Software Manual
Institute of Bioinformatics, Johannes Kepler University Linz

PODKAT

An R Package for Association Testing Involving Rare and Private Variants

Ulrich Bodenhofer

Institute of Bioinformatics, Johannes Kepler University Linz
Altenberger Str. 69, 4040 Linz, Austria
podkat@bioinf.jku.at

Version 1.32.0, November 16, 2022

Scope and Purpose of this Document

This document is a user manual for PODKAT, an R package implementing non-burden association tests for rare and private variants, most importantly, the *position-dependent kernel association test (PODKAT)*. It provides a gentle introduction into how to use PODKAT. Not all features of the R package are described in full detail. Such details can be obtained from the documentation enclosed in the R package. Further note the following: (1) this is not an introduction to statistical genetics or association testing; (2) this is not an introduction to R or any of the Bioconductor packages used in this document; (3) this is not an introduction to statistical hypothesis testing or probability theory. If you lack the background for understanding this manual, you first have to read introductory literature on the subjects mentioned above.

All R code in this document is written to be runnable by any user. However, some of the code chunks require the download of external files, require an Internet connection, or require too much computation time to be runnable when the package is built, checked, or installed. The output lines of R code chunks that are not actually executed when processing this document are marked with '`##!!##`' and, in case that the user needs to perform extra steps to execute the code, these steps are listed explicitly.

Contents

1 Introduction

This user manual describes the R package PODKAT. This R package implements non-burden association tests for rare and private variants, most importantly, the *position-dependent kernel association test (PODKAT)*.

Before discussing details of how to use the package, let us first discuss the general aim and setup of association studies. Suppose we have a certain number of *samples* (study participants, patients, etc.) for each of which we can measure/sequence the *genotype* and for each of which we know/have measured/have observed a certain *trait* that we want to study. This trait may be *continuous*, i.e. real-valued on a continuous scale (for instance, age, height, body mass index, etc.), or *categorical*, i.e. from a discrete set of categories (for instance, case vs. control, treatment outcome, disease type, etc.). In the following, we will only consider continuous traits and categorical traits with two categories and refer to this case as a *binary trait* (sometimes called *dichotomous trait* as well) with values 0 or 1.

The goal of association testing is to find out whether there are any *statistically significant associations between the genotype and the trait*.

In some studies, additional information about the samples' phenotypes or environmental conditions is available that might also have an influence on the trait (for instance, age, sex, ethnicity, family status, etc.). Such additional features can be treated as *covariates*. More specifically, it is rather common to train a model that predicts the trait from the covariates first. Then the association between the genotype and those components of the traits is studied which have not been sufficiently explained by the covariates. In the case of PODKAT, this is done by a *kernel-based variance-component score test* [10, 16].

Assume that, for a given set of samples, we are given a trait vector (one entry for each sample), genotypes of all samples (in matrix format or as a VCF file[1]), and a matrix of covariates (if any). Then an association test using PODKAT consists of the following basic steps:

Training of null model: pre-processing of trait vector and covariates (if any) for later use in an association test (see Section 4);

Selection of regions of interest: specification of one or more genomic regions for which association tests should be performed (see Section 5);

Association testing: testing of association between genotype and trait/null model for each selected region of interest (see Section 6);

Analysis of results: post-processing (such as, multiple testing correction or filtering) and visualization of results (see Section 7);

Figure 1 shows a graphical overview of these basic steps along with dependencies and data types.

This manual is organized as follows: after some basic instructions how to install the package (Section 2), Section 3 provides a simple, yet complete, example that illustrates the general workflow. Sections 4–7 provide more details about the steps necessary to perform association tests with PODKAT. Sections 8–11 provide miscellaneous additional information.

[1] Variant Call Format; see `http://www.1000genomes.org/wiki/analysis/variant-call-format/vcf-variant-call-format-version-42` (last visited 2021-04-30) for a detailed specification of this file format

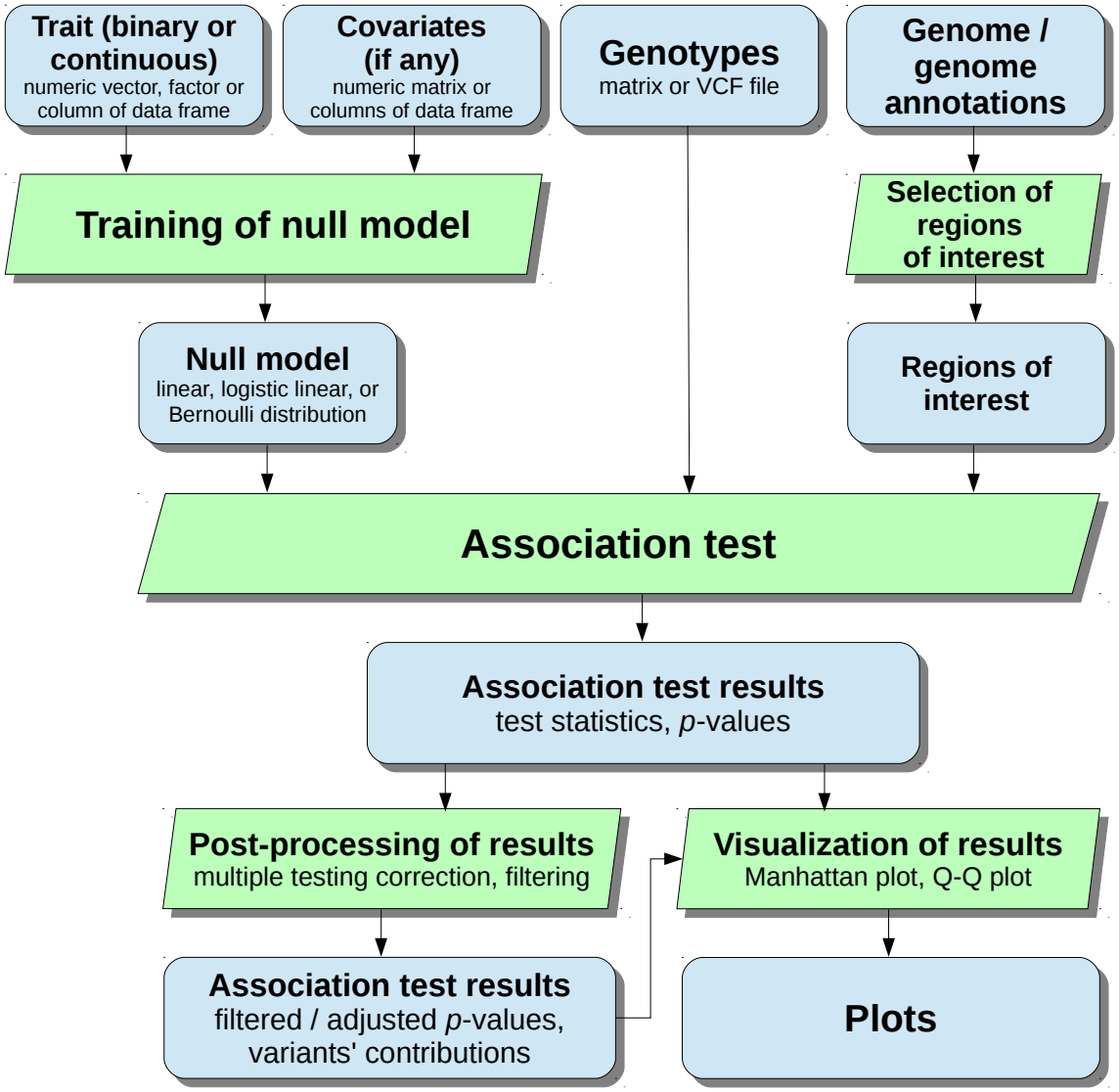


Figure 1: Overview of the basic steps of the data analysis pipeline offered by PODKAT for analyzing associations between traits and genotypes.

2 Installation

The PODKAT R package (current version: 1.32.0) is available via Bioconductor. The simplest way to install the package is the following:

```
if (!requireNamespace("BiocManager", quietly=TRUE))
    install.packages("BiocManager")
BiocManager::install("podkat")
```

If you wish to install the package manually instead, you can download the package archive that fits best to your computer system from the Bioconductor homepage.

To test the installation of the PODKAT package, enter

```
library(podkat)
```

in your R session. If this command terminates without any error message or warning, you can be sure that the PODKAT package has been installed successfully. If so, the PODKAT package is ready for use now and you can start performing association tests.

3 PODKAT for the Impatient

In order to illustrate the basic workflow, this section presents two simple examples without going into the details of each step. Let us first retrieve the file names of the example files that are supplied as part of the PODKAT package:

```
phenoFileLin <- system.file("examples/example1lin.csv", package="podkat")
phenoFileLog <- system.file("examples/example1log.csv", package="podkat")
vcfFile <- system.file("examples/example1.vcf.gz", package="podkat")
```

Now let us train the null model for the continuous trait contained in the file `example1lin.csv`:

```
pheno.c <- read.table(phenoFileLin, header=TRUE, sep=",")
model.c <- nullModel(y ~ ., pheno.c)
model.c

## Linear model:
##  Number of covariates: 2 (+ intercept)
##  Number of samples: 200
##  Variance of residuals: 1.541756
##  No resampling
```

The examples are based on the small artificial genome `hgA` that is also supplied as part of PODKAT. So we load it first and then partition it into overlapping windows:

```
data(hgA)
hgA

## GRanges object with 1 range and 0 metadata columns:
##       seqnames    ranges strand
##          <Rle> <IRanges>  <Rle>
##   [1]     chr1  1-200000      *
##   -------
##   seqinfo: 1 sequence from hgA genome

windows <- partitionRegions(hgA)
windows

## GRanges object with 79 ranges and 0 metadata columns:
##        seqnames        ranges strand
##           <Rle>     <IRanges>  <Rle>
##    [1]     chr1        1-5000      *
##    [2]     chr1     2501-7500      *
##    [3]     chr1    5001-10000      *
##    [4]     chr1    7501-12500      *
##    [5]     chr1   10001-15000      *
##    ...      ...           ...    ...
##   [75]     chr1 185001-190000      *
##   [76]     chr1 187501-192500      *
##   [77]     chr1 190001-195000      *
##   [78]     chr1 192501-197500      *
##   [79]     chr1 195001-200000      *
##   -------
##   seqinfo: 1 sequence from hgA genome
```

The VCF file used for these two examples is small enough to be loadable at once:

```
geno <- readGenotypeMatrix(vcfFile)
geno

## Genotype matrix:
##  Number of samples: 200
##  Number of variants: 962
##
##  Mean MAF:    0.05674116
##  Median MAF:  0.0075
##  Minimum MAF: 0.0025
##  Maximum MAF: 0.455
```

Now we can already perform the two association tests. Let us start with the continuous trait:

```
res.c <- assocTest(geno, model.c, windows)
print(res.c)

## Overview of association test:
##  Null model: linear
##  Number of samples: 200
##  Number of regions: 79
##  Number of regions without variants: 0
##  Average number of variants in regions: 24.1
##  Genome: hgA
##  Kernel: linear.podkat
##  p-value adjustment: none
##
## Overview of significance of results:
##  Number of tests with p < 0.05: 8
##
## Results for the 8 most significant regions:
##   seqnames  start    end width  n         Q      p.value
## 1     chr1   7501  12500  5000 31 769748.34 1.294084e-07
## 2     chr1  10001  15000  5000 33 764828.81 4.874460e-06
## 3     chr1 140001 145000  5000 15  79937.68 3.599077e-03
## 4     chr1   5001  10000  5000 34 152555.30 9.785569e-03
## 5     chr1 132501 137500  5000 21  89287.55 1.349559e-02
## 6     chr1 142501 147500  5000 23  94629.68 3.338620e-02
## 7     chr1  42501  47500  5000 19  58191.23 3.341032e-02
## 8     chr1  25001  30000  5000 23 103713.12 3.754557e-02
```

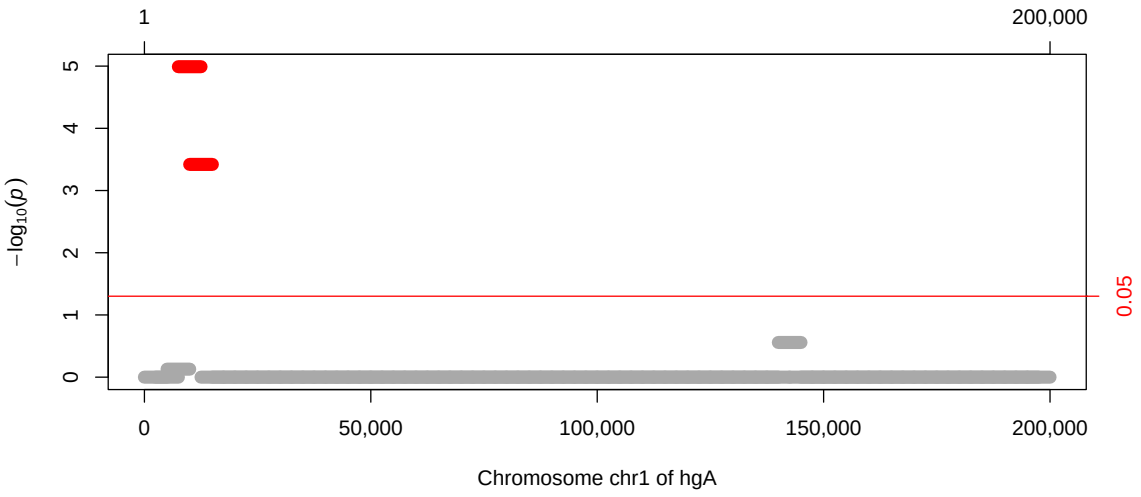
Now we perform multiple testing correction:

```
res.c <- p.adjust(res.c)
print(res.c)

## Overview of association test:
##  Null model: linear
##  Number of samples: 200
##  Number of regions: 79
##  Number of regions without variants: 0
##  Average number of variants in regions: 24.1
##  Genome: hgA
##  Kernel: linear.podkat
##  p-value adjustment: holm
##
## Overview of significance of results:
##  Number of tests with p < 0.05: 8
##  Number of tests with adj. p < 0.05: 2
```

```
##
## Results for the 8 most significant regions:
##   seqnames  start    end width  n         Q      p.value
## 1     chr1   7501  12500  5000 31 769748.34 1.294084e-07
## 2     chr1  10001  15000  5000 33 764828.81 4.874460e-06
## 3     chr1 140001 145000  5000 15  79937.68 3.599077e-03
## 4     chr1   5001  10000  5000 34 152555.30 9.785569e-03
## 5     chr1 132501 137500  5000 21  89287.55 1.349559e-02
## 6     chr1 142501 147500  5000 23  94629.68 3.338620e-02
## 7     chr1  42501  47500  5000 19  58191.23 3.341032e-02
## 8     chr1  25001  30000  5000 23 103713.12 3.754557e-02
##    p.value.adj
## 1 1.022327e-05
## 2 3.802079e-04
## 3 2.771289e-01
## 4 7.437033e-01
## 5 1.000000e+00
## 6 1.000000e+00
## 7 1.000000e+00
## 8 1.000000e+00
```

Finally, we create a Manhattan plot:

```
plot(res.c, which="p.value.adj")
```



For a binary trait, the whole pipeline looks the same. The `nullModel()` function automatically detects that the trait is binary and this information is passed on to the subsequent steps without the need of making additional settings:

```
pheno.b <- read.table(phenoFileLog, header=TRUE, sep=",")
model.b <- nullModel(y ~ ., pheno.b)

## small sample correction applied

model.b

## Logistic model:
##  Number of covariates: 2 (+ intercept)
##  Number of samples: 200
##  Number of positives (cases): 10
##  No resampling
##  Adjustment of higher moments: 10000 repeats (bootstrap)
```

Now we can already perform the association tests for the binary trait. This time, however, we do not load the entire genotype first, but we let `assocTest()` read from the VCF file directly (which is only done piece by piece in order to avoid excessive use of memory):

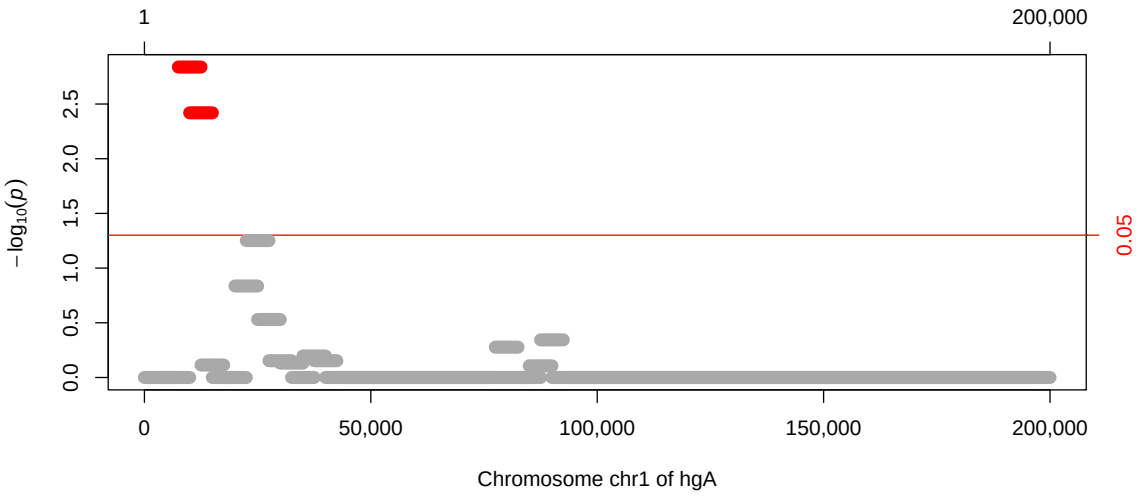
```
res.b <- assocTest(vcfFile, model.b, windows)
print(res.b)

## Overview of association test:
##  Null model: logistic
##  Number of samples: 200
##  Number of regions: 79
##  Number of regions without variants: 0
##  Average number of variants in regions: 24.1
##  Genome: hgA
##  Kernel: linear.podkat
##  p-value adjustment: none
##
## Overview of significance of results:
##  Number of tests with p < 0.05: 23
##
## Results for the 10 most significant regions:
##    seqnames start   end width  n        Q      p.value
## 1      chr1  7501 12500  5000 31 38386.55 1.837935e-05
## 2      chr1 10001 15000  5000 33 43084.90 4.878394e-05
## 3      chr1 22501 27500  5000 27 25640.34 7.271067e-04
## 4      chr1 20001 25000  5000 23 16120.63 1.917894e-03
## 5      chr1 25001 30000  5000 23 10650.48 3.935383e-03
## 6      chr1 87501 92500  5000 30 12891.61 6.121441e-03
## 7      chr1 77501 82500  5000 17  4890.26 7.239687e-03
## 8      chr1 35001 40000  5000 23 22436.29 8.838398e-03
## 9      chr1 37501 42500  5000 25 22570.67 9.892143e-03
## 10     chr1 27501 32500  5000 27 32423.04 9.914104e-03
```

Multiple testing correction again:

```
res.b <- p.adjust(res.b)
print(res.b)

## Overview of association test:
##  Null model: logistic
##  Number of samples: 200
##  Number of regions: 79
##  Number of regions without variants: 0
##  Average number of variants in regions: 24.1
##  Genome: hgA
##  Kernel: linear.podkat
##  p-value adjustment: holm
##
## Overview of significance of results:
##  Number of tests with p < 0.05: 23
##  Number of tests with adj. p < 0.05: 2
##
## Results for the 10 most significant regions:
##    seqnames start   end width  n        Q      p.value
## 1      chr1  7501 12500  5000 31 38386.55 1.837935e-05
## 2      chr1 10001 15000  5000 33 43084.90 4.878394e-05
## 3      chr1 22501 27500  5000 27 25640.34 7.271067e-04
## 4      chr1 20001 25000  5000 23 16120.63 1.917894e-03
## 5      chr1 25001 30000  5000 23 10650.48 3.935383e-03
## 6      chr1 87501 92500  5000 30 12891.61 6.121441e-03
## 7      chr1 77501 82500  5000 17  4890.26 7.239687e-03
## 8      chr1 35001 40000  5000 23 22436.29 8.838398e-03
## 9      chr1 37501 42500  5000 25 22570.67 9.892143e-03
## 10     chr1 27501 32500  5000 27 32423.04 9.914104e-03
##    p.value.adj
## 1  0.001451968
## 2  0.003805147
## 3  0.055987214
## 4  0.145759981
## 5  0.295153720
## 6  0.452986597
## 7  0.528497132
## 8  0.636364656
## 9  0.702342137
## 10 0.702342137
```

Finally, we create a Manhattan plot:

```
plot(res.b, which="p.value.adj")
```



The following sections provide details and more background information about the functions used in the above steps.

4 Training a Null Model

Before an association test can be performed, we have to pre-process the trait vector and create a so-called *null model*, i.e. a probabilistic model of the trait under the null assumption that the trait is independent of the genotype and only depends on the covariates (if any). PODKAT currently offers three types of such null models:

Linear model: the trait is continuous and depends linearly on the covariates, i.e.

$$y = \alpha_0 + \boldsymbol{\alpha}^T \cdot \mathbf{x} + \varepsilon,$$

where y is the trait, α_0 is the intercept, $\boldsymbol{\alpha}$ is a weight vector, $\mathbf{x}$ is the vector of covariates, and ε is normally distributed random noise. If there are no covariates, y is normally distributed around the intercept α_0.

Logistic linear model: the trait is binary and depends on the covariates in the following way (with the same notations as above):

$$\operatorname{logit}\big(p(y=1)\big) = \alpha_0 + \boldsymbol{\alpha}^T \cdot \mathbf{x}$$

If there are no covariates, y is a binary Bernoulli-distributed random variable with constant $p(y=1) = \operatorname{logit}^{-1}(\alpha_0) = 1/(1+\exp(-\alpha_0))$.

Bernoulli-distributed trait: the trait is binary, does not depend on any covariates, and follows a simple Bernoulli distribution with constant p.

PODKAT offers one function `nullModel()` that allows for the creation of any of the above three types of null models. In order to demonstrate how `nullModel()` works, we first load two examples that are shipped with the PODKAT package.

For the subsequent examples, we consider the two data frames `pheno.c` and `pheno.b` that we created in Section 3 and investigate them in more detail. The object `pheno.c` is a data frame with two covariate columns and one column `y` containing a continuous trait:

```
colnames(pheno.c)

## [1] "X.1" "X.2" "y"

summary(pheno.c)

##       X.1                X.2                 y
##  Min.   :-2.75343   Min.   :-3.56170   Min.   :-2.9853
##  1st Qu.:-0.68484   1st Qu.:-0.78153   1st Qu.:-0.3625
##  Median : 0.04127   Median :-0.02714   Median : 0.5965
##  Mean   : 0.02007   Mean   :-0.03562   Mean   : 0.5835
##  3rd Qu.: 0.78158   3rd Qu.: 0.73964   3rd Qu.: 1.6966
##  Max.   : 2.29429   Max.   : 2.73488   Max.   : 3.8589
```

The object `pheno.b` is a data frame with two covariate columns and one column `y` containing a binary trait.

```
colnames(pheno.b)

## [1] "X.1" "X.2" "y"

summary(pheno.b)

##       X.1                X.2                 y
##  Min.   :-2.75343   Min.   :-3.56170   Min.   :0.00
##  1st Qu.:-0.68484   1st Qu.:-0.78153   1st Qu.:0.00
##  Median : 0.04127   Median :-0.02714   Median :0.00
##  Mean   : 0.02007   Mean   :-0.03562   Mean   :0.05
##  3rd Qu.: 0.78158   3rd Qu.: 0.73964   3rd Qu.:0.00
##  Max.   : 2.29429   Max.   : 2.73488   Max.   :1.00

table(pheno.b$y)

##
##   0   1
## 190  10
```

As we have seen in Section 3 already, the simplest way of creating a null model is to call `nullModel()` via the formula interface, in a way that is largely analogous to the R standard functions `lm()` and `glm()`:

```
model.c <- nullModel(y ~ ., pheno.c)
model.c

## Linear model:
##  Number of covariates: 2 (+ intercept)
##  Number of samples: 200
##  Variance of residuals: 1.541756
##  No resampling

model.b <- nullModel(y ~ ., pheno.b)

## small sample correction applied

model.b

## Logistic model:
##  Number of covariates: 2 (+ intercept)
##  Number of samples: 200
##  Number of positives (cases): 10
##  No resampling
##  Adjustment of higher moments: 10000 repeats (bootstrap)
```

Note that, in the above calls to `nullModel()`, we did not explicitly specify the type of the model. Whenever the `type` argument is not specified, `nullModel()` tries to guess the right type of model. If the trait vector/column is a factor or a numeric vector containing only 0's and 1's (where both values must be present, otherwise an association test would be meaningless), the trait is supposed to be binary and a logistic linear model is trained, unless the following conditions are satisfied:

1. The number of samples does not exceed 100.
2. No intercept and no covariates have been specified.

If these two conditions are fulfilled for a binary trait, `nullModel()` considers the trait as a Bernoulli-distributed random variable (i.e. as the third type of model described above). If the trait is numeric and not binary, a linear model is trained. If the user wants to enforce a specific type of model explicitly, he/she can do so by setting the argument `type` to one of the three choices `"linear"`, `"logistic"`, or `"bernoulli"` (see `?nullModel` for details).

An example using only the intercept, but no covariates:

```
nullModel(y ~ 1, pheno.c)

## Linear model:
##  Only intercept (no covariates)
##  Number of samples: 200
##  Variance of residuals: 2.089638
##  No resampling
```

An example in which we want to consider the traits as a Bernoulli-distributed variable:

```
nullModel(y ~ 0, pheno.b, type="bernoulli")

## Simple Bernoulli model:
##  Raw phenotypes (no covariates, no intercept)
##  Number of samples: 200
##  Number of positives (cases): 10
##  No resampling
```

Apart from the formula interface used above, `nullModel()` also allows for supplying a covariate matrix as first argument `X` (optional, omit if no covariates should be considered) and a trait vector as second argument `y`:

```
covX <- as.matrix(pheno.c[, 1:2])
traitY <- pheno.c$y
nullModel(covX, traitY)

## Linear model:
##  Number of covariates: 2 (+ intercept)
##  Number of samples: 200
##  Variance of residuals: 1.541756
##  No resampling

nullModel(y=traitY)

## Linear model:
##  Only intercept (no covariates)
##  Number of samples: 200
##  Variance of residuals: 2.089638
##  No resampling

covX <- as.matrix(pheno.b[, 1:2])
traitY <- pheno.b$y
nullModel(covX, traitY)
```

```
## small sample correction applied

## Logistic model:
##  Number of covariates: 2 (+ intercept)
##  Number of samples: 200
##  Number of positives (cases): 10
##  No resampling
##  Adjustment of higher moments: 10000 repeats (bootstrap)

nullModel(y=traitY)

## small sample correction applied

## Logistic model:
##  Only intercept (no covariates)
##  Number of samples: 200
##  Number of positives (cases): 10
##  No resampling
##  Adjustment of higher moments: 10000 repeats (bootstrap)

nullModel(y=traitY, type="bernoulli")

## Simple Bernoulli model:
##  Raw phenotypes (no covariates, no intercept)
##  Number of samples: 200
##  Number of positives (cases): 10
##  No resampling
```

In the same way this works for many other R functions, it is also possible to attach the data frame with the phenotype data (trait plus covariates) to the global environment. Then it is no longer necessary to the pass the data frame to the `nullModel()` function. However, one has to be more cautious with the selection of the covariates. The option to simply select all covariates with `.` is no longer available then.

```
attach(pheno.c)
nullModel(y ~ X.1 + X.2)

## Linear model:
##  Number of covariates: 2 (+ intercept)
##  Number of samples: 200
##  Variance of residuals: 1.541756
##  No resampling
```

Regardless of the type of model and of which interface has been used to call `nullModel()`, the

function always creates an R object of class `NullModel` (the objects named `model.c` and `model.b` in the examples above) that can be used in subsequent association tests.

Variance-score component tests based on linear logistic models may not necessarily determine the null distribution of the test statistic correctly [8, 10] and, therefore, they may not control the type-I error rate correctly. Following a philosophy inspired by the SKAT package [8, 16] PODKAT offers two means to counteract this issue:

Resampling: under the null assumption that the trait only depends on the covariates (if any) and not on the genotype, a certain number of model residuals are sampled. Then, when association testing is performed, p-values are computed also for all these sampled residuals, and an additional estimated p-value is computed as the relative frequency of p-values of sampled residuals that are at least as significant as the test's p-value. The number of sampled residuals is controlled with the `n.resampling` argument (the default is 0) and the type of sampling procedure is controlled with the `type.resampling` argument (see `?nullModel` for more details).

Small sample correction and adjustment of higher moments: Lee *et al.* [8] proposed a correction of the null distribution for small samples and a sampling method for adjusting higher moments of the null distribution of the test statistic (see also Subsections 9.1 and 9.5). PODKAT implements both corrections (see Subsection 9.5 about implementation details). The argument `adj` controls whether the null model is created such that any of the two corrections can be used later. The default is that the corrections are switched on for samples sizes up to 2,000, while `adj="force"` always turns corrections on and `adj="none"` always turns corrections off. The adjustment of higher moments requires sampled null model residuals. The number of those is controlled with the `n.resampling.adj` argument and the type of sampling procedure is again controlled with the `type.resampling` argument (see `?nullModel` and Subsection 9.5 for more details).

For linear models, there is no need for any correction of the null distribution (cf. Subsection 9.1). Consequently, small sample correction is not available for linear models. Resampling, however, is available for linear models, too. None of the two methods is available for association tests using a Bernoulli-distributed trait.

Some examples showing how to control resampling and small sample corrections for logistic linear models:

```
nullModel(y ~ ., pheno.b, n.resampling=1000, adj="none")

## Logistic model:
##  Number of covariates: 2 (+ intercept)
##  Number of samples: 200
##  Number of positives (cases): 10
##  Resampling: 1000 repeats (bootstrap)

nullModel(y ~ ., pheno.b, n.resampling.adj=2000)

## small sample correction applied
```

```
## Logistic model:
##  Number of covariates: 2 (+ intercept)
##  Number of samples: 200
##  Number of positives (cases): 10
##  No resampling
##  Adjustment of higher moments: 2000 repeats (bootstrap)
```

5 Selection of Regions of Interest

Association tests with PODKAT typically consider multiple regions of interest along the samples' genome. The most common scenarios are whole-genome association testing, whole-exome association testing, or association tests for specific user-defined regions. In the following, we will highlight the basic steps necessary for each of these three scenarios.

5.1 Regions of Interest for Whole-Genome Association Testing

Suppose that the samples' genotypes have been determined by whole-genome sequencing or any other technology that covers variants across the whole genome. The first step for this case is to define the genome and where it has been sequenced. PODKAT comes with four ready-made `GRangesList` objects (see Bioconductor package `GenomicRanges`) that define these regions for autosomal chromosomes, sex chromosomes, and the mitochondrial DNA of the human genome. Those objects are called `hg18Unmasked`, `hg19Unmasked`, `hg38Unmasked`, `b36Unmasked`, and `b37Unmasked`. The three former are the standard hg18, hg19, and hg38 builds as shipped with the Bioconductor packages

- `BSgenome.Hsapiens.UCSC.hg18.masked`,
- `BSgenome.Hsapiens.UCSC.hg19.masked`, and
- `BSgenome.Hsapiens.UCSC.hg38.masked`.

The two latter are basically the same regions as in `hg18Unmasked` and `hg19Unmasked`, but with chromosomes named as in the genomes b36 and b37 that are frequently used by the Genome Analysis Toolkit (GATK).[2] The five objects are available upon `data()` calls as in the following example:

```
data(hg38Unmasked)
hg38Unmasked

## GRangesList object of length 31:
## $chr1
## GRanges object with 15 ranges and 0 metadata columns:
```

[2] `https://www.broadinstitute.org/gatk/` (last visited 2021-04-30)

```
##        seqnames              ranges strand
##           <Rle>           <IRanges>  <Rle>
##    [1]     chr1        10001-207666      *
##    [2]     chr1       257667-297968      *
##    [3]     chr1       347969-535988      *
##    [4]     chr1      585989-2702781      *
##    [5]     chr1    2746291-12954384      *
##    ...      ...                 ...    ...
##   [11]     chr1 125131848-125171347      *
##   [12]     chr1 125173584-125184587      *
##   [13]     chr1 143184588-223558935      *
##   [14]     chr1 223608936-228558364      *
##   [15]     chr1 228608365-248946422      *
##   -------
##   seqinfo: 25 sequences (1 circular) from hg38 genome
##
## ...
## <30 more elements>

names(hg38Unmasked)

##  [1] "chr1"   "chr2"   "chr3"   "chr4"   "chr5"   "chr6"
##  [7] "chr7"   "chr8"   "chr9"   "chr10"  "chr11"  "chr12"
## [13] "chr13"  "chr14"  "chr15"  "chr16"  "chr17"  "chr18"
## [19] "chr19"  "chr20"  "chr21"  "chr22"  "chrX"   "chrY"
## [25] "chrM"   "X.PAR1" "X.PAR2" "X.XTR"  "Y.PAR1" "Y.PAR2"
## [31] "Y.XTR"

hg38Unmasked$chr1

## GRanges object with 15 ranges and 0 metadata columns:
##        seqnames              ranges strand
##           <Rle>           <IRanges>  <Rle>
##    [1]     chr1        10001-207666      *
##    [2]     chr1       257667-297968      *
##    [3]     chr1       347969-535988      *
##    [4]     chr1      585989-2702781      *
##    [5]     chr1    2746291-12954384      *
##    ...      ...                 ...    ...
##   [11]     chr1 125131848-125171347      *
##   [12]     chr1 125173584-125184587      *
##   [13]     chr1 143184588-223558935      *
##   [14]     chr1 223608936-228558364      *
##   [15]     chr1 228608365-248946422      *
##   -------
##   seqinfo: 25 sequences (1 circular) from hg38 genome
```

Table 1: Overview of how the GRangesList objects hg18Unmasked, hg19Unmasked, hg38Unmasked, b36Unmasked, and b37Unmasked are organized: each row corresponds to one chromosome/sequence of the human genome and lists the names of those list components that contain regions from these chromosomes/sequences.

Chromosome	hg*Unmasked	b36Unmasked	b37Unmasked
1 ⋮ 22	"chr1" ⋮ "chr22"	"1" ⋮ "22"	"1" ⋮ "22"
X	"chrX", "X.PAR1", "X.PAR2", "X.XTR"	"X", "X.PAR1", "X.PAR2", "X.XTR"	"X", "X.PAR1", "X.PAR2", "X.XTR"
Y	"chrY", "Y.PAR1", "Y.PAR2", "Y.XTR"	"Y", "Y.PAR1", "Y.PAR2", "Y.XTR"	"Y", "Y.PAR1", "Y.PAR2", "Y.XTR"
mtDNA	"chrM"	"M"	"MT"

```
seqinfo(hg38Unmasked)
```

```
## Seqinfo object with 25 sequences (1 circular) from hg38 genome:
##   seqnames seqlengths isCircular genome
##   chr1      248956422      FALSE   hg38
##   chr2      242193529      FALSE   hg38
##   chr3      198295559      FALSE   hg38
##   chr4      190214555      FALSE   hg38
##   chr5      181538259      FALSE   hg38
##   ...             ...        ...    ...
##   chr21      46709983      FALSE   hg38
##   chr22      50818468      FALSE   hg38
##   chrX      156040895      FALSE   hg38
##   chrY       57227415      FALSE   hg38
##   chrM          16569       TRUE   hg38
```

All four objects are organized in the same way; they consist of 31 components: one for each of the 22 autosomal chromosomes, one for each of the two sex chromosomes, one for the mitochondrial DNA, and two for each of the three pseudoautosomal regions. This structure has been chosen to allow the user to consider different chromosomes and pseudoautosomal regions separately. Table 1 gives an overview of the list components of each of those GRangesList objects, how their list components are named, and how they relate to chromosomes in the human genome.

A simpler structure can be created easily. As an example, the pseudoautosomal regions can be re-united with the X and Y chromosomes as follows:

```
hg38basic <- hg38Unmasked[paste0("chr", 1:22)]
hg38basic$chrX <- reduce(unlist(hg38Unmasked[c("chrX", "X.PAR1",
```

```
                                                    "X.PAR2", "X.XTR")]))
hg38basic$chrY <- reduce(unlist(hg38Unmasked[c("chrY", "Y.PAR1",
                                                    "Y.PAR2", "Y.XTR")]))
hg38basic

## GRangesList object of length 24:
## $chr1
## GRanges object with 15 ranges and 0 metadata columns:
##        seqnames              ranges strand
##           <Rle>           <IRanges>  <Rle>
##    [1]     chr1        10001-207666      *
##    [2]     chr1       257667-297968      *
##    [3]     chr1       347969-535988      *
##    [4]     chr1      585989-2702781      *
##    [5]     chr1    2746291-12954384      *
##    ...      ...                 ...    ...
##   [11]     chr1 125131848-125171347      *
##   [12]     chr1 125173584-125184587      *
##   [13]     chr1 143184588-223558935      *
##   [14]     chr1 223608936-228558364      *
##   [15]     chr1 228608365-248946422      *
##   -------
##   seqinfo: 25 sequences (1 circular) from hg38 genome
##
## ...
## <23 more elements>

names(hg38basic)

##  [1] "chr1"  "chr2"  "chr3"  "chr4"  "chr5"  "chr6"  "chr7"
##  [8] "chr8"  "chr9"  "chr10" "chr11" "chr12" "chr13" "chr14"
## [15] "chr15" "chr16" "chr17" "chr18" "chr19" "chr20" "chr21"
## [22] "chr22" "chrX"  "chrY"
```

If the user prefers to have all unmasked regions in one single `GRanges` object, this can be done as follows:

```
hg38all <- reduce(unlist(hg38Unmasked))
hg38all

## GRanges object with 357 ranges and 0 metadata columns:
##         seqnames            ranges strand
##            <Rle>         <IRanges>  <Rle>
##     [1]     chr1      10001-207666      *
##     [2]     chr1     257667-297968      *
```

```
##     [3]     chr1     347969-535988      *
##     [4]     chr1    585989-2702781      *
##     [5]     chr1  2746291-12954384      *
##     ...      ...                ...    ...
##   [353]     chrY 21750315-21789281      *
##   [354]     chrY 21805282-26673214      *
##   [355]     chrY 56673215-56771509      *
##   [356]     chrY 56821510-57217415      *
##   [357]     chrM           1-16569      *
##   -------
##   seqinfo: 25 sequences (1 circular) from hg38 genome
```

If association testing should be done for any other genome, the user must specify unmasked regions as a `GRanges` or `GRangesList` object first. This can be done manually, but it is more convenient to start from a `MaskedBSgenome` object. Subsection 8.4 provides more details.

It makes little sense to perform association tests for whole chromosomes (or unmasked regions thereof). The most common approach is to split these regions into overlapping windows of (almost) equal lengths. In order to do this conveniently, PODKAT provides the function `partitionRegions()`. A toy example:

```
gr <- GRanges(seqnames="chr1", ranges=IRanges(start=1, end=140000))
partitionRegions(gr, width=10000, overlap=0.5)

## GRanges object with 27 ranges and 0 metadata columns:
##        seqnames        ranges strand
##           <Rle>     <IRanges>  <Rle>
##    [1]     chr1       1-10000      *
##    [2]     chr1    5001-15000      *
##    [3]     chr1   10001-20000      *
##    [4]     chr1   15001-25000      *
##    [5]     chr1   20001-30000      *
##    ...      ...           ...    ...
##   [23]     chr1 110001-120000      *
##   [24]     chr1 115001-125000      *
##   [25]     chr1 120001-130000      *
##   [26]     chr1 125001-135000      *
##   [27]     chr1 130001-140000      *
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths

partitionRegions(gr, width=15000, overlap=0.8)

## GRanges object with 43 ranges and 0 metadata columns:
##        seqnames        ranges strand
##           <Rle>     <IRanges>  <Rle>
```

```
##    [1]     chr1       1-14500      *
##    [2]     chr1    2501-17500      *
##    [3]     chr1    5501-20500      *
##    [4]     chr1    8501-23500      *
##    [5]     chr1   11501-26500      *
##    ...      ...           ...    ...
##   [39]     chr1 113501-128500      *
##   [40]     chr1 116501-131500      *
##   [41]     chr1 119501-134500      *
##   [42]     chr1 122501-137500      *
##   [43]     chr1 125501-140000      *
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths

partitionRegions(gr, width=10000, overlap=0)

## GRanges object with 14 ranges and 0 metadata columns:
##        seqnames        ranges strand
##           <Rle>     <IRanges>  <Rle>
##    [1]     chr1       1-10000      *
##    [2]     chr1   10001-20000      *
##    [3]     chr1   20001-30000      *
##    [4]     chr1   30001-40000      *
##    [5]     chr1   40001-50000      *
##    ...      ...           ...    ...
##   [10]     chr1  90001-100000      *
##   [11]     chr1 100001-110000      *
##   [12]     chr1 110001-120000      *
##   [13]     chr1 120001-130000      *
##   [14]     chr1 130001-140000      *
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Obviously, the `width` argument controls the width of the windows (the default is 5,000) and the `overlap` argument controls the relative overlap (the default is 0.5, which corresponds to 50% overlap). The windows are placed such that possible overhangs are balanced at the beginning and end of the partitioned region.

The choice of the right window width is crucial. If the windows are too narrow, causal regions may be split across multiple windows which may impair statistical power and requires more aggressive multiple testing correction. However, if the windows are too large, associations may be diluted by the large number of variants considered by every single test. We recommend a width between 5,000 bp and 50,000 bp along with 50% overlap.

If called for a `GRanges` object, `partitionRegions()` returns a `GRanges` object with partitioned regions. If called for a `GRangesList` object, `partitionRegions()` returns a `GRangesList`

object, where each component of the output object corresponds to the partitioning of one of the components of the input object.

```
partitionRegions(hg38Unmasked, width=20000)

## GRangesList object of length 31:
## $chr1
## GRanges object with 23041 ranges and 0 metadata columns:
##           seqnames              ranges strand
##              <Rle>           <IRanges>  <Rle>
##       [1]     chr1         10001-28833      *
##       [2]     chr1         18834-38833      *
##       [3]     chr1         28834-48833      *
##       [4]     chr1         38834-58833      *
##       [5]     chr1         48834-68833      *
##       ...      ...                 ...    ...
##   [23037]     chr1 248887394-248907393      *
##   [23038]     chr1 248897394-248917393      *
##   [23039]     chr1 248907394-248927393      *
##   [23040]     chr1 248917394-248937393      *
##   [23041]     chr1 248927394-248946422      *
##   -------
##   seqinfo: 25 sequences (1 circular) from hg38 genome
##
## ...
## <30 more elements>
```

The `partitionRegions()` functions also allows for partitioning only a subset of chromosomes. This can be done by specifying the `chrs` argument, e.g. `chrs="chr22"` only considers regions on chromosome 22 and omits all other regions. This works both for `GRanges` and `GRangesList` objects. However, `partitionRegions()` works for any `GRangesList` object and makes no prior assumption about which chromosomes appear in each of the list components. Technically, this means that all list components will be searched for regions that lie on the specified chromosome(s). The `GRangesList` objects `hg18Unmasked`, `hg19Unmasked`, `hg38Unmasked`, `b36Unmasked`, and `b37Unmasked` included in the PODKAT package, however, are organized that all list components only contain regions from one chromosome (see Table 1). Therefore, it is not necessary to search all list components. The following example does this more efficiently by restricting to chromosomes 21 and 22 from the beginning:

```
partitionRegions(hg38Unmasked[c("chr21", "chr22")], width=20000)

## GRangesList object of length 2:
## $chr21
## GRanges object with 3997 ranges and 0 metadata columns:
##          seqnames            ranges strand
##             <Rle>         <IRanges>  <Rle>
```

```
##      [1]    chr21   5010001-5028123      *
##      [2]    chr21   5018124-5038123      *
##      [3]    chr21   5028124-5048123      *
##      [4]    chr21   5038124-5058123      *
##      [5]    chr21   5048124-5068123      *
##      ...      ...               ...    ...
##   [3993]    chr21 46641223-46661222      *
##   [3994]    chr21 46651223-46671222      *
##   [3995]    chr21 46661223-46681222      *
##   [3996]    chr21 46671223-46691222      *
##   [3997]    chr21 46681223-46699983      *
##   -------
##   seqinfo: 25 sequences (1 circular) from hg38 genome
##
## $chr22
## GRanges object with 3907 ranges and 0 metadata columns:
##          seqnames            ranges strand
##             <Rle>         <IRanges>  <Rle>
##      [1]    chr22 10510001-10527321      *
##      [2]    chr22 10517322-10537321      *
##      [3]    chr22 10527322-10547321      *
##      [4]    chr22 10537322-10557321      *
##      [5]    chr22 10547322-10567321      *
##      ...      ...               ...    ...
##   [3903]    chr22 50751917-50771916      *
##   [3904]    chr22 50761917-50781916      *
##   [3905]    chr22 50771917-50791916      *
##   [3906]    chr22 50781917-50801916      *
##   [3907]    chr22 50791917-50808468      *
##   -------
##   seqinfo: 25 sequences (1 circular) from hg38 genome
```

The following call using the `chrs` argument would give exactly the same result as the command above, but takes approximately 10 times as much time:

```
partitionRegions(hg38Unmasked, chrs=c("chr21", "chr22"), width=20000)

##!!## GRangesList object of length 2:
##!!## $chr21
##!!## GRanges object with 3997 ranges and 0 metadata columns:
##!!##          seqnames               ranges strand
##!!##             <Rle>            <IRanges>  <Rle>
##!!##      [1]    chr21   [5010001, 5028123]      *
##!!##      [2]    chr21   [5018124, 5038123]      *
##!!##      [3]    chr21   [5028124, 5048123]      *
```

```
##!!##      [4]    chr21   [5038124, 5058123]      *
##!!##      [5]    chr21   [5048124, 5068123]      *
##!!##      ...      ...                  ...    ...
##!!##   [3993]    chr21 [46641223, 46661222]      *
##!!##   [3994]    chr21 [46651223, 46671222]      *
##!!##   [3995]    chr21 [46661223, 46681222]      *
##!!##   [3996]    chr21 [46671223, 46691222]      *
##!!##   [3997]    chr21 [46681223, 46699983]      *
##!!##
##!!## ...
##!!## <1 more element>
##!!## -------
##!!## seqinfo: 25 sequences (1 circular) from hg38 genome
```

5.2 Regions of Interest for Whole-Exome Association Testing

Suppose that the samples' genotypes have been determined by whole-exome sequencing. In this case, it makes little sense to use a partition of the whole genome as regions of interest. Instead, the best way is to use exactly those regions that have been targeted by the capturing technology. If these regions are available as a BED file[3], this file can be read with the function `readRegionsFromBedFile()`. In the following example, we demonstrate this for a BED file that specifies the regions targeted by the Illumina® TruSeq DNA Exome Kit. The regions are based on the hg19 human genome build. In order to make this code example work, users must first download the file from the Illumina® website[4]:

```
readRegionsFromBedFile("truseq-exome-targeted-regions-manifest-v1-2.bed")

##!!## GRanges object with 214126 ranges and 0 metadata columns:
##!!#                                seqnames            ranges strand
##!!##                                   <Rle>         <IRanges>  <Rle>
##!!##        CEX-chr1-12099-12258     chr1       12098-12258      *
##!!##        CEX-chr1-12554-12721     chr1       12553-12721      *
##!!##        CEX-chr1-13332-13701     chr1       13331-13701      *
##!!##        CEX-chr1-30335-30503     chr1       30334-30503      *
##!!##        CEX-chr1-35046-35544     chr1       35045-35544      *
##!!##                           ...      ...               ...    ...
##!!##  CEX-chrY-59355682-59355884     chrY 59355681-59355884      *
##!!##  CEX-chrY-59355972-59356131     chrY 59355971-59356131      *
##!!##  CEX-chrY-59356790-59356943     chrY 59356789-59356943      *
##!!##  CEX-chrY-59357687-59357786     chrY 59357686-59357786      *
##!!##  CEX-chrY-59357911-59358045     chrY 59357910-59358045      *
```

[3] `https://genome.ucsc.edu/FAQ/FAQformat.html#format1` (last visited 2021-04-30)

[4] `https://emea.support.illumina.com/downloads/truseq-exome-product-files.html` (last visited: 2021-04-30)

```
##!!##  -------
##!!##  seqinfo: 25 sequences from an unspecified genome; no seqlengths
```

Since a BED file does not contain any genomic annotation, `readRegionsFromBedFile()` is not able to set chromosome names and chromosome lengths properly. In order to overcome this limitation, `readRegionsFromBedFile()` allows for passing a `Seqinfo` object via the `seqInfo` argument. Then the metadata of the returned object are properly set to those passed as `seqInfo` argument:

```
data(hg19Unmasked)
reg <- readRegionsFromBedFile("truseq-exome-targeted-regions-manifest-v1-2.bed",
                              seqInfo=seqinfo(hg19Unmasked))
seqinfo(reg)

##!!## Seqinfo object with 25 sequences from hg19 genome:
##!!##   seqnames seqlengths isCircular genome
##!!##   chr1      249250621       <NA>   hg19
##!!##   chr2      243199373       <NA>   hg19
##!!##   chr3      198022430       <NA>   hg19
##!!##   chr4      191154276       <NA>   hg19
##!!##   chr5      180915260       <NA>   hg19
##!!##   ...             ...        ...    ...
##!!##   chr21      48129895       <NA>   hg19
##!!##   chr22      51304566       <NA>   hg19
##!!##   chrX      155270560       <NA>   hg19
##!!##   chrY       59373566       <NA>   hg19
##!!##   chrM          16571       <NA>   hg19
```

Locations of transcripts can be used as regions of interest, too:

```
library(TxDb.Hsapiens.UCSC.hg38.knownGene)

## Warning:  replacing previous import 'utils::findMatches' by 'S4Vectors::findMatches'
when loading 'AnnotationDbi'

hg38tr <- transcripts(TxDb.Hsapiens.UCSC.hg38.knownGene, columns="tx_name")
hg38tr

## GRanges object with 274031 ranges and 1 metadata column:
##                         seqnames      ranges strand |
##                            <Rle>   <IRanges>  <Rle> |
##        [1]                 chr1 11869-14409      + |
##        [2]                 chr1 12010-13670      + |
##        [3]                 chr1 29554-31097      + |
##        [4]                 chr1 30267-31109      + |
```

```
##         [5]                 chr1 30366-30503      + |
##         ...                  ...          ...    ... .
##    [274027] chr22_KQ759761v1_alt   4297-8606      - |
##    [274028] chr22_KQ759761v1_alt 14178-17674      - |
##    [274029] chr22_KQ759761v1_alt 31687-35861      - |
##    [274030] chr22_KQ759761v1_alt 31687-35861      - |
##    [274031] chr22_KQ759761v1_alt 41305-46422      - |
##                       tx_name
##                   <character>
##         [1] ENST00000456328.2
##         [2] ENST00000450305.2
##         [3] ENST00000473358.1
##         [4] ENST00000469289.1
##         [5] ENST00000607096.1
##         ...                ...
##    [274027] ENST00000637987.1
##    [274028] ENST00000630470.2
##    [274029] ENST00000637440.3
##    [274030] ENST00000643363.2
##    [274031] ENST00000635730.1
##    -------
##    seqinfo: 711 sequences (1 circular) from hg38 genome
```

The `GRanges` object returned by `transcripts()` in the above example code includes strand information. This does no harm to a subsequent association test, since all `assocTest()` methods ignore all information except the chromosome and the start of the region. In any case, the user is advised rather to use exactly those regions that were targeted by the biotechnology that was applied. If this information is not available, we rather recommend to use transcripts, perhaps extended to promotor regions and untranslated regions. Alternatively, to narrow down association analysis by removing introns, it is also possible to use exons only. This can simply be done by replacing the above call to `transcripts()` by `exons()`.

If regions located on sex chromosomes or in pseudo-autosomal regions should be treated differently, the best option is to split the regions object first such that the different regions are grouped together. For convenience, PODKAT provides a `split()` method that allows for splitting a `GRanges` object along grouped regions contained in a `GRangesList` object. The following example splits up transcripts (for the sake of shorter computation times, we restrict to the X chromosome and pseudo-autosomal regions located on the X chromosome):

```
strand(hg38tr) <- "*"
split(hg38tr, hg38Unmasked[c("chrX", "X.PAR1", "X.PAR2", "X.XTR")])

## GRangesList object of length 4:
## $chrX
## GRanges object with 1536 ranges and 0 metadata columns:
##          seqnames              ranges strand
```

```
##             <Rle>           <IRanges>  <Rle>
##      [1]     chrX     2781480-2816500      *
##      [2]     chrX     2828822-2882820      *
##      [3]     chrX     2903972-2929349      *
##      [4]     chrX     2934045-2968475      *
##      [5]     chrX     3006546-3034111      *
##      ...      ...                 ...    ...
##   [1532]     chrX 155380709-155381299      *
##   [1533]     chrX 155382095-155383801      *
##   [1534]     chrX 155456914-155458620      *
##   [1535]     chrX 155459415-155460005      *
##   [1536]     chrX 155466540-155701382      *
##   -------
##   seqinfo: 711 sequences (1 circular) from hg38 genome
##
## ...
## <3 more elements>
```

The `split()` function is strand-specific, that is why we have to discard the strand information in `hg38tr` first (whereas `hg38Unmasked` does not contain any strand information anyway).

The lengths of exons, transcripts, and captured target sequences vary quite a lot. In order to avoid that the results of an association test are biased to the lengths of the regions of interest, we suggest to partition longer exons or transcripts as well. This can be done by simply calling `partitionRegions()` for the `GRanges` objects containing exons or transcripts (the call to `reduce()` removes duplicates and unifies partial overlaps):

```
partitionRegions(reduce(hg38tr))
```

```
## GRanges object with 737745 ranges and 0 metadata columns:
##                         seqnames      ranges strand
##                            <Rle>   <IRanges>  <Rle>
##        [1]                 chr1 11869-16488      *
##        [2]                 chr1 13989-18988      *
##        [3]                 chr1 16489-21488      *
##        [4]                 chr1 18989-23988      *
##        [5]                 chr1 21489-26488      *
##        ...                  ...         ...    ...
##   [737741] chr22_KQ759761v1_alt 10286-10741      *
##   [737742] chr22_KQ759761v1_alt 14178-17674      *
##   [737743] chr22_KQ759761v1_alt 31687-35861      *
##   [737744] chr22_KQ759761v1_alt 41305-45113      *
##   [737745] chr22_KQ759761v1_alt 42614-46422      *
##   -------
##   seqinfo: 711 sequences (1 circular) from hg38 genome
```

5.3 Defining Custom Regions of Interest

If a user is not interested in a genome-wide analysis, he/she might want to restrict to a particular genomic region, for example, a particular gene or set of genes that are likely to be relevant for his/her study. In such a case, there are multiple options to define regions of interest. The simplest, but also most tedious, approach is to enter the regions manually. Let us consider the simple example that we are interested in the two human hemoglobin alpha genes HBA1 and HBA2. If we search for these genes in the UCSC Genome Browser[5] [7], we see that the genes are in the following regions (according to the hg38 human genome build):

```
hbaGenes <- GRanges(seqnames="chr16",
                    ranges=IRanges(start=c(176680, 172847),
                                   end=c(177521, 173710)))
names(hbaGenes) <- c("HBA1", "HBA2")
hbaGenes

## GRanges object with 2 ranges and 0 metadata columns:
##        seqnames        ranges strand
##           <Rle>     <IRanges>  <Rle>
##   HBA1    chr16 176680-177521      *
##   HBA2    chr16 172847-173710      *
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Another variant is to use the `TxDb.Hsapiens.UCSC.hg38.knownGene` package to access the genes' locations. In order to do that, we have to take into account that the corresponding transcripts have the UCSC IDs `uc002cfx.2` and`uc002cfv.4` (which enable us to select the right regions by searching for these IDs in the `tx_name` metadata column):

```
hbaGenes <- hg38tr[which(mcols(hg38tr)$tx_name %in%
                         c("ENST00000320868.9", "ENST00000251595.11"))]
names(hbaGenes) <- c("HBA1", "HBA2")
hbaGenes

## GRanges object with 2 ranges and 1 metadata column:
##        seqnames        ranges strand |            tx_name
##           <Rle>     <IRanges>  <Rle> |        <character>
##   HBA1    chr16 172876-173710      * | ENST00000251595.11
##   HBA2    chr16 176680-177522      * |  ENST00000320868.9
##   -------
##   seqinfo: 711 sequences (1 circular) from hg38 genome
```

The probably most general, most automatic, and most elegant way to determine the genes' locations is via direct access to some biological database. The Bioconductor package `biomaRt`

[5] `http://genome.ucsc.edu/` (last visited 2021-04-30)

facilitates such interfaces. However, this interface returns its results as data frames. So, we have to convert the data to a `GRanges` object ourselves. The hemoglobin alpha example again, this time using `biomaRt`:

```
library(biomaRt)
ensem <- useMart("ensembl")
hsEnsem <- useDataset("hsapiens_gene_ensembl", mart=ensem)
res <- getBM(attributes=c("hgnc_symbol", "chromosome_name",
                          "start_position", "end_position", "ucsc"),
             filters="hgnc_symbol", values=c("HBA1", "HBA2"),
             mart=hsEnsem)
res

##!!##   hgnc_symbol chromosome_name start_position end_position
##!!## 1        HBA1              16         176680       177522
##!!## 2        HBA1              16         176680       177522
##!!## 3        HBA1              16         176680       177522
##!!## 4        HBA1              16         176680       177522
##!!## 5        HBA2              16         172847       173710
##!!## 6        HBA2              16         172847       173710
##!!## 7        HBA2              16         172847       173710
##!!## 8        HBA2              16         172847       173710
##!!##         ucsc
##!!## 1 uc002cfx.2
##!!## 2 uc059ohb.1
##!!## 3 uc059ohc.1
##!!## 4 uc059ohd.1
##!!## 5 uc002cfv.4
##!!## 6 uc059ogy.1
##!!## 7 uc059ogz.1
##!!## 8 uc059oha.1

hbaGenes <- GRanges(seqnames=paste0("chr", res$chromosome_name),
                    ranges=IRanges(start=res$start_position,
                                   end=res$end_position))
names(hbaGenes) <- res$hgnc_symbol
hbaGenes

##!!## GRanges object with 8 ranges and 0 metadata columns:
##!!##        seqnames           ranges strand
##!!##           <Rle>        <IRanges>  <Rle>
##!!##   HBA1    chr16 [176680, 177522]      *
##!!##   HBA1    chr16 [176680, 177522]      *
##!!##   HBA1    chr16 [176680, 177522]      *
##!!##   HBA1    chr16 [176680, 177522]      *
##!!##   HBA2    chr16 [172847, 173710]      *
```

```
##!!##   HBA2    chr16 [172847, 173710]      *
##!!##   HBA2    chr16 [172847, 173710]      *
##!!##   HBA2    chr16 [172847, 173710]      *
##!!##   -------
##!!##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

As already mentioned at the end of Subsection 5.2, the lengths of exons and transcripts vary quite a lot. So it is recommended to partition even custom-defined regions of interest using `partitionRegions()` if the regions' lengths differ strongly and/or if the regions are longer than the window size one would usually employ for whole-genome studies.

6 Performing an Association Test

We have already seen in Section 3 that the function for performing the actual association tests is `assocTest()`. We have also seen that there are basically two ways how to use this function. The simpler one is to load the genotype data into a matrix-like object first and to perform an association test on this matrix-like object. To first load the genotype, the following command can be used:

```
geno <- readGenotypeMatrix(vcfFile)
geno

## Genotype matrix:
##  Number of samples: 200
##  Number of variants: 962
##
##  Mean MAF:    0.05674116
##  Median MAF:  0.0075
##  Minimum MAF: 0.0025
##  Maximum MAF: 0.455
```

The object returned by `readGenotypeMatrix()` is of class `GenotypeMatrix`. This class is defined by PODKAT and essentially consists of the genotypes in column-oriented sparse matrix format (with rows corresponding to samples and columns corresponding to variants) along with information about the genomic positions of the variants. The `readGenotypeMatrix()` function has some additional arguments for controlling how variants are pre-processed and filtered (see `?readGenotypeMatrix` and Subsection 8.2 for more details). Information about the genomic positions of the variants can be obtained as follows:

```
variantInfo(geno)

## VariantInfo object with 962 ranges and 1 metadata column:
##            seqnames    ranges strand |       MAF
##               <Rle> <IRanges>  <Rle> | <numeric>
##      snv:6     chr1       428      * |    0.1025
```

```
##      snv:7     chr1       501      * |    0.0900
##      snv:9     chr1       607      * |    0.0050
##     snv:11     chr1       739      * |    0.0025
##     snv:12     chr1       808      * |    0.0025
##        ...      ...       ...    ... .       ...
##   snv:3838     chr1    199637      * |    0.0175
##   snv:3840     chr1    199676      * |    0.2500
##   snv:3842     chr1    199696      * |    0.0025
##   snv:3843     chr1    199812      * |    0.0025
##   snv:3844     chr1    199879      * |    0.0025
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

Obviously, this is a `GRanges` object that also contains a metadata column with minor allele frequencies (MAFs). For convenience, there is a separate accessor function `MAF()` for retrieving these MAFs:

```
summary(MAF(geno))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## 0.00250 0.00250 0.00750 0.05674 0.05500 0.45500
```

Moreover, there is a custom variant of the `summary()` method:

```
summary(variantInfo(geno), details=TRUE)
```

```
## Variant info:
##  Number of variants: 962
##
##  Mean MAF:    0.05674116
##  Median MAF:  0.0075
##  Minimum MAF: 0.0025
##  Maximum MAF: 0.455
##
##  no metadata column 'type' available
```

The simplest approach is to perform a test for the association between the null model the entire genotype:

```
assocTest(geno, model.c)
```

```
## Association test results:
##  Null model: linear
##  Number of samples: 200
```

```
##  Number of variants: 962
##  Kernel: linear.podkat
##  Test statistic: 3034597
##  p-value: 0.05875229

assocTest(geno, model.b)

## Association test results:
##  Null model: logistic
##  Number of samples: 200
##  Number of variants: 962
##  Kernel: linear.podkat
##  Test statistic: 257220.9
##  p-value: 0.01109616
##  (small sample correction + correction for higher moments applied)
```

As already mentioned in Section 5, it is not necessarily the best idea to consider associations between the null model and all variants of the whole genome at once. The larger the number of variants is that are considered at once, the smaller is the ratio of potentially associated variants. Thus, it may become harder for the test to disentangle random effects from true associations. Therefore, the better and more common approach is to split the genome into a set of (overlapping) windows/regions of interest as described in Section 5 — hoping that potentially causal variants will accumulate in certain regions and, thereby, lead to highly significant p-values for these regions. If we already have a certain set of regions of interest, we can simply pass them to `assocTest()` as third argument:

```
res.c <- assocTest(geno, model.c, windows)
print(res.c)

## Overview of association test:
##  Null model: linear
##  Number of samples: 200
##  Number of regions: 79
##  Number of regions without variants: 0
##  Average number of variants in regions: 24.1
##  Genome: hgA
##  Kernel: linear.podkat
##  p-value adjustment: none
##
## Overview of significance of results:
##  Number of tests with p < 0.05: 8
##
## Results for the 8 most significant regions:
##   seqnames  start    end width  n         Q      p.value
## 1     chr1   7501  12500  5000 31 769748.34 1.294084e-07
```

```
## 2     chr1  10001  15000  5000 33 764828.81 4.874460e-06
## 3     chr1 140001 145000  5000 15  79937.68 3.599077e-03
## 4     chr1   5001  10000  5000 34 152555.30 9.785569e-03
## 5     chr1 132501 137500  5000 21  89287.55 1.349559e-02
## 6     chr1 142501 147500  5000 23  94629.68 3.338620e-02
## 7     chr1  42501  47500  5000 19  58191.23 3.341032e-02
## 8     chr1  25001  30000  5000 23 103713.12 3.754557e-02
```

Obviously, `assocTest()` performs the association test for each region independently and computes a p-value for each region.

The `assocTest()` as used above has a few arguments that influence the way the tests are performed. Most importantly, the `kernel` argument allows for choosing among 6 different kernels. Details about these kernels are available in Subsection 9.2. We suggest the default setting `"linear.podkat"`, i.e. the position-dependent linear kernel, PODKAT's most important contribution and achievement. For comparison purposes, PODKAT also provides an up-to-date implementation of the SKAT test [16] which can be chosen by setting `kernel="linear.SKAT"`. The four other kernels have not turned out to be advantageous in simulations, moreover, they require much longer computation times. Anyway, they are available and ready to be used.

Another important decision is the choice of a weighting schemes, i.e. whether and how to choose weights for variants depending on their minor allele frequency (MAF). Details are provided in Subsection 9.3.

If the genotype matrix is too large to fit into the computer's main memory or if parallelization is desired, the alternative, as already mentioned, is to call `assocTest()` for a VCF file name. Then `assocTest()` splits the regions of interest into batches and only loads those variants from the VCF file at once (see Subsection 8.5.1). If called for a VCF file name (or `TabixFile` object), the same arguments for controlling how variants are pre-processed and filtered are available as for the `readGenotypeMatrix()` function (see `?assocTest`, `?readGenotypeMatrix`, and Subsection 8.2 of this manual for more details). This interface also allows for carrying out these computations on multiple processor cores and/or on a computing cluster (see Subsection 8.5.2). We first provide a simple example here without any parallelization:

```
res.c <- assocTest(vcfFile, model.c, windows)
print(res.c)

## Overview of association test:
##  Null model: linear
##  Number of samples: 200
##  Number of regions: 79
##  Number of regions without variants: 0
##  Average number of variants in regions: 24.1
##  Genome: hgA
##  Kernel: linear.podkat
##  p-value adjustment: none
##
## Overview of significance of results:
```

```
##  Number of tests with p < 0.05: 8
##
## Results for the 8 most significant regions:
##   seqnames  start    end width  n         Q      p.value
## 1     chr1   7501  12500  5000 31 769748.34 1.294084e-07
## 2     chr1  10001  15000  5000 33 764828.81 4.874460e-06
## 3     chr1 140001 145000  5000 15  79937.68 3.599077e-03
## 4     chr1   5001  10000  5000 34 152555.30 9.785569e-03
## 5     chr1 132501 137500  5000 21  89287.55 1.349559e-02
## 6     chr1 142501 147500  5000 23  94629.68 3.338620e-02
## 7     chr1  42501  47500  5000 19  58191.23 3.341032e-02
## 8     chr1  25001  30000  5000 23 103713.12 3.754557e-02
```

The above steps can be carried out for binary traits as well. For brevity, we only show the variant with the genotype matrix here:

```
res.b <- assocTest(geno, model.b, windows)
print(res.b)
```

```
## Overview of association test:
##  Null model: logistic
##  Number of samples: 200
##  Number of regions: 79
##  Number of regions without variants: 0
##  Average number of variants in regions: 24.1
##  Genome: hgA
##  Kernel: linear.podkat
##  p-value adjustment: none
##
## Overview of significance of results:
##  Number of tests with p < 0.05: 23
##
## Results for the 10 most significant regions:
##    seqnames start   end width  n        Q      p.value
## 1      chr1  7501 12500  5000 31 38386.55 7.490309e-05
## 2      chr1 10001 15000  5000 33 43084.90 8.888145e-05
## 3      chr1 22501 27500  5000 27 25640.34 6.978189e-04
## 4      chr1 20001 25000  5000 23 16120.63 2.010308e-03
## 5      chr1 25001 30000  5000 23 10650.48 3.816241e-03
## 6      chr1 77501 82500  5000 17  4890.26 6.730915e-03
## 7      chr1 87501 92500  5000 30 12891.61 8.577176e-03
## 8      chr1 35001 40000  5000 23 22436.29 8.610353e-03
## 9      chr1 27501 32500  5000 27 32423.04 9.666232e-03
## 10     chr1 37501 42500  5000 25 22570.67 9.669373e-03
```

So, it seems as if the computations were completely analogous to continuous traits. However,

there is an important aspect that indeed makes a difference: small sample adjustment. As the reader may have noticed above, the null model `model.b` includes 10,000 resampled residuals for correction of higher moments of the null distribution. This correction has actually been performed by the above association test. Let us shortly run the association test without any correction:

```
res.b.noAdj <- assocTest(geno, model.b, windows, adj="none")
print(res.b.noAdj)

## Overview of association test:
##  Null model: logistic
##  Number of samples: 200
##  Number of regions: 79
##  Number of regions without variants: 0
##  Average number of variants in regions: 24.1
##  Genome: hgA
##  Kernel: linear.podkat
##  p-value adjustment: none
##
## Overview of significance of results:
##  Number of tests with p < 0.05: 27
##
## Results for the 10 most significant regions:
##    seqnames start   end width  n        Q      p.value
## 1      chr1  7501 12500  5000 31 38386.55 1.142026e-06
## 2      chr1 22501 27500  5000 27 25640.34 2.299380e-06
## 3      chr1 10001 15000  5000 33 43084.90 2.878117e-06
## 4      chr1 20001 25000  5000 23 16120.63 6.565331e-05
## 5      chr1 27501 32500  5000 27 32423.04 1.818834e-04
## 6      chr1 30001 35000  5000 32 32024.75 2.270615e-04
## 7      chr1 25001 30000  5000 23 10650.48 3.966917e-04
## 8      chr1 77501 82500  5000 17  4890.26 4.788232e-04
## 9      chr1 87501 92500  5000 30 12891.61 1.809554e-03
## 10     chr1 35001 40000  5000 23 22436.29 3.330749e-03
```

Obviously, the p-values seem to have become more significant. Actually, this is not the case, but only the result of a too crude approximation of the null distribution for smaller sample sizes. So, in a case like this one, small sample adjustment and correction of higher moments are essential.

7 Analyzing and Visualizing Results

7.1 Multiple Testing Correction

As soon as multiple tests are performed in parallel, the tests' raw p-values do not allow for a correct assessment of the overall type I error rate anymore and multiple testing correction must be employed. PODKAT provides a simple method for multiple testing correction that is a wrapper

around the R standard function `p.adjust()` from the `stats` package. If called for an object of class `AssocTestResultRanges` (the class of objects the `assocTest()` function creates when called for multiple regions), `p.adjust()` adds a metadata column named `p.value.adj` with adjusted p-values:

```
print(p.adjust(res.c))

## Overview of association test:
##  Null model: linear
##  Number of samples: 200
##  Number of regions: 79
##  Number of regions without variants: 0
##  Average number of variants in regions: 24.1
##  Genome: hgA
##  Kernel: linear.podkat
##  p-value adjustment: holm
##
## Overview of significance of results:
##  Number of tests with p < 0.05: 8
##  Number of tests with adj. p < 0.05: 2
##
## Results for the 8 most significant regions:
##   seqnames  start    end width  n         Q      p.value
## 1     chr1   7501  12500  5000 31 769748.34 1.294084e-07
## 2     chr1  10001  15000  5000 33 764828.81 4.874460e-06
## 3     chr1 140001 145000  5000 15  79937.68 3.599077e-03
## 4     chr1   5001  10000  5000 34 152555.30 9.785569e-03
## 5     chr1 132501 137500  5000 21  89287.55 1.349559e-02
## 6     chr1 142501 147500  5000 23  94629.68 3.338620e-02
## 7     chr1  42501  47500  5000 19  58191.23 3.341032e-02
## 8     chr1  25001  30000  5000 23 103713.12 3.754557e-02
##    p.value.adj
## 1 1.022327e-05
## 2 3.802079e-04
## 3 2.771289e-01
## 4 7.437033e-01
## 5 1.000000e+00
## 6 1.000000e+00
## 7 1.000000e+00
## 8 1.000000e+00
```

For consistency with the standard `p.adjust()` function, the default correction procedure is `"holm"`, which corresponds to the Holm-Bonferroni method for controlling the familywise error rate (FWER) [6]. If a different method is desired, e.g. the popular Benjamini-Hochberg false discovery rate (FDR) correction [2], the method argument must be used to select the desired method:

```
res.c.adj <- p.adjust(res.c, method="BH")
print(res.c.adj)

## Overview of association test:
##  Null model: linear
##  Number of samples: 200
##  Number of regions: 79
##  Number of regions without variants: 0
##  Average number of variants in regions: 24.1
##  Genome: hgA
##  Kernel: linear.podkat
##  p-value adjustment: BH
##
## Overview of significance of results:
##  Number of tests with p < 0.05: 8
##  Number of tests with adj. p < 0.05: 2
##
## Results for the 8 most significant regions:
##   seqnames  start    end width  n         Q      p.value
## 1     chr1   7501  12500  5000 31 769748.34 1.294084e-07
## 2     chr1  10001  15000  5000 33 764828.81 4.874460e-06
## 3     chr1 140001 145000  5000 15  79937.68 3.599077e-03
## 4     chr1   5001  10000  5000 34 152555.30 9.785569e-03
## 5     chr1 132501 137500  5000 21  89287.55 1.349559e-02
## 6     chr1 142501 147500  5000 23  94629.68 3.338620e-02
## 7     chr1  42501  47500  5000 19  58191.23 3.341032e-02
## 8     chr1  25001  30000  5000 23 103713.12 3.754557e-02
##    p.value.adj
## 1 1.022327e-05
## 2 1.925412e-04
## 3 9.477570e-02
## 4 1.932650e-01
## 5 2.132303e-01
## 6 3.707625e-01
## 7 3.707625e-01
## 8 3.707625e-01

res.b.adj <- p.adjust(res.b, method="BH")
print(res.b.adj)

## Overview of association test:
##  Null model: logistic
##  Number of samples: 200
##  Number of regions: 79
##  Number of regions without variants: 0
```

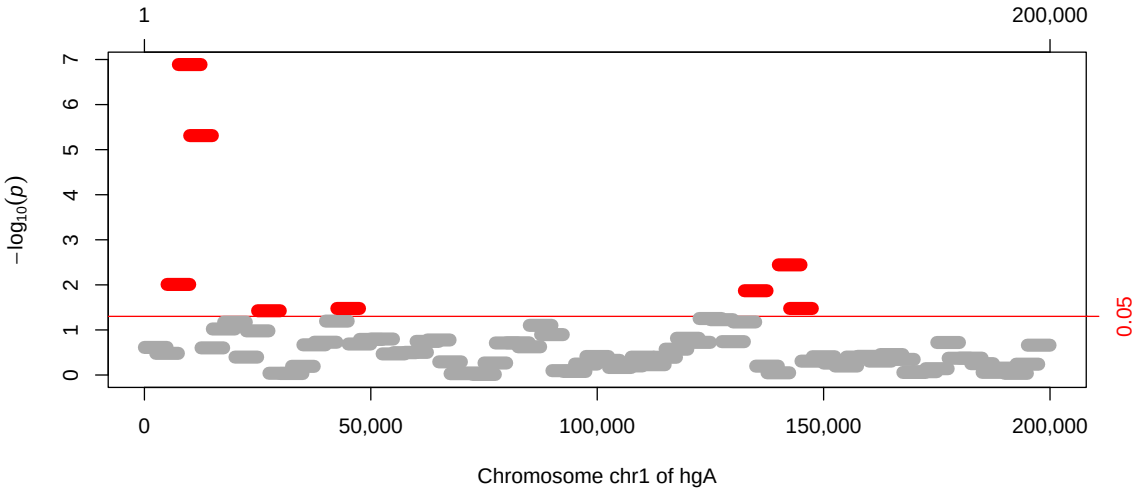
```
##  Average number of variants in regions: 24.1
##  Genome: hgA
##  Kernel: linear.podkat
##  p-value adjustment: BH
##
## Overview of significance of results:
##  Number of tests with p < 0.05: 23
##  Number of tests with adj. p < 0.05: 4
##
## Results for the 10 most significant regions:
##    seqnames start   end width  n        Q      p.value
## 1      chr1  7501 12500  5000 31 38386.55 7.490309e-05
## 2      chr1 10001 15000  5000 33 43084.90 8.888145e-05
## 3      chr1 22501 27500  5000 27 25640.34 6.978189e-04
## 4      chr1 20001 25000  5000 23 16120.63 2.010308e-03
## 5      chr1 25001 30000  5000 23 10650.48 3.816241e-03
## 6      chr1 77501 82500  5000 17  4890.26 6.730915e-03
## 7      chr1 87501 92500  5000 30 12891.61 8.577176e-03
## 8      chr1 35001 40000  5000 23 22436.29 8.610353e-03
## 9      chr1 27501 32500  5000 27 32423.04 9.666232e-03
## 10     chr1 37501 42500  5000 25 22570.67 9.669373e-03
##    p.value.adj
## 1  0.003510817
## 2  0.003510817
## 3  0.018375898
## 4  0.039703590
## 5  0.060296605
## 6  0.073175318
## 7  0.073175318
## 8  0.073175318
## 9  0.073175318
## 10 0.073175318
```

The user has to make sure to choose a correction method the assumptions of which are fulfilled by the given setup. Note that the tests performed by PODKAT are usually not independent of each other, at least not if overlapping windows and/or windows close to each other are tested.

7.2 Visualization

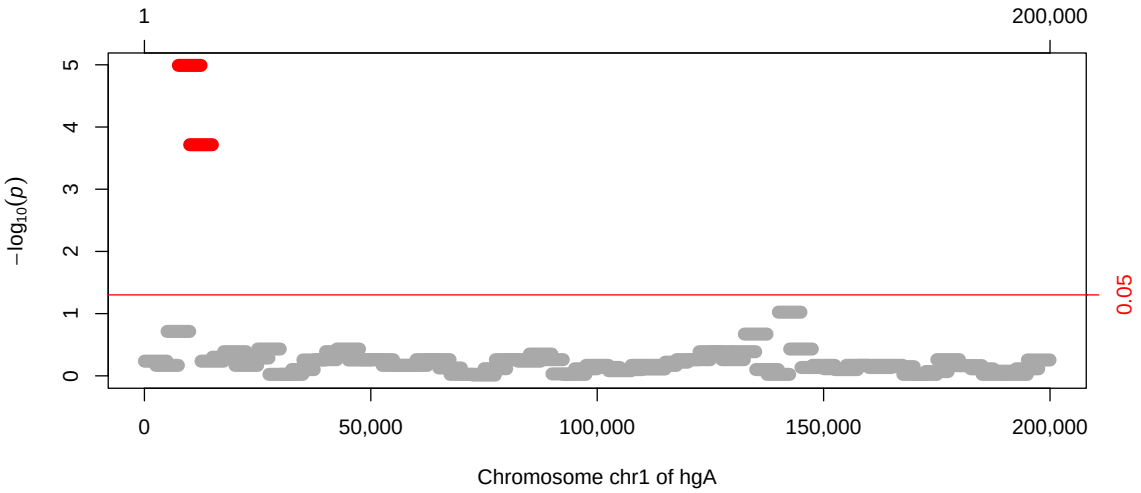
PODKAT also offers functions for visualizing the results of association tests. Suppose we have carried out an association test involving multiple regions, e.g. a whole-genome or whole-exome association test. In such cases, the `assocTest()` returns an `AssocTestResultRanges` object that contains a metadata column with p-values and, if multiple testing correction has been employed (see Subsection 7.1 above), another metadata column with adjusted p-values. If the R standard generic function `plot()` is called on such an object, a so-called *Manhattan plot* is produced, that is, log-transformed p-values are plotted along the genome:

```
plot(res.c.adj)
```



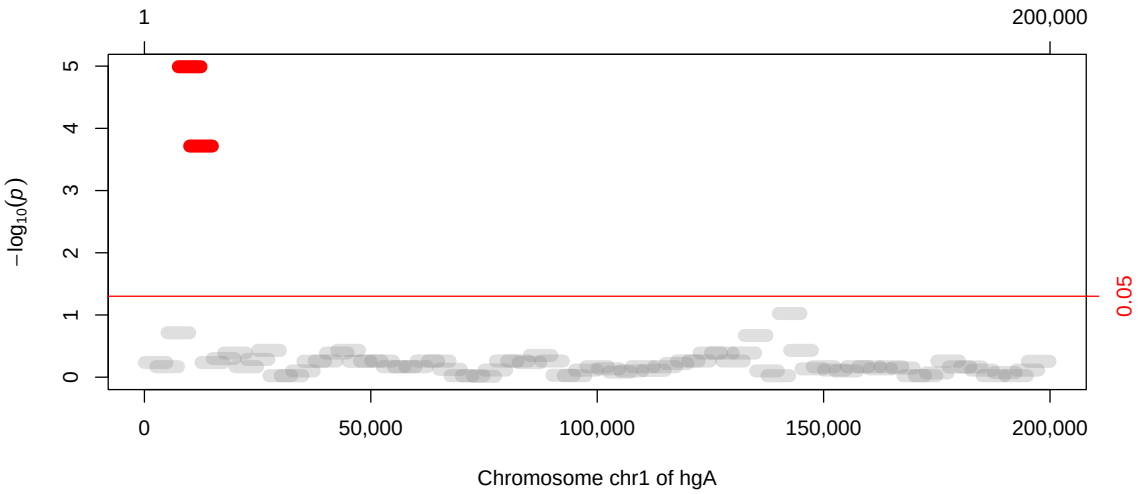
Obviously, the function plots raw (uncorrected) p-values, where the ones passing the significance threshold are plotted in red and the insignificant ones are plotted in gray. In order to correctly account for multiple testing, we would either have to choose a stricter threshold or use adjusted p-values instead. The latter can be accomplished by choosing a different p-value column for plotting:

```
plot(res.c.adj, which="p.value.adj")
```



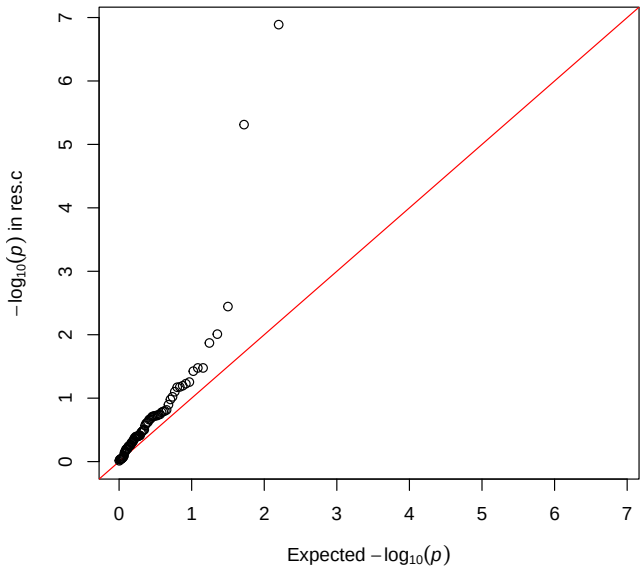
If the genome consists of multiple chromosomes, the default is to plot the insignificant p-values in alternating colors (gray/light gray by default). If very many regions have been tested, in particular in whole-genome studies, it is advisable to use semi-transparent colors (using the alpha channel of functions like `gray()`, `rgb()`, or `hsv()`) to get an impression of the density of p-values. Although this simple example does not necessitate this technique, we show an example in order to demonstrate how it works:

```
plot(res.c.adj, which="p.value.adj", col=gray(0.5, alpha=0.25))
```



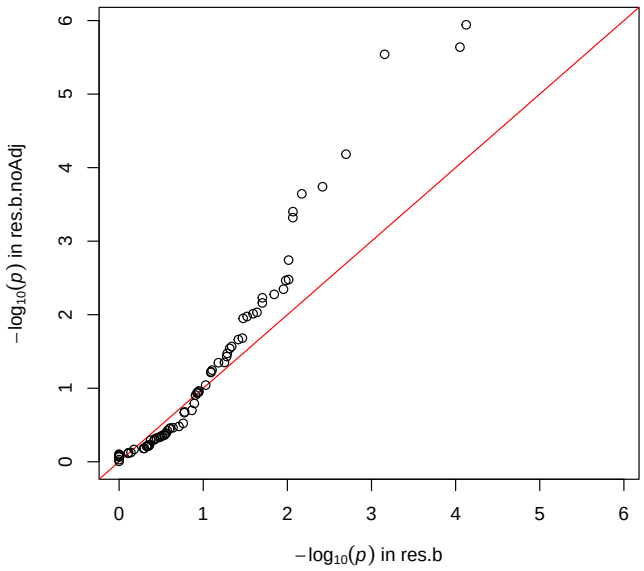
PODKAT further provides a function for making quantile-quantile (Q-Q) plots. If called for a single `AssocTestResultRanges` object, the function `qqplot()` plots log-transformed p-values against a uniform distribution of p-values (which one would expect under the null hypothesis):

```
qqplot(res.c)
```



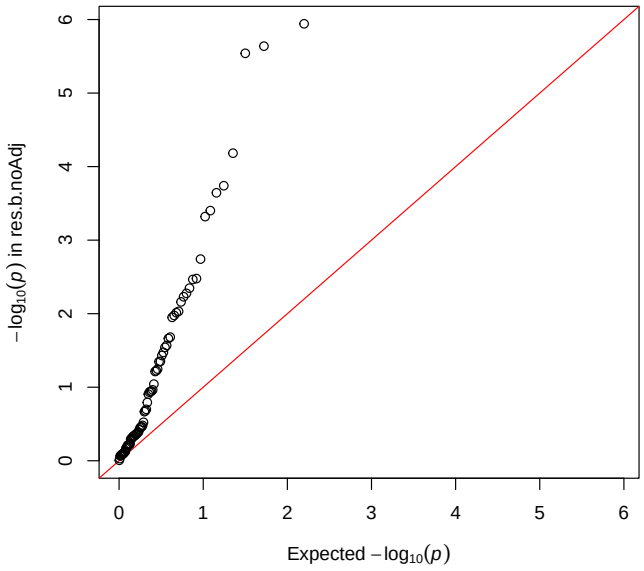
The same function can also be used to compare two association test results in terms of their distributions of p-values, e.g. to compare two different kernels or to compare results with and without small-sample correction:

```
qqplot(res.b, res.b.noAdj)
```



For the above example, we see that the p-values without small-sample correction are supposedly more significant. The reason is that, in this example, the p-values without correction are actually systematically inflated:

```
qqplot(res.b.noAdj)
```



7.3 Filtering Significant Regions

PODKAT offers a simple method for stripping off all insignificant results from an association test result. The method is called `filterResult()` and can be applied to association test results given as objects of class `AssocTestResultRanges`. The user can choose the significance threshold and which p-value column the filter should be applied to. The result is a subset of the input object consisting of those regions the p-value of which passed the threshold.

```
res.c.f <- filterResult(res.c, cutoff=1.e-6)
print(res.c.f, cutoff=1.e-6)

## Overview of association test:
##  Null model: linear
##  Number of samples: 200
##  Number of regions: 1
##  Number of regions without variants: 0
##  Average number of variants in regions: 31.0
##  Genome: hgA
##  Kernel: linear.podkat
```

```
##  p-value adjustment: none
##
## Overview of significance of results:
##  Number of tests with p < 1e-06: 1
##
## Results for the 1 most significant regions:
##   seqnames start   end width  n        Q      p.value
## 1     chr1  7501 12500  5000 31 769748.3 1.294084e-07

res.c.adj.f <- filterResult(res.c.adj, filterBy="p.value.adj")
print(res.c.adj.f)

## Overview of association test:
##  Null model: linear
##  Number of samples: 200
##  Number of regions: 2
##  Number of regions without variants: 0
##  Average number of variants in regions: 32.0
##  Genome: hgA
##  Kernel: linear.podkat
##  p-value adjustment: BH
##
## Overview of significance of results:
##  Number of tests with p < 0.05: 2
##  Number of tests with adj. p < 0.05: 2
##
## Results for the 2 most significant regions:
##   seqnames start   end width  n        Q      p.value
## 1     chr1  7501 12500  5000 31 769748.3 1.294084e-07
## 2     chr1 10001 15000  5000 33 764828.8 4.874460e-06
##    p.value.adj
## 1 1.022327e-05
## 2 1.925412e-04
```

7.4 Contributions of Individual Variants

The association tests provided by PODKAT do not test single-locus variants, but consider multiple variants located in the same genomic window simultaneously, i.e. multiple variants are "collapsed" into a single score and tested together. As a consequence, PODKAT provides association test results per window, but does not allow for pinpointing which individual variants may have made a major contribution to the test's outcome. For linear kernels (choices `kernel="linear.podkat"` and `kernel="linear.SKAT"`), PODKAT offers a method named `weights()` that allows for computing the individual contribution that individual variants made to the outcome of a test. It should be clear that this makes little sense for non-significant windows. Hence, this method should be applied to a filtered result.

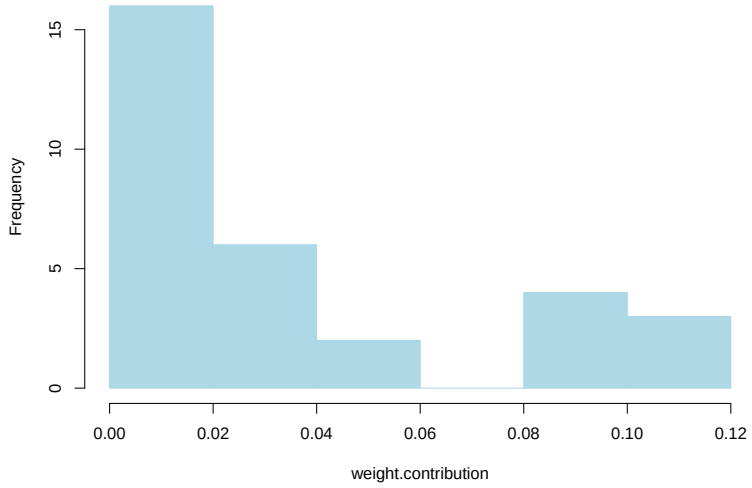
```
w.res.c.adj <- weights(res.c.adj.f, geno, model.c)
w.res.c.adj

## GRangesList object of length 2:
## $`chr1:7501-12500`
## GRanges object with 31 ranges and 2 metadata columns:
##           seqnames    ranges strand | weight.raw
##              <Rle> <IRanges>  <Rle> |  <numeric>
##   snv:160     chr1      7713      * |   -44.4915
##   snv:164     chr1      7834      * |   -40.8922
##   snv:166     chr1      7932      * |   -43.5583
##   snv:167     chr1      7976      * |   -44.3762
##   snv:177     chr1      8342      * |   -56.9144
##       ...      ...       ...    ... .        ...
##   snv:249     chr1     11888      * |    324.931
##   snv:256     chr1     12191      * |    189.297
##   snv:258     chr1     12273      * |    161.128
##   snv:261     chr1     12307      * |    144.859
##   snv:264     chr1     12369      * |    115.202
##           weight.contribution
##                     <numeric>
##   snv:160         0.000833988
##   snv:164         0.000704510
##   snv:166         0.000799370
##   snv:167         0.000829671
##   snv:177         0.001364738
##       ...                 ...
##   snv:249          0.04448243
##   snv:256          0.01509710
##   snv:258          0.01093828
##   snv:261          0.00884089
##   snv:264          0.00559144
##   -------
##   seqinfo: 1 sequence from hgA genome
##
## $`chr1:10001-15000`
## GRanges object with 33 ranges and 2 metadata columns:
##           seqnames    ranges strand | weight.raw
##              <Rle> <IRanges>  <Rle> |  <numeric>
##   snv:223     chr1     10665      * |    211.879
##   snv:227     chr1     10800      * |    277.783
##   snv:233     chr1     11173      * |    441.157
##   snv:234     chr1     11242      * |    460.515
##   snv:237     chr1     11387      * |    494.788
##       ...      ...       ...    ... .        ...
##   snv:311     chr1     14701      * |   -213.368
```

```
##   snv:313     chr1     14800      * |   -196.870
##   snv:314     chr1     14821      * |   -191.199
##   snv:315     chr1     14831      * |   -188.138
##   snv:318     chr1     14974      * |   -135.752
##           weight.contribution
##                     <numeric>
##   snv:223           0.0190356
##   snv:227           0.0327191
##   snv:233           0.0825231
##   snv:234           0.0899245
##   snv:237           0.1038073
##       ...                 ...
##   snv:311          0.01930411
##   snv:313          0.01643423
##   snv:314          0.01550112
##   snv:315          0.01500875
##   snv:318          0.00781414
##   -------
##   seqinfo: 1 sequence from hgA genome
```

Obviously, `weights()` returns a `GRangesList` object with as many components as the first argument has regions. Each of the list components is a `GRanges` object with two metadata columns `weight.raw` and `weight.contribution`. The former corresponds to the raw contributions of the variants. For each variant, the corresponding entry in the column `weight.raw` is positive if the variant is positively associated with the residual/trait. It is negative if the variant is negatively associated with the residual/trait. The value is around zero if there is (almost) no association. The absolute value gives an indication about the magnitude of contribution to the test's statistic. The metadata column `weight.contribution` corresponds to the relative contribution of each variant. It is nothing else but the squares of the `weight.raw` column, but normalized to a sum of 1. As an example, a value of 0.2 means that this variant contributed 20% to the test statistic of this region. Subsection 7.4 provides more details about the contributions are computed.

As an example, we plot a histogram of relative contributions of variants of the first region:

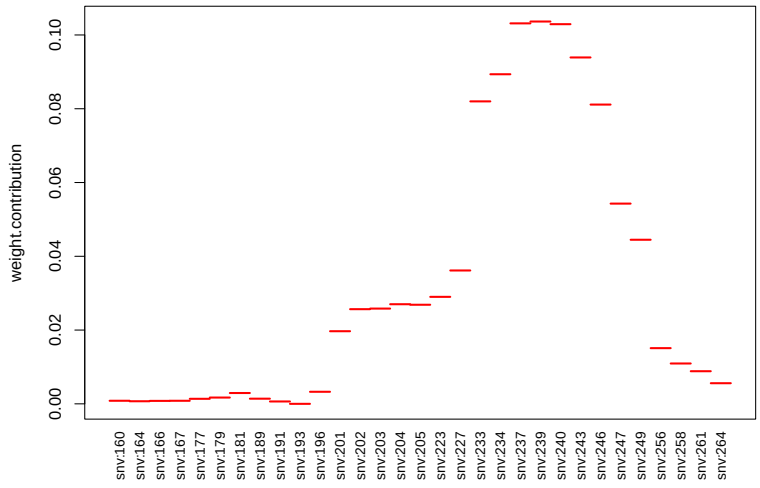
```
hist(mcols(w.res.c.adj[[1]])$weight.contribution, col="lightblue",
     border="lightblue", xlab="weight.contribution", main="")
```



The histogram shows a bimodal distribution which indicates that the major contributions are made by only a few variants, whereas all others only make a minor contribution.

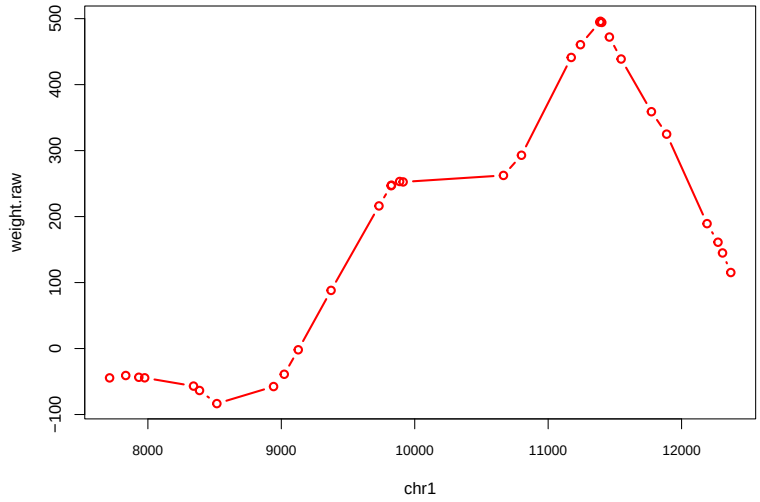
The PODKAT package provides a method for plotting numerical metadata columns of `GRanges` objects. This method can be used to plot the contributions of individual variants in a window:

```
plot(w.res.c.adj[[1]], "weight.contribution")
```



In the above plot, each variant is visualized as an equally large bar/interval along the horizontal axis. Alternatively, the function can also plot contributions (or any other numerical metadata column) along the genome. In the following plot, the type `"b"` has been chosen with the aim to indicate the positions of variants:

```
plot(w.res.c.adj[[1]], "weight.raw", alongGenome=TRUE, type="b")
```



In order to allow the user to easily find the most indicative variant, the `filterResult()` method can be applied to weights, too. If called for `GRanges` or `GRangesList` objects, the function `filterResult()` checks if a metadata column `weight.contribution` is available and strips off all variant with a relative contribution lower than the given threshold `cutoff` (the default is 0.1):

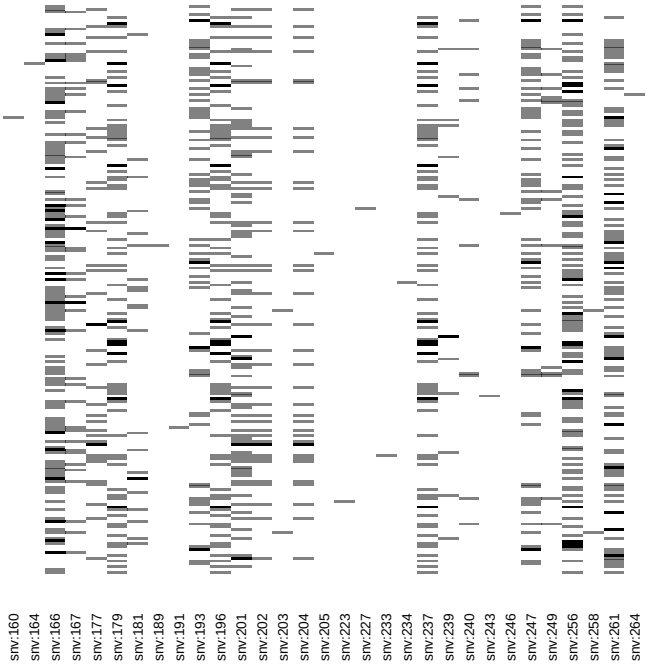
```
filterResult(w.res.c.adj, cutoff=0.07)

## GRangesList object of length 2:
## $`chr1:7501-12500`
## GRanges object with 7 ranges and 2 metadata columns:
##           seqnames    ranges strand | weight.raw
##              <Rle> <IRanges>  <Rle> |  <numeric>
##   snv:233     chr1     11173      * |    441.157
##   snv:234     chr1     11242      * |    460.515
##   snv:237     chr1     11387      * |    494.788
##   snv:239     chr1     11392      * |    495.984
##   snv:240     chr1     11402      * |    494.259
##   snv:243     chr1     11459      * |    472.102
##   snv:246     chr1     11547      * |    438.813
##           weight.contribution
##                     <numeric>
##   snv:233           0.0819957
##   snv:234           0.0893498
##   snv:237           0.1031439
##   snv:239           0.1036431
##   snv:240           0.1029235
```

```
##   snv:243           0.0939027
##   snv:246           0.0811269
##   -------
##   seqinfo: 1 sequence from hgA genome
##
## $`chr1:10001-15000`
## GRanges object with 7 ranges and 2 metadata columns:
##           seqnames    ranges strand | weight.raw
##              <Rle> <IRanges>  <Rle> |  <numeric>
##   snv:233     chr1     11173      * |    441.157
##   snv:234     chr1     11242      * |    460.515
##   snv:237     chr1     11387      * |    494.788
##   snv:239     chr1     11392      * |    495.984
##   snv:240     chr1     11402      * |    494.259
##   snv:243     chr1     11459      * |    472.102
##   snv:246     chr1     11547      * |    438.813
##           weight.contribution
##                     <numeric>
##   snv:233           0.0825231
##   snv:234           0.0899245
##   snv:237           0.1038073
##   snv:239           0.1043098
##   snv:240           0.1035856
##   snv:243           0.0945067
##   snv:246           0.0816488
##   -------
##   seqinfo: 1 sequence from hgA genome
```

That the same variants stand out twice in the above result is neither an error nor a coincidence: the most indicative variants of both windows appear are located in their overlap.

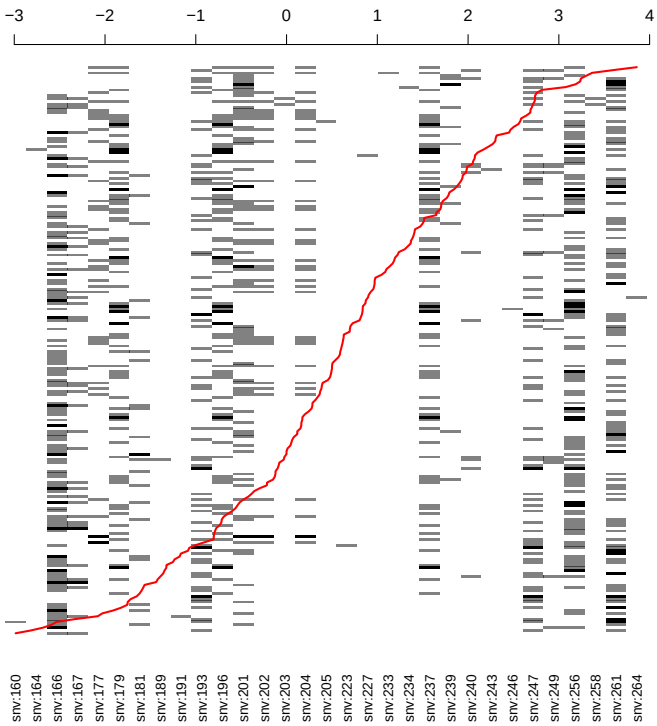
As a further analysis tool, PODKAT offers to plot the genotype in a heatmap-like fashion, as it is or with respect to traits/phenotypes. In the following example, we read the region that has been identified as most significant from the VCF file and display it:

```
res.c.adj.sorted <- sort(res.c.adj, sortBy="p.value.adj")
Zi <- readGenotypeMatrix(vcfFile, regions=res.c.adj.sorted[1])
plot(Zi, labRow=NA)
```



The plot is more expressive if the genotypes are plotted along with the trait/phenotype:

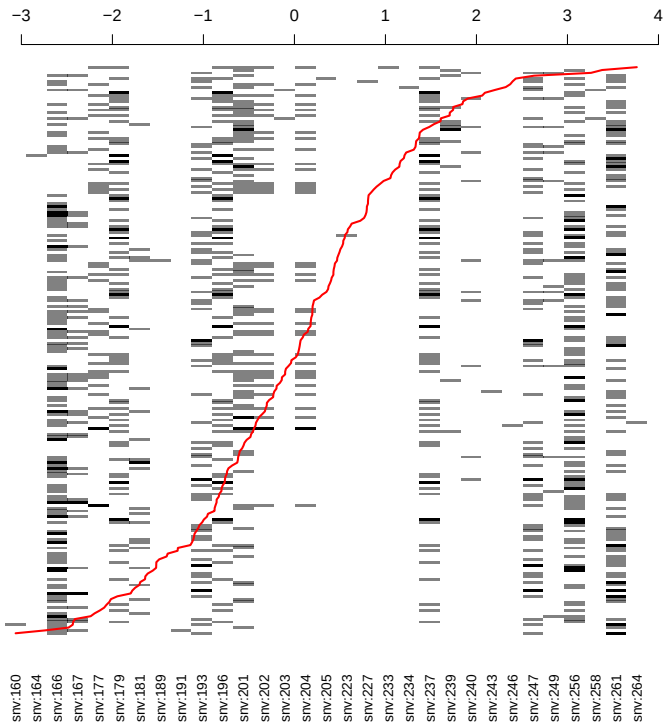
```
plot(Zi, y=pheno.c$y, labRow=NA)
```



In this plot, the samples (rows) are sorted according to the phenotype value. This allows for better finding variants whose minor alleles (gray or black) accumulate in the upper or lower part of the plot. For instance, it is clearly visible that the minor alleles of variant `snv:239`, the one that had the highest contribution, accumulate in the upper part of the plot.

Note, however, that the genotypes have not been tested for associations with the trait directly, but for associations with the trait after correction for covariates (i.e. with the null model residuals):

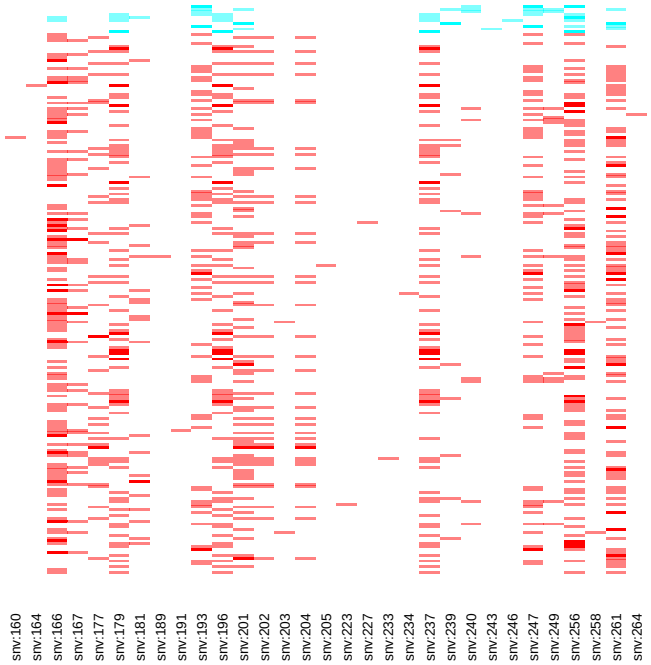
```
plot(Zi, y=residuals(model.c), labRow=NA)
```



Now the accumulation of minor alleles of variant `snv:239` in the upper part of the plot becomes even more prominent.

Analogous functionality is also available for binary traits. In such a case, however, samples are sorted according to class (trait 0 or 1) and color-coded:

```
res.b.adj.sorted <- sort(res.b.adj, sortBy="p.value.adj")
Zi <- readGenotypeMatrix(vcfFile, regions=res.b.adj.sorted[1])
plot(Zi, y=factor(pheno.b$y), labRow=NA)
```



Note that, if the binary trait is numerical, it must be passed as a factor in order to ensure that `plot()` correctly recognizes it as a categorical entity.

8 Miscellanea

8.1 Creating Suitable VCF Files

The main file format for storing variant calls, no matter whether they have been determined by microarrays or next-generation sequencing, is the *Variant Call Format (VCF)*.[6] Not surprisingly, this is the main file format that is used and supported by PODKAT. However, in order to make VCF files suitable as input files for PODKAT, some preparations may be necessary. Most importantly, *PODKAT expects all samples of a study to be included in one single VCF file.* In other words, if samples are spread over multiple VCF files, these files must be *merged* before PODKAT can be used.

8.1.1 Software tools

All steps described below require that `tabix` and `bgzip` are available on the computer system on which the VCF preprocessing steps are to be performed. If they are not yet available, the latest

[6] `http://www.1000genomes.org/wiki/analysis/variant-call-format/vcf-variant-call-format-version-42` (last visited 2021-04-30)

version of `tabix` [9] must be downloaded,[7] compiled, and installed. This package also includes `bgzip`, so no extra effort is necessary to install `bgzip`. Make sure that the executables `tabix` and `bgzip` are in the default search path.

For merging and concatentating VCF files, either install the Perl-based VCFtools[8] [4] or the faster bcftools/htslib VCF commands.[9] Make sure that the necessary executables are in the default search path.

All examples below are supposed to run on Unix-like systems (including Linux and Mac OS X). The examples below that are prefixed with `$` are supposed to be run in a Unix/Linux terminal, not in an R session. If the user wants to run them on a MS Windows system, some modifications may be necessary.

8.1.2 Merging VCF files

Suppose that the samples of a study are distributed across multiple Gnu-zipped VCF files, e.g. `s1.vcf.gz` – `s100.vcf.gz`. Further suppose that a Tabix index file is available for each of these files. If this is not the case, `tabix` must be run on the VCF files to create this index file. As an example,

```
$ tabix -p vcf s1.vcf.gz
```

will create an index file `s1.vcf.gz.tbi`. As said, if an index file is available for each VCF file, the files are ready to be merged. Using the Perl-based VCFtools, this can be done as follows:

```
$ vcf-merge -c both s*.vcf.gz | bgzip -c > merged.vcf.gz
$ tabix -p vcf merged.vcf.gz
```

The first command above merges all files `s1.vcf.gz` – `s10.vcf.gz` into a newly created VCF file `merged.vcf.gz`. The second command again creates a Tabix index file of the merged VCF file. The option `-c both` ensures that single-nucleotide variants (SNVs)[10] and indels ocurring at overlapping locations are kept separate and not merged.

If the bcftools/htslib VCF commands are used, merging can be done in a very similar way:

```
$ bcftools merge -m both s*.vcf.gz | bgzip -c > merged.vcf.gz
$ tabix -p vcf merged.vcf.gz
```

Note that `-c both` must be replaced by `-m both` in this variant; the meaning, however, is the same.

The above commands are to be used in cases where the sample sets of the individual VCF files are disjoint/non-overlapping. Do not use them in case that the sample sets of the VCF files overlap, as this would lead to duplication of samples in the merged VCF file!

[7] `http://sourceforge.net/projects/samtools/files/tabix/` (last visited 2021-04-30)
[8] `https://vcftools.github.io/index.html` (last visited 2021-04-30)
[9] `https://vcftools.github.io/htslib.html` (last visited 2021-04-30)
[10] which of course includes single-nucleotide polymorphisms (SNPs)

8.1.3 Concatenating VCF files

Suppose that the study data are split over multiple files each of which contains the same samples, but each of which contains different variants, e.g. each file contains variants of one chromosome. Then PODKAT can be used in two ways: (1) by running association tests for each VCF file/chromosome independently and merging the results afterwards or (2) by *concatenating the VCF files into one single VCF file.* The latter can be accomplished as follows (for an example in which we have files `chr1.vcf.gz` – `chr22.vcf.gz` and `chrX.vcf.gz` all of which have been indexed previously):

```
$ vcf-concat chr*.vcf.gz | bgzip -c > concat.vcf.gz
$ tabix -p vcf concat.vcf.gz
```

Suppose the same scenario as above, but with an additional file `chrY.vcf.gz` which contains only the male samples of the study. This case can be handled with the `-p` option which merges files even if the sample sets do not agree. In such a case, missing values are imputed on the Y chromosome for female samples:

```
$ vcf-concat -p chr*.vcf.gz | bgzip -c > concat.vcf.gz
$ tabix -p vcf concat.vcf.gz
```

8.1.4 Filtering VCF files

If some special filtering steps should be performed prior to running an association test with PODKAT, the commands `vcf-annotate` or `bcftools annotate` can be used. See the tools' documentation for more information [4].

8.2 Reading from VCF Files

The PODKAT package provides a lightweight, fast, and memory-efficient method for reading genotypes from VCF files. Like the `readVcf()` function from the `VariantAnnotation` package [15], it uses the `tabix` API provided by the `Rsamtools` package [9, 14]. In contrast to `readVcf()`, however, it concentrates on the absolutely necessary minimum and passes the result as a sparse matrix, which greatly reduces memory usage, in particular, if many variants have a low minor allele frequency. The name of the function that has already been used above is `readGenotypeMatrix()`. As a first argument, it expects a `TabixFile` object or simply the filename of a VCF file. The function allows for reading the entire VCF file at once or for limiting to certain genomic regions. The latter can be accomplished by passing a `GRanges` object with the regions of interest to `readGenotypeMatrix()` as argument `regions`. All of this functionality has been used above already (cf. Sections 3 and 6 above).

The function `readGenotypeMatrix()` can be used as an alternative to the `readVcf()` function from the `VariantAnnotation` package [15]. However, the following restrictions have to be taken into account:

- The returned object does not provide the exact genotypes. Instead, only integer values are returned in sparse matrix format. A 1 corresponds to one minor allele, whereas higher numbers correspond to a higher number of minor alleles. Phasing information is not taken into account and different minor alleles are not distinguished either. All values not present in the sparse matrix object are considered as major alleles only.
- The `readGenotypeMatrix()` function does not allow for returning missing values. The `na.action` option allows for three ways of treating missing values in the VCF file: if `"impute.major"` (the default), all missing values are imputed with major alleles; if `"omit"`, all variants with missing values are ignored and omitted from the output object; if `"fail"`, the function stops with an error when it encounters the first missing value.

The function further allows for omitting indels, for omitting variants that do not have a `PASS` in the `FILTERS` column of the VCF file, for omitting variants exceeding a certain ratio of missing values, for omitting variants with an MAF above a certain threshold, and for swapping minor and major alleles if a variant has an MAF greater than 50%. For details, see the help page of `readGenotypeMatrix()`.

The PODKAT package further provides a method for reading basic info, such as, alleles, types of mutations, and minor allele frequencies (MAFs), from a VCF file without actually reading the genotypes:

```
vInfo <- readVariantInfo(vcfFile, omitZeroMAF=FALSE, refAlt=TRUE)
vInfo
```

```
## VariantInfo object with 3117 ranges and 4 metadata columns:
##            seqnames    ranges strand |         type       MAF
##               <Rle> <IRanges>  <Rle> |     <factor> <numeric>
##      snv:1     chr1        79      * | TRANSITION      0.0000
##      snv:2     chr1       281      * | TRANSVERSION    0.0000
##      snv:6     chr1       428      * | TRANSVERSION    0.1025
##      snv:7     chr1       501      * | TRANSITION      0.0900
##      snv:8     chr1       536      * | TRANSVERSION    0.0000
##        ...      ...       ...    ... .           ...       ...
##   snv:3840     chr1    199676      * | TRANSVERSION    0.2500
##   snv:3842     chr1    199696      * | TRANSVERSION    0.0025
##   snv:3843     chr1    199812      * | TRANSVERSION    0.0025
##   snv:3844     chr1    199879      * | TRANSVERSION    0.0025
##   snv:3845     chr1    199956      * | TRANSVERSION    0.0000
##                    ref         alt
##            <character> <character>
##      snv:1           C           T
##      snv:2           G           C
##      snv:6           A           C
##      snv:7           G           A
##      snv:8           A           T
##        ...         ...         ...
```

```
##   snv:3840          T          A
##   snv:3842          G          C
##   snv:3843          A          C
##   snv:3844          T          A
##   snv:3845          G          T
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths

summary(vInfo)

## [1] "VariantInfo object with 3117 ranges and 4 metadata columns"
```

The object returned by `readVariantInfo()` is of class `VariantInfo` which is essentially a `GRanges` object with the information about the variants stored in its metadata columns. By default, `readVariantInfo()` does not return reference and alternate alleles. This must be enforced by `refAlt=TRUE`. In any case, even if reference and alternate alleles are not returned, the function returns information about the type of mutation (transition, transversion, multiple alternate alleles, indel, or unknown/other).

8.3 Using Genotypes from Other Data Sources

The Variant Call Format[11] (VCF) is the primary file format supported by the PODKAT package. If a user has genotype data in another format, there are basically two ways of using such data:

1. Use some software tool to convert the genotype data into a VCF file.
2. Convert the data to a matrix format (inside or outside of R) and pass the matrix data to PODKAT.

The first approach above obviously requires no adapation of the PODKAT workflow. For the second approach, the matrix data have to be converted to a `GenotypeMatrix` object first. This can be done simply by the `genotypeMatrix()` constructor. This function converts the matrix to a columnwise sparse matrix and attaches positional information to it. The original matrix has to be passed to `genotypeMatrix()` such that rows correspond to samples and columns correspond to variants. The values in the matrix need to conform to PODKAT's interpretation (0 ... only major alleles / other values ... number of minor alleles). Here is a simple example with a random matrix:

```
A <- matrix(rbinom(10000, size=1, prob=0.05), 200, 50)
pos <- sort(sample(1:200000, 50))
Z <- genotypeMatrix(A, pos=pos, seqnames="chr1")
Z
```

[11] `http://www.1000genomes.org/wiki/analysis/variant-call-format/vcf-variant-call-format-version-42` (last visited 2021-04-30)

```
## Genotype matrix:
##  Number of samples: 200
##  Number of variants: 50
##
##  Mean MAF:    0.02425
##  Median MAF:  0.025
##  Minimum MAF: 0.01
##  Maximum MAF: 0.04

variantInfo(Z)

## VariantInfo object with 50 ranges and 1 metadata column:
##        seqnames    ranges strand |       MAF
##           <Rle> <IRanges>  <Rle> | <numeric>
##    [1]     chr1      7366      * |    0.0225
##    [2]     chr1      9838      * |    0.0225
##    [3]     chr1     10460      * |    0.0275
##    [4]     chr1     12599      * |    0.0250
##    [5]     chr1     18120      * |    0.0350
##    ...      ...       ...    ... .       ...
##   [46]     chr1    192558      * |    0.0225
##   [47]     chr1    193800      * |    0.0200
##   [48]     chr1    195719      * |    0.0300
##   [49]     chr1    197415      * |    0.0225
##   [50]     chr1    199436      * |    0.0250
##   -------
##   seqinfo: 1 sequence from an unspecified genome; no seqlengths
```

In the example above, positions and chromosome names are supplied as a numeric vector and a character vector, respectively. It is also possible to supply a `GRanges` object directly.

Though PODKAT has mainly been designed for analyzing sequencing data, there are also `genotypeMatrix()` constructors that create a `GenotypeMatrix` object from an expression set (`eSet`) object. Finally, it is worth to mention that PODKAT also can handle VCF data that are stored in an object of class `VCF` (cf. package `VariantAnnotation` [15]). For details, see the help page of `genotypeMatrix()`.

8.4 Preparations for a New Genome

As described in Section 5, it is necessary to define regions of interest for association testing. For whole-genome association testing, this is typically done by partitioning the sequenced regions of a genome into windows (overlapping or non-overlapping). The PODKAT package provides readymade `GRangesList` objects with the sequenced regions of the following genomes: hg18, hg19, hg38, b36, and b37. For other genomes, the sequenced regions must be pre-processed first and stored to an object of class `GRanges` or `GRangesList`. PODKAT provides a func-

tion `unmaskedRegions()` that allows for performing this pre-processing step conveniently for genomes given as `MaskedBSgenome` objects.

The following example shows how this can be done for the autosomal chromosomes of the mouse mm10 genome:

```
library(BSgenome.Mmusculus.UCSC.mm10.masked)
regions <- unmaskedRegions(BSgenome.Mmusculus.UCSC.mm10.masked,
                           chrs=paste0("chr", 1:19))
names(regions)

##  [1] "chr1"  "chr2"  "chr3"  "chr4"  "chr5"  "chr6"  "chr7"
##  [8] "chr8"  "chr9"  "chr10" "chr11" "chr12" "chr13" "chr14"
## [15] "chr15" "chr16" "chr17" "chr18" "chr19"

regions$chr1

## GRanges object with 21 ranges and 0 metadata columns:
##        seqnames              ranges strand
##           <Rle>           <IRanges>  <Rle>
##    [1]     chr1     3000001-3002129      *
##    [2]     chr1    3003119-22433268      *
##    [3]     chr1   22466511-75055556      *
##    [4]     chr1   75121557-78608149      *
##    [5]     chr1   78609095-78610150      *
##    ...      ...                 ...    ...
##   [17]     chr1 183503932-192001776      *
##   [18]     chr1 192051777-192152616      *
##   [19]     chr1 192152949-192156656      *
##   [20]     chr1 192163814-192184090      *
##   [21]     chr1 192185012-195371971      *
##   -------
##   seqinfo: 19 sequences from mm10 genome
```

PODKAT also allows for treating pseudo-autosomal regions separately. In order to facilitate association testing in which pseudo-autosomal regions are treated like autosomal regions (and unlike sex chromosomes), it is necessary to define pseudo-autosomal regions from the beginning. The `unmaskedRegions()` function can do that and all it needs is a data frame with positional information about pseudo-autosomal regions. The format of the data frame has been chosen deliberately to make direct use of the definitions of pseudo-autosomal regions as provided by the `GWASTools` package. The following code shows how to extract unmasked regions of sex chromosomes taking pseudoautosomal regions into account (based on hg38):

```
library(BSgenome.Hsapiens.UCSC.hg38.masked)
library(GWASTools)
```

```
pseudoautosomal.hg38 ## from GWASTools package

##        chrom region start.base  end.base
## X.PAR1     X   PAR1      10001   2781479
## X.PAR2     X   PAR2  155701383 156030895
## X.XTR      X    XTR   89140830  93328068
## Y.PAR1     Y   PAR1      10001   2781479
## Y.PAR2     Y   PAR2   56887903  57217415
## Y.XTR      Y    XTR    3058342   6675059

psaut <- pseudoautosomal.hg38
psaut$chrom <- paste0("chr", psaut$chrom)
psaut

##        chrom region start.base  end.base
## X.PAR1  chrX   PAR1      10001   2781479
## X.PAR2  chrX   PAR2  155701383 156030895
## X.XTR   chrX    XTR   89140830  93328068
## Y.PAR1  chrY   PAR1      10001   2781479
## Y.PAR2  chrY   PAR2   56887903  57217415
## Y.XTR   chrY    XTR    3058342   6675059

regions <- unmaskedRegions(BSgenome.Hsapiens.UCSC.hg38.masked,
                           chrs=c("chrX", "chrY"), pseudoautosomal=psaut)
names(regions)

## [1] "chrX"   "chrY"   "X.PAR1" "X.PAR2" "X.XTR"  "Y.PAR1"
## [7] "Y.PAR2" "Y.XTR"

regions$chrX

## GRanges object with 12 ranges and 0 metadata columns:
##        seqnames              ranges strand
##           <Rle>           <IRanges>  <Rle>
##    [1]     chrX    2781480-37099262      *
##    [2]     chrX   37285838-49348394      *
##    [3]     chrX   49528395-50228964      *
##    [4]     chrX   50278965-58555579      *
##    [5]     chrX   58605580-62412542      *
##    ...      ...                 ...    ...
##    [8]     chrX 114331199-115738949      *
##    [9]     chrX 115838950-116557779      *
##   [10]     chrX 116595567-120879381      *
##   [11]     chrX 120929382-144425606      *
##   [12]     chrX 144475607-155701382      *
##   -------
##   seqinfo: 2 sequences from hg38 genome
```

```
regions$X.PAR1

## GRanges object with 4 ranges and 0 metadata columns:
##       seqnames          ranges strand
##          <Rle>       <IRanges>  <Rle>
##   [1]     chrX     10001-44821      *
##   [2]     chrX    94822-133871      *
##   [3]     chrX  222347-1949345      *
##   [4]     chrX 2132995-2781479      *
##   -------
##   seqinfo: 2 sequences from hg38 genome
```

Except for the fact that the above example only considers the two sex chromosomes, this is exactly the R code that was used to create the `hg38Unmasked` data object provided by PODKAT. The other objects for hg18, hg19, b36, and b37 also contain pseudo-autosomal regions as separate components.

8.5 Handling Large Data Sets

Small or moderately sized association tests like the examples presented above can be performed within seconds on a regular desktop computer. Large whole-genome studies, for example, the two whole-genome cohorts from the UK10K project[12] comprise thousands of samples and tens of millions of variants, and the compressed VCF files amount to hundreds of gigabytes. In order to analyze such vast amounts of data, PODKAT implements two complementary strategies, chunking and parallelization, that we will describe in more detail in the following.

8.5.1 Chunking

The genotype data stored in a 300GB compressed VCF file would hardly fit into the main memory of a supercomputer, let alone a desktop computer. It has already been mentioned above that `assocTest()` can be called for a `TabixFile` object or simply the file name of a compressed VCF file, though we have not yet gone into detail how `assocTest()` processes a VCF file. In fact, it does *not load the entire file* (as we have done above by calling `readGenotypeMatrix()` and then calling `assocTest()` on the returned `GenotypeMatrix` object). Instead, it processes batches of regions, i.e. the variants from a certain number of regions are read from the VCF file and processed at once. How many regions are processed at once is determined by the `batchSize` argument (the default is 20). So suppose that `assocTest()` is called for a `ranges` argument that contains 1000 regions to be tested, then 50 read operations are performed, each time 20 regions are read from the VCF file at once and then processed.

The `batchSize` should be chosen such that the entire genotype matrix of the regions of the batch still fits into the computer's main memory. Otherwise swapping will severly slow down the computations. A batch size of 1 is not optimal either, since the large number of reading operations and the redundancies from reading variants of overlapping regions may lead to much

[12] `http://www.uk10k.org/studies/cohorts.html` (last visited 2021-04-30)

overhead. The default of 20 regions is a cautious choice that need not be optimal under all possible circumstances. Depending on the number of samples and the size of the regions, one may safely increase the batch size to 100 or even higher.

By means of its chunking strategy, PODKAT is in principle able to process large VCF files even on desktop computers in a single-processor manner. The resulting computation times, however, can be considerably long. If this is not acceptable, a multi-core system must be used along with PODKAT's parallelization abilities (see next).

8.5.2 Parallel Processing

As mentioned above already, PODKAT allows for performing large association tests on multiple processors. PODKAT makes use of R socket clusters as provided by the `parallel` package (formerly `snow`/`snowfall`). The simplest way of using this approach is to set the `nnodes` argument to a number larger than 1:

```
assocTest(vcfFile, model.b, windows, nnodes=8)

##!!## Overview of association test:
##!!##  Null model: logistic
##!!##  Number of samples: 200
##!!##  Number of regions: 79
##!!##  Number of regions without variants: 0
##!!##  Average number of variants in regions: 24.1
##!!##  Genome: hgA
##!!##  Kernel: linear.podkat
##!!##  p-value adjustment: none
```

In this example, computations are carried out by 8 parallel R client processes. If the computer system has 8 or more cores/processors, these R client processes are typically assigned to different cores/processors. So it makes no sense to set `nnodes` to a number higher than the number of cores/processors; otherwise only unnecessary overhead would be caused.

If the `nnodes` argument is used, the `assocTest()` function creates the socket cluster internally and also shuts it down as soon as the computations have been finished. This is surely acceptable for one-time analyses, but starting and shutting down the cluster creates unnecessary overhead if multiple association tests are to be performed. In such a scenario, it is more efficient if the user creates the socket cluster outside of `assocTest()` and uses it multiple times via the `cl` argument, as in the following example which creates a socket cluster with 8 R client processes, runs two association tests on this cluster, and then shuts down the cluster:

```
cl <- makePSOCKcluster(8)
assocTest(vcfFile, model.b, windows, cl=cl)

##!!## Overview of association test:
##!!##  Null model: logistic
```

```
##!!##  Number of samples: 200
##!!##  Number of regions: 79
##!!##  Number of regions without variants: 0
##!!##  Average number of variants in regions: 24.1
##!!##  Genome: hgA
##!!##  Kernel: linear.podkat
##!!##  p-value adjustment: none

assocTest(vcfFile, model.c, windows, cl=cl)

##!!## Overview of association test:
##!!##  Null model: linear
##!!##  Number of samples: 200
##!!##  Number of regions: 79
##!!##  Number of regions without variants: 0
##!!##  Average number of variants in regions: 24.1
##!!##  Genome: hgA
##!!##  Kernel: linear.podkat
##!!##  p-value adjustment: none

stopCluster(cl)
```

The granularity of parallelization is determined by the `batchSize` argument described in 8.5.1 above: each client process reads and processes as many regions at once as determined by the `batchSize` argument, then hands back control to the R master session until it is assigned the next chunk/batch. Each client reads directly from the VCF file itself. Therefore, no large genomic data need to be exchanged between master and client processes.

Note that socket connections are used for the communication between the R master session and its clients. Since the number of connections that can be opened in an R session is currently limited to 128, the maximum number of possible clients is also limited by 128 — or less if other connections are open in the R master session at the same time.

The above examples are geared to running parallelized association tests on a multi-core/multi-processor machine. The `parallel` package also allows for distributing computations over multiple machines (see documentation of package `parallel` for more details). This also works with PODKAT, however, only under the following two conditions:

1. All necessary packages (including PODKAT) are installed on all machines. Moreover, R versions and package versions need to be at least compatible on the different machines. (better: exactly the same)

2. The path to the VCF file that should be analyzed is accessible to all R processes (master and clients). This can be accomplished by (1) storing a copy of the VCF file on each machine at exactly the same location or (2) by sharing the file between all machines over the network or, in the ideal case, within a clustered file system. Moreover, when `assocTest()` distributes its computations over multiple client processes, the exchange of the null model

object between the master process and the client processes is done via a temporary file. This file must reside in a directory that is accessible with exactly the same path from all clients. For details on how to ensure that, see `?assocTest` for details (`tmpdir` argument).

As said, it is in principle possible to distribute association tests over multiple machines, however, it is more convenient and more efficient to run large association tests on sufficiently large multi-core/multi-processor computers.

It is hard to give general estimations of computation times, since the performance of PODKAT's association tests depends on various factors, such as, number of cores/processors, main memory, bus architecture, file system performance, etc. So we restrict to one, not necessarily representative example: Johannes Kepler University Linz (JKU) operates an SGI® Altix®UltraViolet 1000 compute server with 2,048 cores. On this system, a whole-genome association test (all autosomal human chromosomes) with about 1,800 samples and about 570,000 regions completed within less than 15 minutes (testing against continuous trait; size of compressed VCF file: about 350GB; computation distributed over 120 cores). Despite the intimidating figures of this example, PODKAT is implemented such that it can perform this analysis also on a regular desktop computer with a single processor only. The computation time, however, would amount to about 30 hours.

9 More Details About PODKAT

9.1 Test Statistics

In line with the SNP-set Kernel Association Test[13] (SKAT) [16], PODKAT uses a variance component score test to test for associations between genotypes and traits. SKAT assumes that traits are distributed according to the following semi-parametric mixed models:

$$\begin{aligned} \operatorname{logit}\big(p(y=1)\big) &= \alpha_0 + \boldsymbol{\alpha}^T \cdot \boldsymbol{x} + h(\boldsymbol{z}) && \text{(binary trait)} \\ y &= \alpha_0 + \boldsymbol{\alpha}^T \cdot \boldsymbol{x} + h(\boldsymbol{z}) + \varepsilon && \text{(continuous trait)} \end{aligned}$$

In the above formulas, y is the trait, $\boldsymbol{x}$ is the covariate vector, $\boldsymbol{z}$ is the genotype vector, α_0 is the intercept, $\boldsymbol{\alpha}$ are the fixed effect coefficients, $h(.)$ is an unknown centered smooth function and ε is the error term. SKAT and PODKAT both assume that the function $h(.)$ is from a function space that is generated by a given positive semi-definite kernel function $K(.,.)$ [11].

SKAT's and PODKAT's null hypothesis is that y is not influenced by the genotype:

$$\begin{aligned} p(y=1) &= \operatorname{logit}^{-1}\big(\alpha_0 + \boldsymbol{\alpha}^T \cdot \boldsymbol{x}\big) && \text{(binary trait)} \\ y &= \alpha_0 + \boldsymbol{\alpha}^T \cdot \boldsymbol{x} + \varepsilon && \text{(continuous trait)} \end{aligned}$$

As mentioned above, we use a *variance component score test* [10, 11, 16] to test against the null hypothesis. More specifically, assume that a study consists of l samples. The traits are, therefore, given as a vector $\boldsymbol{y} \in \{0,1\}^l$ (if the trait is binary) or $\boldsymbol{y} \in \mathbb{R}^l$ (if the trait is continuous). Covariates are given as an $l \times n$ matrix $\mathbf{X}$, and genotypes are given as an $l \times d$ matrix $\mathbf{Z}$. Further suppose that a (logistic) linear model has been trained to predict $\boldsymbol{y}$ from the covariates only. For

[13] formerly known as Sequence Kernel Association Test

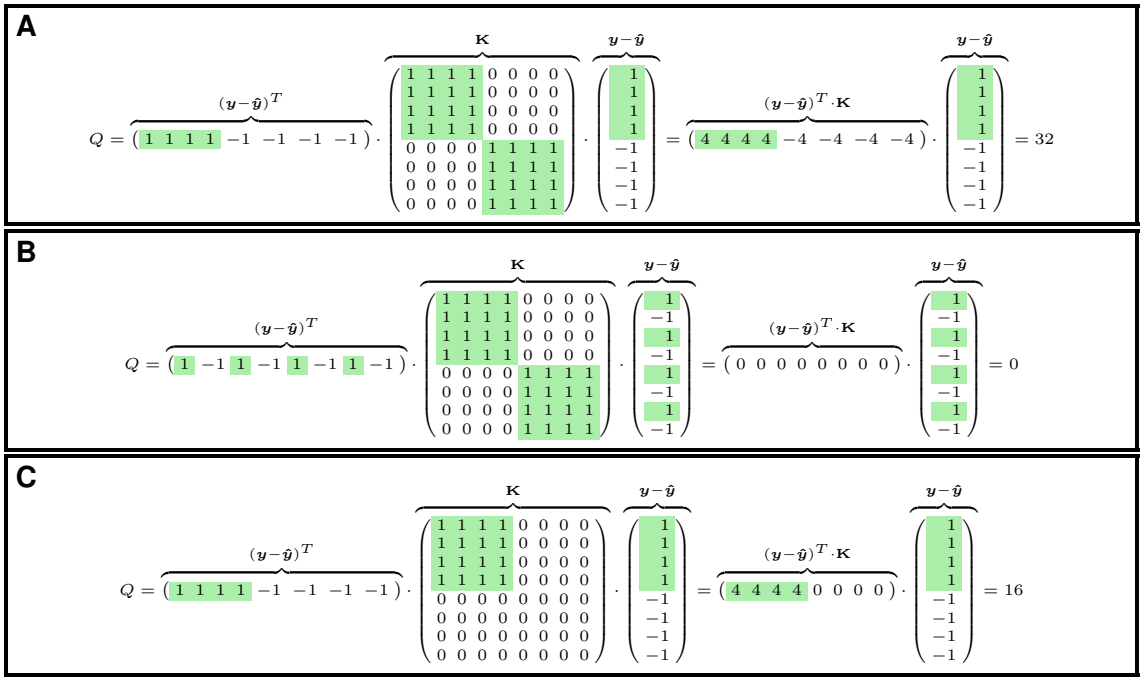


Figure 2: Examples with 8 samples illustrating the variance component test score. **A:** strong correspondence between the signs of residuals and the blocks in the kernel matrix $\mathbf{K}$ results in high Q. **B:** no correspondence between signs of residuals and blocks in $\mathbf{K}$ results in low Q. **C:** even if there is only a partial correspondence between the signs of the residuals and the blocks in $\mathbf{K}$, a relatively high Q is obtained.

a continuous trait, we denote the obtained predictions/fitted values with $\hat{\boldsymbol{y}}$. For a binary trait, $\hat{\boldsymbol{y}}$ denotes the estimated/fitted probabilities that each sample belongs to the positive class. Then, in both cases, $\boldsymbol{y} - \hat{\boldsymbol{y}}$ corresponds to the null model residuals, i.e. what cannot be explained by the covariates only. Then the test statistic is defined as

$$Q = (\boldsymbol{y} - \hat{\boldsymbol{y}})^T \cdot \mathbf{K} \cdot (\boldsymbol{y} - \hat{\boldsymbol{y}}),$$

where $\mathbf{K}$ is an $l \times l$ positive semi-definite kernel matrix defined as $K_{i,j} = K(\boldsymbol{z}_i, \boldsymbol{z}_j)$, where $\boldsymbol{z}_i$ and $\boldsymbol{z}_j$ are the genotypes of the i-th and j-th sample, respectively, i.e. the i-th and j-th row of the genotype matrix $\mathbf{Z}$. The kernel matrix $\mathbf{K}$ can be understood as a matrix that measures the pairwise similarity of genotypes of samples. Since $\mathbf{K}$ is positive semi-definite, Q is non-negative. The more structure the residual $\boldsymbol{y} - \hat{\boldsymbol{y}}$ and the matrix $\mathbf{K}$ share, the larger Q (see Figure 2 for illustrative examples). If the residuals and the genotypes are independent, i.e. if the test's null hypothesis holds true, large values can only occur by pure chance with a low probability. Hence, we test whether the actually observed Q value is higher than expected by pure chance. In other words, the test's p-value is computed as the (estimated) probability of observing a value under the null hypothesis of independence between genotypes and traits that is at least as large as the observed Q.

In order to compute the p-value, we need to know the null distribution of Q, i.e. how Q is distributed under the assumption that y does not depend on $\boldsymbol{z}$. For continuous traits and normally

distributed noise ε, the residuals are normally distributed and the distribution of Q is obviously a *mixture of χ^2 distributions*. For binary traits, Q approximately follows a *mixture of χ^2 distributions*, too [10, 16]. In either case,

$$Q \sim \sum_{k=1}^{q} \lambda_k \cdot \chi^2_{1,k}, \tag{1}$$

where $\chi^2_{1,k}$ are independent χ^2-distributed random variables with one degree of freedom and $\lambda_1, \ldots, \lambda_q$ are the non-zero eigenvalues of the positive semi-definite $l \times l$ matrix

$$\mathbf{P}_0^{\frac{1}{2}} \cdot \mathbf{K} \cdot \mathbf{P}_0^{\frac{1}{2}}.$$

with

$$\mathbf{P}_0 = \mathbf{V} - \mathbf{V} \cdot \tilde{\mathbf{X}} \cdot \left(\tilde{\mathbf{X}}^T \cdot \mathbf{V} \cdot \tilde{\mathbf{X}}\right)^{-1} \cdot \tilde{\mathbf{X}}^T \cdot \mathbf{V}. \tag{2}$$

In (2), $\mathbf{V}$ is a diagonal matrix defined as $\mathbf{V} = \hat{\sigma} \cdot \mathbf{I}$ for continuous traits (with $\hat{\sigma}$ being the estimated variance of regression residuals under the null hypothesis) and as $\mathbf{V} = \operatorname{diag}\big(\hat{y}_1(1-\hat{y}_1), \ldots, \hat{y}_l(1-\hat{y}_l)\big)$ for binary traits, where the diagonal entries $\hat{y}_i(1-\hat{y}_i)$ are obviously the estimated individual variances of the fitted values of the null model [16]. Furthermore, $\tilde{\mathbf{X}} = (\mathbf{1} \mid \mathbf{X})$ is the model matrix of the (logistic) linear model trained on the covariates. Once the eigenvalues $\lambda_1, \ldots, \lambda_q$ have been determined, a method for estimating the probability distribution function of a mixture of χ^2 distributions can be used to compute the tests p-value, such as, Davies' method [5] or Liu's method [12]. Like SKAT does too, PODKAT implements both methods and the method can be chosen with the `method` argument (unless small sample correction is used; see Subsection 9.5 below).

PODKAT further implements a new variant of the above test that is suitable for smaller studies with binary traits in which no covariates need to be taken into account. This test assumes the following random effects model:

$$\operatorname{logit}\big(p(y=1)\big) = \alpha_0 + h(\boldsymbol{z})$$

The null hypothesis of this variant is that the trait is Bernoulli-distributed with a constant probability and independent of the genotype, i.e.

$$p(y=1) = \operatorname{logit}^{-1}(\alpha_0).$$

With the same assumptions as above, the test statistic is given as

$$Q = \boldsymbol{y}^T \cdot \mathbf{K} \cdot \boldsymbol{y}.$$

Since all y_i are Bernoulli-distributed (we estimate the probability of a positive outcome as the relative frequency of positive outcomes $\bar{p} = \frac{1}{l}\sum_{i=1}^{l} y_i$), Q is distributed according to a *mixture of Bernoulli distribitions* and the exact p-value of the test can be computed as

$$p = \sum_{\boldsymbol{y}} p(\boldsymbol{y}) \cdot \left\{ \begin{array}{ll} 1 & \text{if } \boldsymbol{y}^T \cdot \mathbf{K} \cdot \boldsymbol{y} \geq Q \\ 0 & \text{otherwise} \end{array} \right\},$$

i.e. the sum of probabilities of outcomes that produce a test statistic at least as large as Q. For a given outcome $\boldsymbol{y} = (y_1, \ldots, y_l)$, the probability is given as

$$p(\boldsymbol{y}) = \bar{p}^k \cdot (1-\bar{p})^{l-k} \qquad \text{with } k = \sum_{i=1}^{l} y_i.$$

Since all y_i are binary, $Q = \boldsymbol{y}^T \cdot \mathbf{K} \cdot \boldsymbol{y}$ is nothing else but the sum of the sub-matrix of $\mathbf{K}$ that consists of all those rows and columns i for which $y_i = 1$ holds. If all entries of $\mathbf{K}$ are non-negative, the test's exact p-value can be computed by a recursive algorithm that starts from $\boldsymbol{y} = (1, \ldots, 1)$ and recursively removes ones as long as $\boldsymbol{y}^T \cdot \mathbf{K} \cdot \boldsymbol{y} \geq Q$ holds.

In order to use this variant, the null model must be created with `type="bernoulli"`. With the default setting `type="automatic"`, this variant is chosen if the trait is binary, no covariates are present, and the number of samples does not exceed 100. The latter restriction is necessary to avoid excessive computation times caused by the recursive computation of exact p-values.

9.2 Kernels

It is clear from the previous section that the association tests implemented in PODKAT rely on the choice of a kernel function that computes the pairwise similarities of the samples' genotypes. The simplest kernel is the *linear kernel* that computes the similarities as the outer product of the genotype matrices:

$$\mathbf{K} = \mathbf{Z} \cdot \mathbf{Z}^T$$

This kernel can be used by calling `assocTest()` with the option `kernel="linear.SKAT"` and without any weighting (`weightFunc=NULL`). The kernel's name has been deliberately chosen to indicate that this is nothing else but the linear SKAT test without weights. The same test can also be carried out with the *weighted linear kernel*

$$\mathbf{K} = \mathbf{Z} \cdot \mathbf{W} \cdot \mathbf{W}^T \cdot \mathbf{Z}^T, \tag{3}$$

where $\mathbf{W}$ is a diagonal matrix of weights $\mathbf{W} = \mathrm{diag}(w_1, \ldots, w_d)$ with which the columns (i.e. variants) in the genotype matrix $\mathbf{Z}$ are scaled before the outer product is computed. If `assocTest()` is called for a VCF file, the weighting must be done via a *weighting function* that computes the variants' weights as a function of their minor allele frequencies (see Subsection 9.3 below). If `assocTest()` is called for a matrix-like object, weights can be also specified as a per-column weight vector using the `weights` argument.

The acronym PODKAT stands for *Position-Dependent Kernel Association Test*. This test uses a *position-dependent linear kernel*:

$$\mathbf{K} = \mathbf{Z} \cdot \mathbf{W} \cdot \mathbf{P} \cdot \mathbf{P}^T \cdot \mathbf{W}^T \cdot \mathbf{Z}^T, \tag{4}$$

The $d \times d$ matrix $\mathbf{P}$ is a positive semi-definite kernel matrix that measures the similarities/closeness of positions of variants, i.e.

$$P_{i,j} = \max\left(1 - \tfrac{1}{w}|\mathrm{pos}_i - \mathrm{pos}_j|, 0\right),$$

where pos_i and pos_j are the genomic positions of the i-th and the j-th variant, respectively. The parameter w determines the *maximal radius of tolerance*: if two positions are the same, the kernel gives a similarity of 1; the similarity decreases linearly with increasing genomic distance of the two variants under consideration; if the distance is w or larger, the positional similarity is 0. This similarity is actually a positive semi-definite kernel [1, 3]. Figure 3 shows how the similarity depends on the difference of positions. This kernel can be used by calling `assocTest()` with the option `kernel="linear.podkat"` (which is the default). Weighting can be configured in the

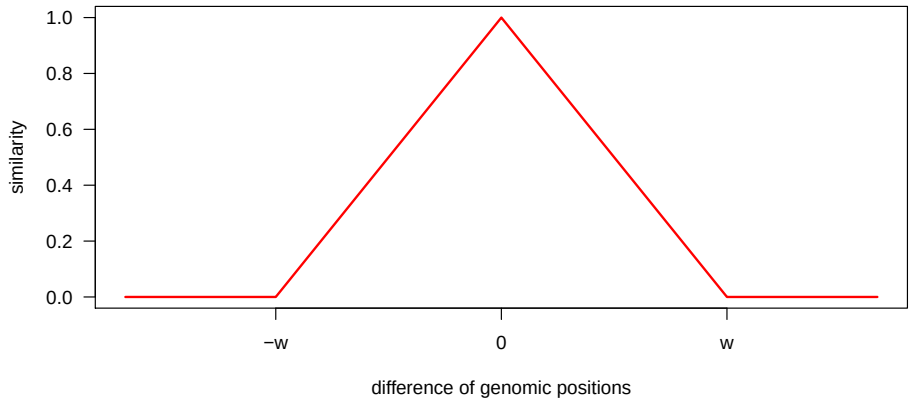


Figure 3: Graph demonstrating how the similarity of genomic positions is computed depending on the difference of positions.

same way as described for the linear kernel above. The radius of tolerance w can be set with the parameter `width` (the default is 1,000 bp).

The motivation behind PODKAT's position-dependent kernel is to better account for very rare or even *private variants*, that is, variants that only occur in one sample. The linear kernel is not able to take private variants into account. In order to demonstrate that, consider how a single entry of the kernel matrix is computed (for simplicity without any weighting):

$$K_{i,j} = \sum_{k=1}^{d} Z_{i,k} \cdot Z_{j,k}$$

If the k-th variant is private, there is only one i for which $Z_{i,k} > 0$, whereas all other $Z_{j,k} = 0$. Hence, the k-th variant makes no contribution to $K_{i,j}$ for $i \neq j$. The position-dependent kernel, however, computes a convolution of the genotype matrix with the position kernel and thereby makes use of agglomerations of private variants in the same genomic region.

Figure 4 shows a simple example that demonstrates how PODKAT's position-dependent kernel takes private variants into account. On the left, Panel A, shows the genotype matrix $\mathbf{Z}$. Obviously, $\mathbf{Z}$ consists of 6 samples and 11 variants, each of which occurs in only one sample. The right-hand side of Panel A shows the kernel matrix that would be obtained for the linear kernel. Since all variants are private, the kernel matrix is diagonal, which does not allow for any meaningful association testing, since there are no blocks of similar samples in the kernel matrix (compare with Section 9.1 and the examples in Figure 2). The left side of Panel B shows the convolution of the genotype matrix $\mathbf{Z}$ with the positional similarities $\mathbf{P}$, and the right side shows the resulting kernel matrix for the position-dependent kernel. Suddenly, two blocks of samples (samples 1–3 and samples 4–6) become visible, as a result of the fact that samples 1–3 have minor alleles in the left half of the sequence and samples 4–6 have minor alleles in the right half of the sequence. Now suppose that samples 1–3 are cases and samples 4–6 are controls. If the mutations/minor alleles in the left half of the sequence (e.g. a particular exon or transcription factor binding site) are causal for the disease, PODKAT would be able to detect that, whereas SKAT with the regular linear kernel would not be able to detect that.

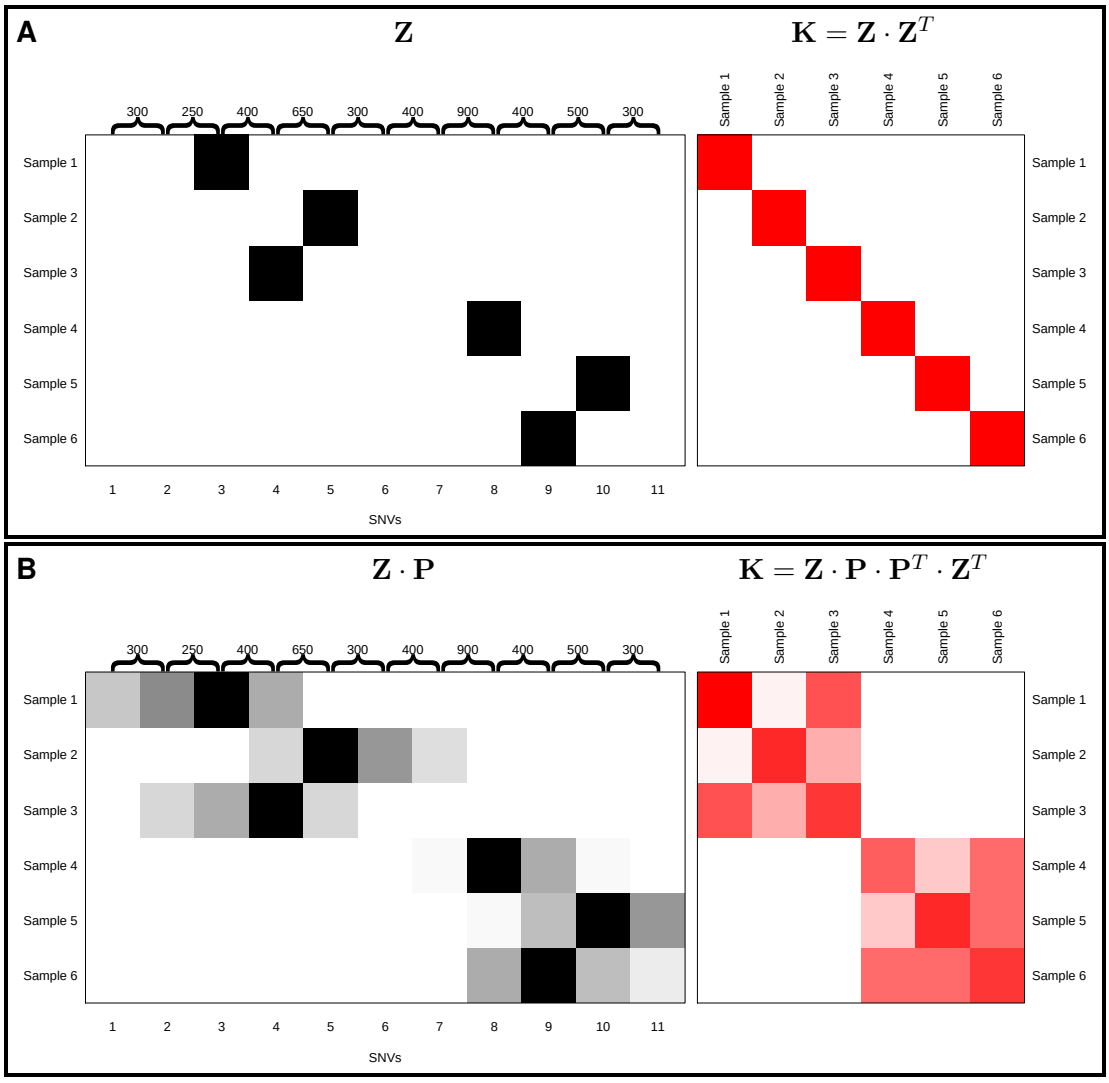


Figure 4: Simple example demonstrating how PODKAT's position-dependent kernel takes private variants into account. **A:** genotype matrix $\mathbf{Z}$ (left) and kernel matrix of the linear kernel (right); **B:** convolution of genotype matrix $\mathbf{Z}$ with positional similarities $\mathbf{P}$ (left) and kernel matrix of the position-dependent kernel (right).

PODKAT further provides the *IBS (identity by state) kernel*

$$K_{i,j} = \frac{1}{\sum_{k=1}^{d} w_d} \sum_{k=1}^{d} w_k \cdot (2 - |Z_{i,k} - Z_{j,k}|)$$

and the *quadratic kernel*

$$K_{i,j} = \left(1 + \sum_{k=1}^{d} w_k \cdot Z_{i,k} \cdot Z_{j,k}\right)^2$$

that are also available in SKAT [16]. In order to make use of these two kernels, `assocTest()` must be called with the parameters `kernel="localsim.SKAT"` or `kernel="quadratic.SKAT"`, respectively. Analogously to the linear kernel, weighting must be controlled with the argument `weightFunc` (or `weights`) and can be switched off with `weightFunc=NULL` which means that `assocTest()` internally sets all $w_k = 1$. Additionally, in order to enable these kernels also to take private variants into account, there are position-dependent variants of the IBS kernel and the quadratic kernel that can be used with kernels `"localsim.podkat"` or `"quadratic.podkat"`, respectively. These kernels first compute the convolution of the genotype matrix $\mathbf{Z}$ with the positional similarities $\mathbf{P}$ (compare with example in Figure 4B) and then compute the kernel matrices according to the formulas above.

It must be pointed out that the linear kernel and the position-dependent linear kernel (kernel choices `linear.SKAT` and `linear.podkat`) can be represented as outer products, which, in many cases, allows for much more efficient computations than the other four kernels. So, the computing times for large whole-genome studies with the four kernel choices `localsim.SKAT`, `localsim.podkat`, `quadratic.SKAT`, and `quadratic.podkat` can be prohibitely long.

9.3 Weighting Functions

As mentioned above, all six kernels implemented in PODKAT can be used with and without weighting. In all six kernels, weights need to be defined in a per-variant fashion (i.e. one weight per column of the genotype matrix $\mathbf{Z}$). If `assocTest()` is called for a VCF file, i.e. its argument `Z` is the file name of a VCF file or a `TabixFile` object, then `assocTest()` requires a one-argument function that computes a vector of weights from a vector of minor allele frequencies (MAFs). This function must be passed as `weightFunc` argument (if it is `NULL`, then the kernels are used without weights).

For convenience, PODKAT provides three built-in functions. Firstly, there is `invSdWeights` which is defined as

$$f(x) = \frac{1}{\sqrt{x \cdot (1 - x)}}, \tag{5}$$

i.e. it computes weights as the reciprocals of the standard deviations of minor allele probabilities [13]:

$$w_k = \frac{1}{\sqrt{\mathrm{MAF}_k \cdot (1 - \mathrm{MAF}_k)}}$$

This variant is provided as function `invSdWeights()`. Figure 5A visualizes this weighting function.

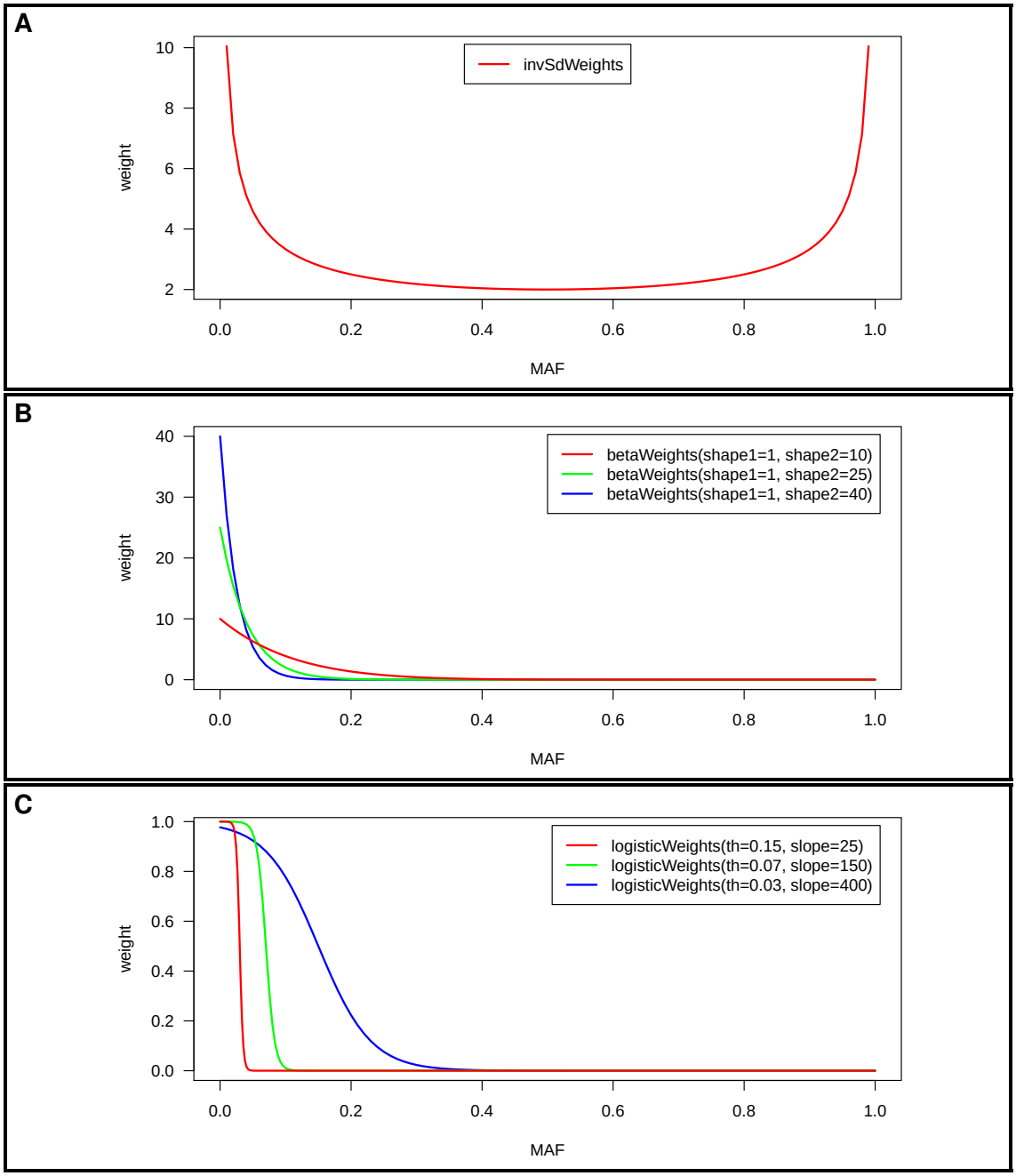


Figure 5: **A:** weighting function (5); **B:** beta distribution weighting functions 6 for different parameters; **C:** soft threshold weighting functions 7 for different parameters.

Secondly, there is a parameterized family of weighting functions that correspond to the densities of the beta distribution:

$$f_{\alpha,\beta}(x) = \frac{x^{\alpha-1} \cdot (1-x)^{\beta-1}}{B(\alpha,\beta)}, \tag{6}$$

where α, β are the two shape parameters and $B(\alpha,\beta)$ is a normalization constant that only depends on the shape parameters. The package provides the function `betaWeights()`. If called with the two arguments `shape1` and `shape2`, this function creates the weighting function with these particular shape parameters. The default weight parameters are $\alpha = 1$ and $\beta = 25$ as suggested by Wu *et al.* [16]. This is actually the default setting of the `weightFunc` parameter of the `assocTest()` function. Figure 5B shows some examples with different shape parameters.

Thirdly, PODKAT offers the possibility to use a *soft threshold function* (logistic function)

$$f(x) = \frac{1}{1 + \exp\big(a \cdot (x-\theta)\big)}, \tag{7}$$

where θ is the threshold and a is the slope that determines how hard/soft the threshold is. The function `logisticWeights()` can be used to create a weighting function with given threshold and slope. Figure 5C shows some examples with parameters.

Users are not limited to the three weighting schemes described above: it is possible to pass arbitrary weighting functions as `weightFunc` argument. However, user-provided weighting functions must conform to some conventions: (1) they must be written such that MAFs can be passed as first arguments; (2) MAFs can be passed as numeric vectors and the result is a numeric vector of the same length; (3) the returned weights are non-negative; (4) the function requires no arguments other than the first one.

9.4 Computing Single-Variant Contributions

As mentioned in Subsection 7.4 above, PODKAT allows for decomposing the test statistic Q into individual contributions of single variants. This is possible for the linear kernel and the position-dependent linear kernel. The linear kernel can be reformulated as

$$Q = (\boldsymbol{y}-\hat{\boldsymbol{y}})^T \cdot \mathbf{Z} \cdot \mathbf{W} \cdot \mathbf{W}^T \cdot \mathbf{Z}^T \cdot (\boldsymbol{y}-\hat{\boldsymbol{y}}) = \| \underbrace{\mathbf{W}^T \cdot \mathbf{Z}^T \cdot (\boldsymbol{y}-\hat{\boldsymbol{y}})}_{=\boldsymbol{p}} \|^2,$$

where $\boldsymbol{p} = \mathbf{W}^T \cdot \mathbf{Z}^T \cdot (\boldsymbol{y}-\hat{\boldsymbol{y}})$ is a vector of length d. In the same way, the position-dependent linear kernel can be rewritten as

$$Q = (\boldsymbol{y}-\hat{\boldsymbol{y}})^T \cdot \mathbf{Z} \cdot \mathbf{W} \cdot \mathbf{P} \cdot \mathbf{P}^T \cdot \mathbf{W}^T \cdot \mathbf{Z}^T \cdot (\boldsymbol{y}-\hat{\boldsymbol{y}}) = \| \underbrace{\mathbf{P}^T \cdot \mathbf{W}^T \cdot \mathbf{Z}^T \cdot (\boldsymbol{y}-\hat{\boldsymbol{y}})}_{=\boldsymbol{p}} \|^2.$$

Again $\boldsymbol{p} = \mathbf{P}^T \cdot \mathbf{W}^T \cdot \mathbf{Z}^T \cdot (\boldsymbol{y}-\hat{\boldsymbol{y}})$ is a vector of length d. So, in both cases, Q can be represented as the squared norm, i.e. the sum of squares, of a vector $\boldsymbol{p}$ of length d, i.e. with one entry per variant:

$$Q = \|\boldsymbol{p}\|^2 = \sum_{k=1}^{d} p_k^2$$

So, the k-th variant contributes p_k^2 to the test statistic Q. The function `weights()` described in Subsection 7.4 computes two values per variant: on the one hand, the raw contribution p_k is stored in the metadata column `weight.raw`. These values are particularly helpful to find out how a variant is associated with the residuals $\boldsymbol{y} - \hat{\boldsymbol{y}}$. If it is positive, the association is positive. If it is negative, the association is negative. On the other hand, the *relative contribution*

$$p'_k = \frac{p_k^2}{\|\boldsymbol{p}\|^2} = \frac{p_k^2}{Q}$$

is stored to the metadata column `weight.contribution`. This value measures how much each variant contributes to the test statistic Q, where these contributions are normalized to sum up to one.

9.5 Details on the Small Sample Correction

As mentioned in the Subsection 9.1, the test statistic of PODKAT's association tests is assumed to follow a mixture of χ^2 distributions (1) [16], where, as already mentioned above, the mixture weights $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_q)$ (with $q \leq l$) are given as the non-zero eigenvalues of the matrix

$$\mathbf{P}_0^{\frac{1}{2}} \cdot \mathbf{K} \cdot \mathbf{P}_0^{\frac{1}{2}} \tag{8}$$

(with $\mathbf{P}_0$ being defined as described in (2) above). This is theoretically provable for continuous traits under the assumptions of the null hypothesis, but holds only approximately for binary traits. In many cases, the identifiation of the mixture of χ^2 distributions as proposed above leads to inflated type I error rates for binary traits. In order to overcome this problem, Lee *et al.* [8] have proposed a *small sample correction* for binary traits for SKAT. PODKAT introduces a new position-dependent kernel for being able to take private and very rare variants into account, but otherwise builds upon the same test framework as SKAT. Therefore, PODKAT also implements the small sample correction according to [8].

The small sample correction alternatively computes the p-value of an association test for binary traits (as described in Subsection 9.1) as

$$1 - P_{\chi^2_{\mathrm{df}}}\Big(\frac{(Q - \mu_Q) \cdot \sqrt{2\mathrm{df}}}{\sigma_Q} + \mathrm{df}\Big), \tag{9}$$

where $P_{\chi^2_{\mathrm{df}}}$ is the cumulative distribution function of a χ^2 distribution with df degrees of freedom, μ_Q is the theoretical mean of the mixture

$$\mu_Q = \sum_{k=1}^{q} \lambda_k \tag{10}$$

i.e. the expected test statistic under the null hypothesis, σ_Q^2 is the theoretical variance of the mixture

$$\sigma_Q^2 = \boldsymbol{\lambda}^T \cdot \mathbf{C} \cdot \boldsymbol{\lambda} \tag{11}$$

and df is the number of degrees of freedom that is computed as

$$\mathrm{df} = \frac{\Big(\sum\limits_{k=1}^{l} \lambda_k'^2\Big)^2}{\sum\limits_{k=1}^{l} \lambda_k'^4}. \tag{12}$$

In (11) above, $\mathbf{C}$ is a $q \times q$ matrix defined as

$$\mathbf{C} = \mathbf{U}_2^T \cdot \mathrm{diag}(\boldsymbol{\varphi}) \cdot \mathbf{U}_2 + 2 \cdot \mathbf{I}$$

where $\mathbf{U}_2$ is an $l \times q$ matrix containing the elementwise squares of $\mathbf{U}$, the $l \times q$ matrix of eigenvectors of non-zero eigenvalues of the matrix (8) and $\boldsymbol{\varphi}$ is a vector of length l the entries of which are defined as

$$\varphi_k = \frac{3p(y_i = 1)^2 - 3p(y_i = 1) + 1}{p(y_i = 1)(1 - p(y_i = 1))} - 3.$$

Practically, the probabilities $p(y_i = 1)$ are estimated by the null model's fitted probabilities $\hat{y}_i$ (see Subsection 9.1). Finally, the values λ'_k in (12) are defined as

$$\lambda'_k = \frac{\lambda_k \cdot C_{k,k}}{\sqrt{2}},$$

where $C_{k,k}$ are the diagonal elements of the matrix $\mathbf{C}$ defined above.

The R package `SKAT` does not exactly use the above method, but uses the matrix

$$\mathbf{V}^{\frac{1}{2}} \cdot \mathbf{K} \cdot \mathbf{V}^{\frac{1}{2}} \tag{13}$$

(with $\mathbf{V}$ defined as in Subsection 9.1) instead to determine the mixture weights. This is mainly for computational efficiency, since the square root of the diagonal matrix $\mathbf{V}$ is much easier to compute than the square root of the dense matrix $\mathbf{P}_0$. It can be proven easily that the eigenvalues of matrix (13) are the same as of matrix (8). However, the eigenvectors, which are needed for computing the matrix $\mathbf{C}$ are not identical, but sufficiently close, as computational simulations have demonstrated. PODKAT, by default, uses this approximation too, but also allows for using the exact matrix (8). In order to make use of this option, the null model has to be created by calling `nullModel()` with the option `adjExact=TRUE`. With this option, `nullModel()` pre-computes the square root of matrix $\mathbf{P}$ and stores it to the slot `P0sqrt` of the returned `NullModel` object.

Lee *et al.* [8] further suggested to adjust the higher moments of the estimated null distribution used in (9) by correcting the degrees of freedom parameter df such that the kurtosis fits to the kurtosis observed in residuals sampled according to the null distribution. PODKAT, like SKAT, offers two methods that can be specified when training a null model with `nullModel()`: `type.resampling="bootstrap"` creates residuals according the estimated Bernoulli distributions with probabilities $\hat{y}_i$; `type.resampling="permutations"` computes permutations of the residuals. The former variant is the default and strongly recommended. So, given the test statistics $Q_1, \ldots, Q_N$ of N sampled vectors of residuals, the correction for higher moments estimates the excess kurtosis $\hat{\gamma}$ and then replaces the parameter df in (9) by

$$\mathrm{df} = \frac{12}{\hat{\gamma}}.$$

PODKAT offers multiple ways for estimating the excess kurtosis from the sampled test statistics $Q_1, \ldots, Q_N$ (the number N can be determined with the `n.resampling.adj` argument of the function `nullModel()`). Which variant is chosen, can be determined with the argument `method` when calling `assocTest()`:

Unbiased sample kurtosis: this method uses the theoretical mean of the null distribution μ_Q (if available; see (10)) and estimates the excess kurtosis as

$$\hat{\gamma} = \frac{\frac{1}{N}\sum_{k=1}^{N}(Q_k - \mu_Q)^4}{\left(\frac{1}{N}\sum_{k=1}^{N}(Q_k - \mu_Q)^2\right)^2} - 3.$$

This variant can be selected with `method="unbiased"`.

Biased sample kurtosis: first computes the empirical mean $\hat{\mu}_1$ of the sampled test and estimates the excess kurtosis as

$$\hat{\gamma} = \frac{\frac{1}{N}\sum_{k=1}^{N}(Q_k - \hat{\mu}_1)^4}{\left(\frac{1}{N}\sum_{k=1}^{N}(Q_k - \hat{\mu}_1)^2\right)^2} - 3.$$

This variant can be selected with `method="sample"`.

Corrected sample kurtosis: this variant is aimed at computing an (almost) unbiased estimator of the excess kurtosis as

$$\gamma = \frac{(N+1)\cdot N\cdot(N-1)}{(N-2)\cdot(N-3)} \cdot \frac{\sum_{k=1}^{N}(Q_k - \hat{\mu}_1)^4}{\left(\sum_{k=1}^{N}(Q_k - \hat{\mu}_1)^2\right)^2} - 3\cdot\frac{(N-1)^2}{(N-2)\cdot(N-3)}.$$

This method can be selected with `method="population"`.

SKAT compatibility mode: this variant is aimed at consistency with the implementation in the R package `SKAT`; it estimates the excess kurtosis as

$$\hat{\gamma} = \frac{\frac{1}{N}\sum_{k=1}^{N}(Q_k - \hat{\mu}_1)^4}{\left(\frac{1}{N-1}\sum_{k=1}^{N}(Q_k - \hat{\mu}_1)^2\right)^2} - 3.$$

This method can be selected with `method="SKAT"`.

As already mentioned briefly in Section 4, the argument `adj` of the `nullModel()` function controls how the null model is prepared to be ready for small sample correction and higher moment correction. The setting `adj="force"` creates sampled residuals for later higher moment correction in any case, while `adj="none"` generally switches off the creation of sample residuals. The default is `adj="automatic"` which creates sampled residuals if the number of samples in the study is at most 2,000. The number of sampled residuals is controlled by the `n.resampling.adj` argument as noted in Section 4 already. The default number of sampled residuals is 10,000.

The function `assocTest()` has an `adj` argument too, the meaning of which is similar to the `adj` argument of the `nullModel()` function: if `assocTest()` is called with `adj="automatic"`

(which is the default), corrections are only made for at most 2,000 samples. For more than 2,000 samples, all corrections are switched off. For `adj="force"`, corrections are generally switched on, and for `adj="none"`, corrections are generally switched off.

If corrections are switched on, the small sample correction described above is performed. As already mentioned, which variant is used is determined by whether the null model has been created with `adjExact=TRUE` or `adjExact=FALSE`.

If the null model includes sampled residuals, the small sample correction is complemented by the higher moment correction described above. In case that `assocTest()` fails to compute the mixture weights $\boldsymbol{\lambda} = (\lambda_1, \ldots, \lambda_q)$, it still uses the formula (9) for computing the p-values, but uses values for μ_Q and σ_Q that were also estimated from the sampled test statistics.

It must be emphasized that both small sample correction and higher moment correction, especially the latter, result in a significant increase of computation times. For larger studies, we suggest to try `assocTest()` without small sample correction first and to create a Q-Q plot to analyze the results. If the p-values in the Q-Q plot are sufficiently close to the diagonal, the test correctly controls the type I error rate and no correction is necessary anyway. If this not the case and the test is either too conservative (i.e. p-values in the Q-Q plot are consistently below the diagonal) or the p-values are inflated (i.e. p-values in the Q-Q plot are consistently above the diagonal), some correction should be applied. To use both corrections regardless of the number of samples, both `nullModel()` and `assocTest()` must be called with `adj="force"`. To use small sample correction, but without correction for higher moments, call `nullModel()` with `adj="none"` (which turns off the creation of sampled residuals), but call `assocTest()` with `adj="force"`.

10 Future Extensions

We plan or at least consider the following extensions in future version of this package:

- Option in the VCF reader that allows for splitting up multiple minor alleles into separate variants. Currently, multiple minor alleles are considered together as if they were synonymous.

11 How to Cite This Package

If you use this package for research that is published later, you are kindly asked to cite it as follows:

> U. Bodenhofer (2022). *PODKAT: an R package for association testing involving rare and private variants.* R package version 1.32.0.

References

[1] L. Belanche, J. L. Vázquez, and M. Vázquez. Distance-based kernels for real-valued data. In C. Preisach, H. Burkhardt, L. Schmidt-Thieme, and R. Decker, editors, *Data Analysis, Machine Learning and Applications*, Studies in Classification, Data Analysis, and Knowledge Organization, pages 3–10. Springer, Berlin, 2008.

[2] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *J. Roy. Statist. Soc. Ser. B*, 57(1):289–300, 1995.

[3] U. Bodenhofer, K. Schwarzbauer, M. Ionescu, and S. Hochreiter. Modeling position specificity in sequence kernels by fuzzy equivalence relations. In J. P. Carvalho, D. Dubois, U. Kaymak, and J. M. C. Sousa, editors, *Proc. Joint 13th IFSA World Congress and 6th EUSFLAT Conference*, pages 1376–1381, Lisbon, July 2009.

[4] P. Danecek, A. Auton, G. Abecasis, C. A. Albers, E. Banks, M. A. DePristo, R. E. Handsaker, G. Lunter, G. T. Marth, S. T. Sherry, G. McVean, R. Durbin, and 1000 Genomes Project Analysis Group. The variant call format and VCFtools. *Bioinformatics*, 27(15):2156–2158, 2011.

[5] R. B. Davies. The distribution of a linear combination of χ^2 random variables. *J. R. Stat. Soc. Ser. C-Appl. Stat.*, 29:323–333, 1980.

[6] S. Holm. A simple sequentially rejective multiple test procedure. *Scand. J. Stat.*, 6(2):65–70, 1979.

[7] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler. The human genome browser at UCSC. *Genome Res.*, 12:996–1006, 2002.

[8] S. Lee, M. J. Emond, M. J. Bamshad, K. C. Barnes, M. J. Rieder, D. A. Nickerson, NHLBI GO Exome Sequencing Project—ESP Lung Project Team, D. C. Christiani, M. M. Wurfel, and X. Lin. Optimal unified approach for rare-variant association testing with application to small-sample case-control whole-exome sequencing studies. *Am. J. Hum. Genet.*, 91(2):224–237, 2012.

[9] H. Li, B. Handsaker, A. Wysoker, T. Fenell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and 1000 Genome Project Data Processing Subgroup. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[10] X. Lin. Variance component testing in generalised linear models with random effects. *Biometrika*, 84(2):309–326, 1997.

[11] D. Liu, D. Ghosh, and X. Lin. Estimation and testing for the effect of a genetic pathway on a disease outcome using logistic kernel machine regression via logistic mixed models. *BMC Bioinformatics*, 9:292, 2008.

[12] H. Liu, Y. Tang, and H. Zhang. A new chi-square approximation to the distribution of non-negative definite quadratic forms in non-central normal variables. *Comput. Stat. Data Anal.*, 53:853–856, 2009.

[13] B. E. Madsen and S. R. Browning. A groupwise association test for rare mutations using a weighted sum statistic. *PLoS Genetics*, 5(2):e1000384, 2009.

[14] M. Morgan, H. Pagès, V. Obenchain, and N. Hayden. *Rsamtools: Binary alignment (BAM), FASTA, variant call (BCF), and tabix file import*, 2015. R package version 1.19.47.

[15] V. Obenchain, M. Lawrence, V. Carey, S. Gogarten, P. Shannon, and M. Morgan. VariantAnnotation: a Bioconductor package for exploration and annotation of genetic variants. *Bioinformatics*, 30(14):2076–2078, 2014.

[16] M. C. Wu, S. Lee, T. Cai, Y. Li, M. Boehnke, and X. Lin. Rare-variant association testing for sequencing data with the sequence kernel association test. *Am. J. Hum. Genet.*, 89(1):82–93, 2011.