An Introduction to *VariantTools*

Michael Lawrence, Jeremiah Degenhardt

April 29, 2014

Contents

1 Introduction **2**

2 Calling single-sample variants **2**
2.1 Basic usage 2
2.2 Step by step 3
2.3 Diagnosing the filters 4
2.4 Extending and customizing the workflow 6

3 Comparing variant sets across samples **6**
3.1 Calling sample-specific variants 6

4 Exporting the calls as VCF **6**

5 Finding Wildtype and No-call Regions **7**

1 Introduction

This vignette outlines the basic usages of the *VariantTools*package and the general workflow for loading data, calling single sample variants and tumor-specific somatic mutations or other sample-specific variant types (eg RNA editing). Most of the functions operate on alignments (BAM files) or datasets of called variants. The user is expected to have already aligned the reads with a separate tool, e.g., GSNAP via *gmapR*.

2 Calling single-sample variants

2.1 Basic usage

For our example, we take paired-end RNA-seq alignments from two lung cancer cell lines from the same individual. H1993 is derived from a metastatis and H2073 is derived from the primary tumor.

Below, we call variants from a region around the p53 gene:

```
> library(VariantTools)
> p53 <- gmapR:::exonsOnTP53Genome("TP53")
> bams <- LungCancerLines::LungCancerBamFiles()
> bam <- bams$H1993
> tally.param <- TallyVariantsParam(gmapR::TP53Genome(),
+                                   high_base_quality = 23L,
+                                   which = range(p53) + 5e4,
+                                   indels = TRUE, read_length = 75L)
> called.variants <- callVariants(bam, tally.param)
```

In the above, we load the genome corresponding to the human p53 gene region and the H1993 BAM file (stripped down to the same region). We pass the BAM, genome, read length and quality cutoff to the `callVariants` workhorse. The read length is not strictly required, but it is necessary for one of the QA filters. The value given for the high base quality cutoff is appropriate for Sanger and Illumina 1.8 or above. By default, the high quality counts are used by the likelihood ratio test during calling.

The returned `called_variants` is a variant *GRanges*, in the same form as that returned by `bam_tally` in the *gmapR* package. Unsurprisingly, `callVariants` uses `bam_tally` internally to generate the per-nucleotide counts (pileup) from the BAM file. The result is then filtered to generate the variant calls. The *VCF* class holds similar information; however, we favor the simple tally *GRanges*, because it has a separate record for each ALT, at each position. *VCF*, the class and the file format, has a single record for a position, collapsing over multiple ALT alleles, and this is much less convenient for our purposes.

We can post-filter the variants for those that are clustered too closely on the genome:

```
> pf.variants <- postFilterVariants(called.variants)
```

We can subset the variants by those in an actual p53 exon (not an intron):

```
> subsetByOverlaps(called.variants, p53, ignore.strand = TRUE)
```

```
VRanges with 3 ranges and 15 metadata columns:
      seqnames             ranges strand         ref              alt
         <Rle>          <IRanges>  <Rle> <character> <characterOrRle>
  [1]     TP53 [1012459, 1012459]      +           G               C
  [2]     TP53 [1013114, 1013114]      +           T               C
  [3]     TP53 [1014376, 1014376]      +           G               C
          totalDepth       refDepth       altDepth   sampleNames
      <integerOrRle> <integerOrRle> <integerOrRle> <factorOrRle>
  [1>              8              0              8          <NA>
```

```
[2]              4              0              4          <NA>
[3]              4              0              4          <NA>
    softFilterMatrix | raw.count raw.count.ref raw.count.total
            <matrix> | <integer>     <integer>       <integer>
[1]                  |         8             0               8
[2]                  |         4             0               4
[3]                  |         4             0               4
    mean.quality mean.quality.ref count.plus count.plus.ref
       <numeric>        <numeric>  <integer>      <integer>
[1]       35.375              NaN          6              0
[2]        35.75              NaN          0              0
[3]        34.75              NaN          1              0
    count.minus count.minus.ref read.pos.mean read.pos.mean.ref
      <integer>       <integer>     <numeric>         <numeric>
[1]           2               0        64.125               NaN
[2]           4               0         49.75               NaN
[3]           3               0          31.5               NaN
    read.pos.var read.pos.var.ref     mdfne mdfne.ref
       <numeric>        <numeric> <numeric> <numeric>
[1]   990.984375             <NA>        17      <NA>
[2]    2435.9375             <NA>        17      <NA>
[3]      1363.75             <NA>        14      <NA>
---
seqlengths:
    TP53
 2025772
hardFilters(4): nonRef nonNRef readCount likelihoodRatio
```

The next section goes into further detail on the process, including the specific filtering rules applied, and how one might, for example, tweak the parameters to avoid calling low-coverage variants, like the one above.

2.2 Step by step

The `callVariants` method for BAM files, introduced above, is a convenience wrapper that delegates to several low-level functions to perform each step of the variant calling process: generating the tallies, basic QA filtering and the actual variant calling. Calling these functions directly affords the user more control over the process and provides access to intermediate results, which is useful e.g. for diagnostics and for caching results. The workflow consists of three function calls that rely on argument defaults to achieve the same result as our call to `callVariants` above. Please see their man pages for the arguments available for customization.

The first step is to tally the variants from the BAM file. By default, this will return observed differences from the reference, excluding N calls and only counting reads above 13 in mapping quality (MAPQ) score. There are three read position bins: the first 10 bases, the final 10 bases, and the stretch between them (these will be used in the QA step).

```
> raw.variants <- tallyVariants(bam, tally.param)
```

Unless one is running a variant caller in a routine fashion over familiar types of data, we highly recommend performing detailed QC of the tally results. *VariantTools* provides several QA filters that aim to expose artifacts, especially those generated during alignment. These filters are *not* designed for filtering during actual calling; rather, they are meant for annotating the variants during exploratory analysis. The filters include a check on the median distance of alt calls from their nearest end of the read (default passing cutoff

>= 10), as well as a Fisher Exact Test on the per-strand counts vs. reference for strand bias (p-value cutoff: 0.001). The intent is to ensure that the data are not due to strand-specific nor read position-specific artifacts.

The `qaVariants` function will *soft* filter the variants via `softFilter`. No variants are removed; the filter results are added to the `softFilterMatrix` component of the object.

```
> qa.variants <- qaVariants(raw.variants)
> summary(softFilterMatrix(qa.variants))
```

```
   <initial>        mdfne fisherStrand      <final>
          95           82           95           82
```

The final step is to actually call the variants. The `callVariants` function uses a binomial likelihood ratio test for this purpose. The ratio is $P(D|p = p_{lower})/P(D|p = p_{error})$, where $p_{lower} = 0.2$ is the assumed lowest variant frequency and $p_{error} = 0.001$ is the assumed error rate in the sequencing (default: 0.001).

```
> called.variants <- callVariants(qa.variants)
```

The `callVariants` function applies an additional set of filters after the actual variant calling. These are known as "post" filters and consider the putative variant calls as a set, independent of the calling algorithm. Currently, there is only one post filter by default, and it discards variants that are clumped together along the chromosome, as these often result from mapping difficulties.

2.3 Diagnosing the filters

The calls to `qaVariants` and `callVariants` are essentially filtering the tallies, so it is important to know, especially when faced with a new dataset, the effect of each filter and the effect of the individual parameters on each filter.

The filters are implemented as modules and are stored in a *FilterRules* object from the *IRanges* package. We can create those filters directly and rely on some *FilterRules* utilities to diagnose the filtering process.

Here we construct the *FilterRules* that implements the `qaVariants` function. Again, we rely on the argument defaults to generate the same answer.

```
> qa.filters <- VariantQAFilters()
```

We can now ask for a summary of the filtering process, which gives the number of variants that pass each filter, separately and then combined:

```
> summary(qa.filters, raw.variants)
```

```
   <initial>        mdfne fisherStrand      <final>
          95           82           95           82
```

Now we retrieve only the variants that pass the filters:

```
> qa.variants <- subsetByFilter(raw.variants, qa.filters)
```

We could do the same, except modify a filter parameter, such as the p-value cutoff for the Fisher Exact Test for strand bias:

```
> qa.filters.custom <- VariantQAFilters(fisher.strand.p.value = 1e-4)
> summary(qa.filters.custom, raw.variants)
```

```
   <initial>        mdfne fisherStrand      <final>
          95           82           95           82
```

To get a glance at the additional variants we are discarding compared to the previous cutoff, we can subset the filter sets down to the Fisher strand filter, evaluate the old and new filter, and compare the results:

```
> fs.original <- eval(qa.filters["fisherStrand"], raw.variants)
> fs.custom <- eval(qa.filters.custom["fisherStrand"], raw.variants)
> raw.variants[fs.original != fs.custom]

VRanges with 0 ranges and 15 metadata columns:
   seqnames    ranges strand         ref              alt
      <Rle> <IRanges>  <Rle> <character> <characterOrRle>
       totalDepth       refDepth       altDepth   sampleNames
   <integerOrRle> <integerOrRle> <integerOrRle> <factorOrRle>
   softFilterMatrix | raw.count raw.count.ref raw.count.total
           <matrix> | <integer>     <integer>       <integer>
   mean.quality mean.quality.ref count.plus count.plus.ref count.minus
      <numeric>        <numeric>  <integer>      <integer>   <integer>
   count.minus.ref read.pos.mean read.pos.mean.ref read.pos.var
         <integer>     <numeric>         <numeric>    <numeric>
   read.pos.var.ref     mdfne mdfne.ref
          <numeric> <numeric> <numeric>
  ---
  seqlengths:
      TP53
   2025772
  hardFilters: NULL
```

Below, we demonstrate how one might add a mask to e.g. filter out variants in low complexity regions, where mapping errors tend to dominate:

```
> tally.param@mask <- GRanges("TP53", IRanges(1010000, width=10000))
> raw.variants.masked <- tallyVariants(bam, tally.param)
```

We can also diagnose the filters for calling variants after basic QA checks.

```
> calling.filters <- VariantCallingFilters()
> summary(calling.filters, qa.variants)

      <initial>          nonRef         nonNRef       readCount
             82              82              82               3
likelihoodRatio         <final>
             16               3
```

Check how the post filter would perform prior to variant calling:

```
> post.filters <- VariantPostFilters()
> summary(post.filters, qa.variants)

   <initial> avgNborCount      <final>
          82           63           63
```

What about if we preserved the ones we have already called?

```
> post.filters <- VariantPostFilters(whitelist = called.variants)
> summary(post.filters, qa.variants)

   <initial> avgNborCount      <final>
          82           68           68
```

2.4 Extending and customizing the workflow

Since the built-in filters are implemented using *FilterRules*, it is easy to mix and match different filters, including those implemented externally to the *VariantTools* package. This is the primary means of extending and customizing the variant calling workflow.

3 Comparing variant sets across samples

So far, we have called variants for the metastatic H1993 sample. We leave the processing of the primary tumor H2073 sample as an exercise to the reader and instead turn our attention to detecting the variants that are specific to the metastatic sample, as compared to the primary tumor.

3.1 Calling sample-specific variants

The function `callSampleSpecificVariants` takes the case (e.g., tumor) sample and control (e.g., matched normal) sample as input. In our case, we are comparing the metastatic line (H1993) to the primary tumor line (H2073) from the same patient, a smoker. To avoid inconsistencies, it is recommended to pass BAM files as input, for which tallies are automatically generated, subjected to QA, and called as variants vs. reference, prior to determining the sample-specific variants.

Here, we find the somatic mutations from a matched tumor/normal pair. Since we are starting from BAM files, we have to provide `tally.param` for the tally step.

```
> tally.param@bamTallyParam@indels <- FALSE
> somatic <- callSampleSpecificVariants(bams$H1993, bams$H2073, tally.param)
```

This can be time-consuming for the entire genome, since the tallies need to be computed. To avoid repeated computation of the tallies, the user can pass the raw tally *GRanges* objects instead of the BAM files. This is less safe, because anything could have happened to those *GRanges* objects.

The QA and initial calling are optionally controlled by passing *FilterRules* objects, typically those returned by `VariantQAFilters` and `VariantCallingFilters`, respectively. For controlling the final step, determining the sample-specific variants, one may pass filter parameters directly to `callSampleSpecificVariants`. Here is an example of customizing some parameters.

```
> calling.filters <- VariantCallingFilters(read.count = 3L)
> somatic <- callSampleSpecificVariants(bams$H1993, bams$H2073, tally.param,
+                                       calling.filters = calling.filters,
+                                       power = 0.9, p.value = 0.001)
```

4 Exporting the calls as VCF

VCF is a common file format for communicating variants. To export our variants to a VCF file, we first need to coerce the *GRanges* to a *VCF* object. Then, we use `writeVcf` from the *VariantAnnotation* package to write the file (indexing is highly recommended for large files). Note that the sample names need to be non-missing to generate the VCF. Also, for simplicity and scalability, we typically do not want to output all of our metadata columns, so we remove all of them here.

```
> sampleNames(called.variants) <- "H1993"
> mcols(called.variants) <- NULL
> vcf <- asVCF(called.variants)

> writeVcf(vcf, "H1993.vcf", index = TRUE)
```

5 Finding Wildtype and No-call Regions

So far, our analysis has yielded a set of positions that are likely to be variants. We have not made any claims about the status of the positions outside of that set. For this, we need to decide, for each position, whether there was sufficient coverage to detect a variant, if one existed. The following call carries out a power test to decide whether a region is variant, wildtype or is unable to be called due to lack of coverage. The variants must have been called using the filters returned by `VariantCallingFilters`. The algorithm depends on the filter parameter settings, so it is possible and indeed required for the user to pass filter object used for calling the variants. This requirement is an attempt to ensure consistency and will be made more convenient in the future. To request the calls for a particular set of positions, one can pass a *GenomicRanges* (where all the ranges are of width 1) as the `pos` argument. When `pos` is specfied, each element of the result corresponds to an element in `pos`.

```
> called.variants <- called.variants[!isIndel(called.variants)]
> pos <- c(called.variants, shift(called.variants, 3))
> wildtype <- callWildtype(bam, called.variants, VariantCallingFilters(),
+                          pos = pos, power = 0.85)
```

The returned object is a logical *RleList* object, with `TRUE` for wildtype, `FALSE` for variant and `NA` for no-call. Thus, we could calculate the fraction called as follows:

```
> mean(!is.na(wildtype))
```

```
[1] 0.5
```

Sometimes it is desirable for the wildtype calls to be returned as simple vector, with a logical value for each position (range of width one) in `bamWhich`. Such a vector is returned when `global = FALSE` is passed to `callWildtype`. This is the same as extracting the positions from the ordinary *Rle* return value, but it is implemented more efficiently, at least for a relatively small number of positions.