-centric annotation package: SNPlocs.Hsapiens.dbSNP.20120608 (dbSNP Build 137)
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
Other annotation pkg/obj: MafDb.ESP6500SI.V2.SSA137,
                        MafDb.ALL.wgs.phase1.release.v3.20101123,
                        MafDb.ExAC.r0.3.sites,
                        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.UCSC.hg19",
  orgdb = "org.Hs.eg.db", txdb = "TxDb.Hsapiens.UCSC.hg19.knownGene",
  snpdb = "SNPlocs.Hsapiens.dbSNP.20120608", spliceSiteMatricesFileNames = NA,
  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("SGC0"),
  allTranscripts = FALSE, otherAnnotations = c("MafDb.ESP6500SI.V2.SSA137",
    "MafDb.ALL.wgs.phase1.release.v3.20101123", "MafDb.ExAC.r0.3.sites",
    "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)

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

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

Quality filters

	INDELQual	LowQual
	TRUE	TRUE
	SNPQual	VQSRTrancheINDEL99.00to99.90
	TRUE	TRUE
VQSRTrancheINDEL99.90to100.00+	VQSRTrancheINDEL99.90to100.00	
	TRUE	TRUE
VQSRTrancheSNP99.00to99.90	VQSRTrancheSNP99.90to100.00+	
	TRUE	TRUE
VQSRTrancheSNP99.90to100.00		
	TRUE	

Functional annotation filters

dbSNP	OMIM	variantType	aaChangeType	S0terms
FALSE	FALSE	FALSE	FALSE	FALSE

Populations used for MAF filtering: AFESP, EA_AFESP, AA_AFESP, AFKG, AMR_AFKG, ASN_AFKG, AFR_AFKG, EUR,

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	S0:0001629	splice_site_variant	2	0.33
2	S0:0001583	missense_variant	15	2.45
3	S0:0002012	start_lost	1	0.16
4	S0:0001587	stop_gained	1	0.16
5	S0:0001624	3_prime_UTR_variant	16	2.62
6	S0:0001623	5_prime_UTR_variant	7	1.15
7	S0:0001631	upstream_gene_variant	43	7.04
8	S0:0001627	intron_variant	308	50.41
9	S0:0001819	synonymous_variant	24	3.93
10	S0:0001628	intergenic_variant	268	43.86
11	S0:0001589	frameshift_variant	1	0.16

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
1	spliceSite	2	0.33
2	intron	308	50.41
3	fiveUTR	7	1.15
4	threeUTR	16	2.62
5	coding	41	6.71
6	intergenic	268	43.86
7	promoter	43	7.04
8	frameshift	1	0.16
9	nonsense	1	0.16
10	nonsynonymous	15	2.45
11	synonymous	24	3.93

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	S0:0001629	6	splice_site_variant	2	0.33
2	S0:0001568	5	splicing_variant	2	0.33
3	S0:0001624	7	3_prime_UTR_variant	16	2.62
4	S0:0001627	5	intron_variant	308	50.41
5	S0:0001623	7	5_prime_UTR_variant	7	1.15
6	S0:0001622	6	UTR_variant	23	3.76
7	S0:0001583	10	missense_variant	15	2.45
8	S0:0002012	8	start_lost	1	0.16
9	S0:0001582	7	initiator_codon_variant	1	0.16
10	S0:0001589	8	frameshift_variant	1	0.16
11	S0:0001587	4	stop_gained	1	0.16
12	S0:0001906	3	feature_truncation	1	0.16
13	S0:0001992	9	nonsynonymous_variant	16	2.62
14	S0:0001650	8	inframe_variant	16	2.62
15	S0:0001818	7	protein_altering_variant	17	2.78
16	S0:0001631	4	upstream_gene_variant	43	7.04
17	S0:0001628	3	intergenic_variant	299	48.94
18	S0:0001819	7	synonymous_variant	24	3.93
19	S0:0001580	6	coding_sequence_variant	41	6.71
20	S0:0001968	5	coding_transcript_variant	59	9.66
21	S0:0001791	5	exon_variant	59	9.66
22	S0:0001576	4	transcript_variant	353	57.77
23	S0:0001564	3	gene_variant	353	57.77
24	S0:0001878	2	feature_variant	611	100.00
25	S0:0001537	1	structural_variant	611	100.00
26	S0:0001060	0	sequence_variant	611	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
1	S0:0001629	6	splice_site_variant	2	0.33
6	S0:0001622	6	UTR_variant	23	3.76
19	S0:0001580	6	coding_sequence_variant	41	6.71

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")
```

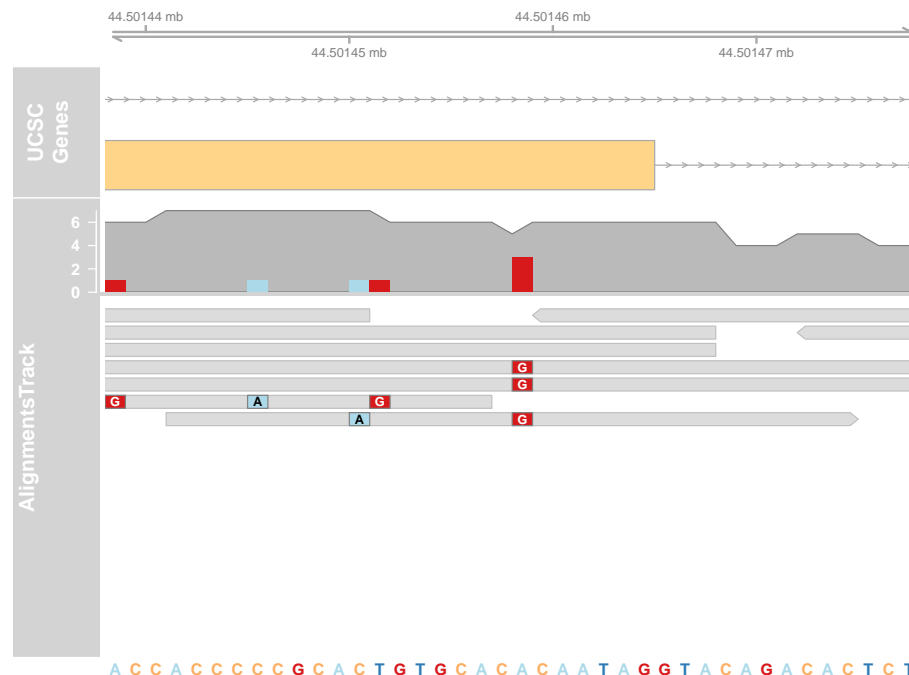


Figure 1: Browser-like display of a variant.

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)

Number of individuals: 2 (NA12891, NA12892)

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

Quality filters

	INDELQual	LowQual
	TRUE	TRUE
	SNPQual	VQSRTrancheINDEL99.00to99.90
	TRUE	TRUE
VQSRTrancheINDEL99.90to100.00+	VQSRTrancheINDEL99.90to100.00	
	TRUE	TRUE
VQSRTrancheSNP99.00to99.90	VQSRTrancheSNP99.90to100.00+	
	TRUE	TRUE
VQSRTrancheSNP99.90to100.00		
	TRUE	

Functional annotation filters

dbSNP	OMIM	variantType	aaChangeType	S0terms
FALSE	FALSE	FALSE	FALSE	FALSE

Populations used for MAF filtering: AFESP, EA_AFESP, AA_AFESP, AFKG, AMR_AFKG, ASN_AFKG, AFR_AFKG, EUR,

Include MAF NA values: yes

Maximum MAF: 1.00

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

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

	S0ID	Level	Description	Nr. Variants	% Variants
1	S0:0001629	6	splice_site_variant	1	0.19
6	S0:0001622	6	UTR_variant	20	3.75
19	S0:0001580	6	coding_sequence_variant	41	7.69

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)

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

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

Quality filters

	INDELQual	LowQual
	TRUE	TRUE
	SNPQual	VQSRTrancheINDEL99.00to99.90

```

                TRUE                TRUE
VQSRTrancheINDEL99.90to100.00+ VQSRTrancheINDEL99.90to100.00
                TRUE                TRUE
      VQSRTrancheSNP99.00to99.90 VQSRTrancheSNP99.90to100.00+
                TRUE                TRUE
VQSRTrancheSNP99.90to100.00
                TRUE
Functional annotation filters
      dbSNP      OMIM  variantType aaChangeType      S0terms
      FALSE      FALSE      FALSE      FALSE      FALSE
Populations used for MAF filtering: AFESP, EA_AFESP, AA_AFESP, AFKG, AMR_AFKG, ASN_AFKG, AFR_AFKG, EUR.
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 14
names(14): INDELQual LowQual SNPQual ... 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
                TRUE                TRUE
                SNPQual  VQSRTrancheINDEL99.00to99.90
                TRUE                TRUE
VQSRTrancheINDEL99.90to100.00+ VQSRTrancheINDEL99.90to100.00
                TRUE                TRUE
      VQSRTrancheSNP99.00to99.90 VQSRTrancheSNP99.90to100.00+
                TRUE                TRUE
VQSRTrancheSNP99.90to100.00                dbSNP
                TRUE                FALSE
                OMIM                variantType
                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
                TRUE                TRUE
                SNPQual  VQSRTrancheINDEL99.00to99.90
                TRUE                TRUE
VQSRTrancheINDEL99.90to100.00+ VQSRTrancheINDEL99.90to100.00
                TRUE                TRUE
      VQSRTrancheSNP99.00to99.90 VQSRTrancheSNP99.90to100.00+
                TRUE                TRUE

```

```

VQSRTrancheSNP99.90to100.00      dbSNP
                                TRUE
                                TRUE
                                OMIM
                                variantType
                                TRUE
                                TRUE
                                aaChangeType
                                S0terms
                                TRUE
                                TRUE

```

```
> summary(uind)
```

	SOID	Description	Nr. Variants	% Variants
1	S0:0001583	missense_variant	7	3.07
2	S0:0002012	start_lost	1	0.44
3	S0:0001624	3_prime_UTR_variant	7	3.07
4	S0:0001623	5_prime_UTR_variant	5	2.19
5	S0:0001631	upstream_gene_variant	29	12.72
6	S0:0001627	intron_variant	194	85.09
7	S0:0001819	synonymous_variant	20	8.77
8	S0:0001628	intergenic_variant	6	2.63

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)

```

	SOID	Description	Nr. Variants	% Variants
1	S0:0001629	splice_site_variant	1	0.12
2	S0:0001583	missense_variant	18	2.25
3	S0:0002012	start_lost	1	0.12
4	S0:0001587	stop_gained	1	0.12
5	S0:0001624	3_prime_UTR_variant	21	2.62
6	S0:0001623	5_prime_UTR_variant	7	0.88
7	S0:0001631	upstream_gene_variant	49	6.12
8	S0:0001627	intron_variant	361	45.12
9	S0:0001819	synonymous_variant	27	3.38
10	S0:0001628	intergenic_variant	406	50.75

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

```

```
> summary(uind, method="S0full")
```

	SOID	Level	Description	Nr. Variants	% Variants
1	S0:0001624	7	3_prime_UTR_variant	21	75
2	S0:0001623	7	5_prime_UTR_variant	7	25
3	S0:0001622	6	UTR_variant	28	100
4	S0:0001968	5	coding_transcript_variant	28	100
5	S0:0001791	5	exon_variant	28	100
6	S0:0001576	4	transcript_variant	28	100
7	S0:0001564	3	gene_variant	28	100

8	SO:0001878	2	feature_variant	28	100
9	SO:0001537	1	structural_variant	28	100
10	SO:0001060	0	sequence_variant	28	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="SOfull"`, 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
	FALSE	FALSE
	SNPQual	VQSRTrancheINDEL99.00to99.90
	FALSE	FALSE
VQSRTrancheINDEL99.90to100.00.	VQSRTrancheINDEL99.90to100.00	
	FALSE	FALSE
VQSRTrancheSNP99.00to99.90	VQSRTrancheSNP99.90to100.00.	
	FALSE	FALSE
VQSRTrancheSNP99.90to100.00		dbSNP
	FALSE	FALSE
OMIM		variantType
FALSE		FALSE
aaChangeType		SOterms
FALSE		FALSE
startLost		
TRUE		

```
> summary(uind)
```

	SOID	Description	Nr. Variants	% Variants
1	SO:0001583	missense_variant	1	100
2	SO:0002012	start_lost	1	100
3	SO:0001623	5_prime_UTR_variant	1	100
4	SO:0001631	upstream_gene_variant	1	100
5	SO: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 minimum allele frequency (MAF) values:

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

AFESP	EA_AFESP	AA_AFESP	AFKG	AMR_AFKG	ASN_AFKG	AFR_AFKG
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
EUR_AFKG	AFExAC	AFR_AFExAC	AMR_AFExAC	Adj_AFExAC	EAS_AFExAC	FIN_AFExAC
TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
NFE_AFExAC	OTH_AFExAC	SAS_AFExAC				
TRUE	TRUE	TRUE				

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

AFESP	EA_AFESP	AA_AFESP	AFKG	AMR_AFKG	ASN_AFKG	AFR_AFKG
FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
EUR_AFKG	AFExAC	AFR_AFExAC	AMR_AFExAC	Adj_AFExAC	EAS_AFExAC	FIN_AFExAC
FALSE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
NFE_AFExAC	OTH_AFExAC	SAS_AFExAC				
FALSE	FALSE	FALSE				

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

	SOID	Description	Nr. Variants	% Variants
1	S0:0001583	missense_variant	7	2.55
2	S0:0001624	3_prime_UTR_variant	7	2.55
3	S0:0001631	upstream_gene_variant	11	4.00
4	S0:0001627	intron_variant	130	47.27
5	S0:0001819	synonymous_variant	5	1.82
6	S0:0001628	intergenic_variant	138	50.18
7	S0:0001589	frameshift_variant	1	0.36

In this case we selected variants with $MAF < 0.01$ in the asian 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.

8 Session information

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

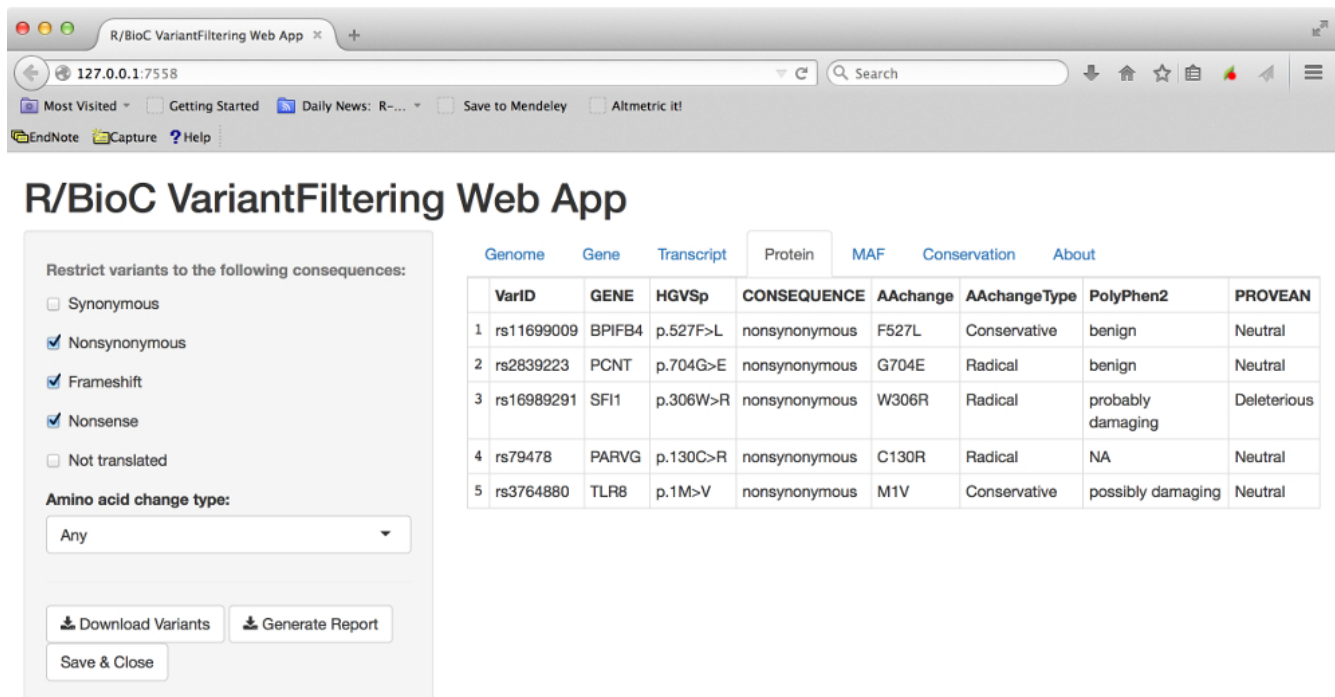


Figure 2: Snapshot of the shiny web app run from VariantFiltering with the function `reportVariants()`. Some of the parameters has been filled for illustrative purposes.

- R version 3.2.0 Patched (2015-04-19 r68206), x86_64-apple-darwin10.8.0
- Locale: en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
- Base packages: base, datasets, graphics, grDevices, methods, parallel, stats, stats4, utils
- Other packages: AnnotationDbi 1.30.1, Biobase 2.28.0, BiocGenerics 0.14.0, Biostrings 2.36.1, BSgenome 1.36.0, BSgenome.Hsapiens.UCSC.hg19 1.4.0, DBI 0.3.1, GenomInfoDb 1.4.0, GenomicFeatures 1.20.1, GenomicRanges 1.20.3, IRanges 2.2.1, MafDb.ALL.wgs.phase1.release.v3.20101123 3.1.0, MafDb.ESP6500SI.V2.SSA137 3.1.0, MafDb.ExAC.r0.3.sites 3.1.0, org.Hs.eg.db 3.1.2, phastCons100way.UCSC.hg19 3.1.0, PolyPhen.Hsapiens.dbSNP131 1.0.2, Rsamtools 1.20.2, RSQLite 1.0.0, rtracklayer 1.28.3, S4Vectors 0.6.0, SIFT.Hsapiens.dbSNP137 1.0.0, SNPlocs.Hsapiens.dbSNP.20120608 0.99.9, TxDb.Hsapiens.UCSC.hg19.knownGene 3.1.2, VariantAnnotation 1.14.1, VariantFiltering 1.4.3, XVector 0.8.0
- Loaded via a namespace (and not attached): acepack 1.3-3.3, BiocParallel 1.2.2, BiocStyle 1.6.0, biomaRt 2.24.0, biovizBase 1.16.0, bitops 1.0-6, cluster 2.0.1, colorspace 1.2-6, dichromat 2.0-0, digest 0.6.8, foreign 0.8-63, Formula 1.2-1, futile.logger 1.4.1, futile.options 1.0.0, GenomicAlignments 1.4.1, ggplot2 1.0.1, graph 1.46.0, grid 3.2.0, gridExtra 0.9.1, gtable 0.1.2, Gviz 1.12.0, Hmisc 3.16-0, htmltools 0.2.6, httpuv 1.3.2, lambda.r 1.1.7, lattice 0.20-31, latticeExtra 0.6-26, magrittr 1.5, MASS 7.3-40, matrixStats 0.14.0, mime 0.3, munsell 0.4.2, nnet 7.3-9, plyr 1.8.2, proto 0.3-10, R6 2.0.1, RBGL 1.44.0, RColorBrewer 1.1-2, Rcpp 0.11.6, RCurl 1.95-4.6, reshape2 1.4.1, rpart 4.1-9, scales 0.2.4, shiny 0.12.0, splines 3.2.0, stringi 0.4-1, stringr 1.0.0, survival 2.38-1, tools 3.2.0, XML 3.98-1.1, xtable 1.7-4, zlibbioc 1.14.0