Imputed SNP analyses and meta-analysis with snpStats

David Clayton

March 28, 2025

Getting started

The need for imputation in SNP analysis studies occurs when we have a smaller set of samples in which a large number of SNPs have been typed, and a larger set of samples typed in only a subset of the SNPs. We use the smaller, complete dataset (which will be termed the *training dataset*) to impute the missing SNPs in the larger, incomplete dataset (the *target dataset*). Examples of such applications include:

- use of HapMap data to impute association tests for a large number of SNPs, given data from genome-wide studies using, for example, a 500K SNP array, and
- meta-analyses which seek to combine results from two platforms such as the Affymetrix 500K and Illumina 550K platforms.

Here we will not use a real example such as the above to explore the use of `snpStats` for imputation, but generate a fictitious example using the data analysed in earlier exercises. This is particularly artificial in that we have seen that these data suffer from extreme heterogeneity of population structure.

We start by attaching the required libraries and accessing the data used in the exercises:

```
> library(snpStats)
> library(hexbin)
> data(for.exercise)
```

We shall sample 200 subjects in our fictitious study as the training data set, select every second SNP to be missing in the target dataset, and split the training set into two parts accordingly:

```
> training <- sample(1000, 200)
> select <- seq(1, ncol(snps.10),2)
> missing <- snps.10[training, select]
> present <- snps.10[training, -select]
> missing
```

```
A SnpMatrix with  200 rows and  14251 columns
Row names:  ceu.779 ... ceu.9
Col names:  rs7909677 ... rs12218790
```

```
> present
```

```
A SnpMatrix with  200 rows and  14250 columns
Row names:  ceu.779 ... ceu.9
Col names:  rs7093061 ... rs7899159
```

Thus the training dataset consists of the objects `missing` and `present`. The target dataset holds a subset of the SNPs for the remaining 800 subjects.

```
> target <- snps.10[-training, -select]
> target
```

```
A SnpMatrix with  800 rows and  14250 columns
Row names:  jpt.869 ... ceu.464
Col names:  rs7093061 ... rs7899159
```

But, in order to see how successful we have been with imputation, we will also save the SNPs we have removed from the target dataset

```
> lost <- snps.10[-training, select]
> lost
```

```
A SnpMatrix with  800 rows and  14251 columns
Row names:  jpt.869 ... ceu.464
Col names:  rs7909677 ... rs12218790
```

We also need to know where the SNPs are on the chromosome in order to avoid having to search the entire chromosome for suitable predictors of a missing SNP:

```
> pos.miss <- snp.support$position[select]
> pos.pres <- snp.support$position[-select]
```

Calculating the imputation rules

The next step is to calculate a set of rules which for imputing the `missing` SNPs from the `present` SNPs. This is carried out by the function `snp.imputation`[1]:

```
> rules <- snp.imputation(present, missing, pos.pres, pos.miss)
```

[1]Sometimes this command generates a warning message concerning the maximum number of EM iterations. If this only concerns a small proportion of the SNPs to be imputed it can be ignored.

```
SNPs tagged by a single SNP: 4902
SNPs tagged by multiple tag haplotypes (saturated model): 9177
```

This step executes remarkably quickly when we consider what the function has done. For each of the 14251 SNPs in the "missing" set, the function has performed a forward step-wise regression on the 50 nearest SNPs in the "present" set, stopping each search either when the R^2 for prediction exceeds 0.95, or after including 4 SNPs in the regression, or until R^2 is not improved by at least 0.05. The figure 50 is the default value of the `try` argument of the function, while the values 0.95, 4 and 0.05 together make up the default value of the `stopping` argument. After the predictor, or "tag" SNPs have been chosen, the haplotypes of the target SNP plus tags was phased and haplotype frequencies calculated using the EM algorithm. These frequencies were then stored in the `rules` object. [2]

A short listing of the first 10 rules follows:

```
> rules[1:10]

rs7909677  ~  rs2496276+rs7898275+rs3123252+rs2436024 (MAF = 0.07106599, R-squared = 0.5
rs12773042  ~  rs2496276+rs7898275+rs3123252+rs2436024 (MAF = 0.06313131, R-squared = 0.
rs11253563  ~  rs7093061+rs9419496+rs7898275+rs12356744 (MAF = 0.2247475, R-squared = 0.
rs4881552  ~  rs7475011+rs2379078+rs4881551+rs7093061 (MAF = 0.365, R-squared = 0.971211
rs10904596  ~  rs7093061+rs9419496+rs7898275+rs12356744 (MAF = 0.2230769, R-squared = 0.
rs4880781  ~  rs4880983+rs10794885+rs3125027+rs9419496 (MAF = 0.2160804, R-squared = 1)
rs7910845  ~  rs7898275 (MAF = 0.05181347, R-squared = 0.9999981)
rs6560730  ~  rs9419496 (MAF = 0.281407, R-squared = 0.9622619)
rs9329280  ~  rs7898275 (MAF = 0.05778894, R-squared = 0.9537572)
rs4880517  ~  rs9419498+rs7898275+rs3740304 (MAF = 0.03768844, R-squared = 0.9999998)
```

The rules are also selectable by SNP name for detailed examination:

```
> rules[c('rs11253563', 'rs2379080')]

rs11253563  ~  rs7093061+rs9419496+rs7898275+rs12356744 (MAF = 0.2247475, R-squared = 0.
rs2379080  ~  rs4880983+rs2288680+rs2379078+rs10794885 (MAF = 0.2449495, R-squared = 0.9
```

Rules are shown with a + symbol separating predictor SNPs. (It is important to know which SNPs were used for each imputation when checking imputed test results for artifacts.)

A summary table of all the 14,251 rules is generated by

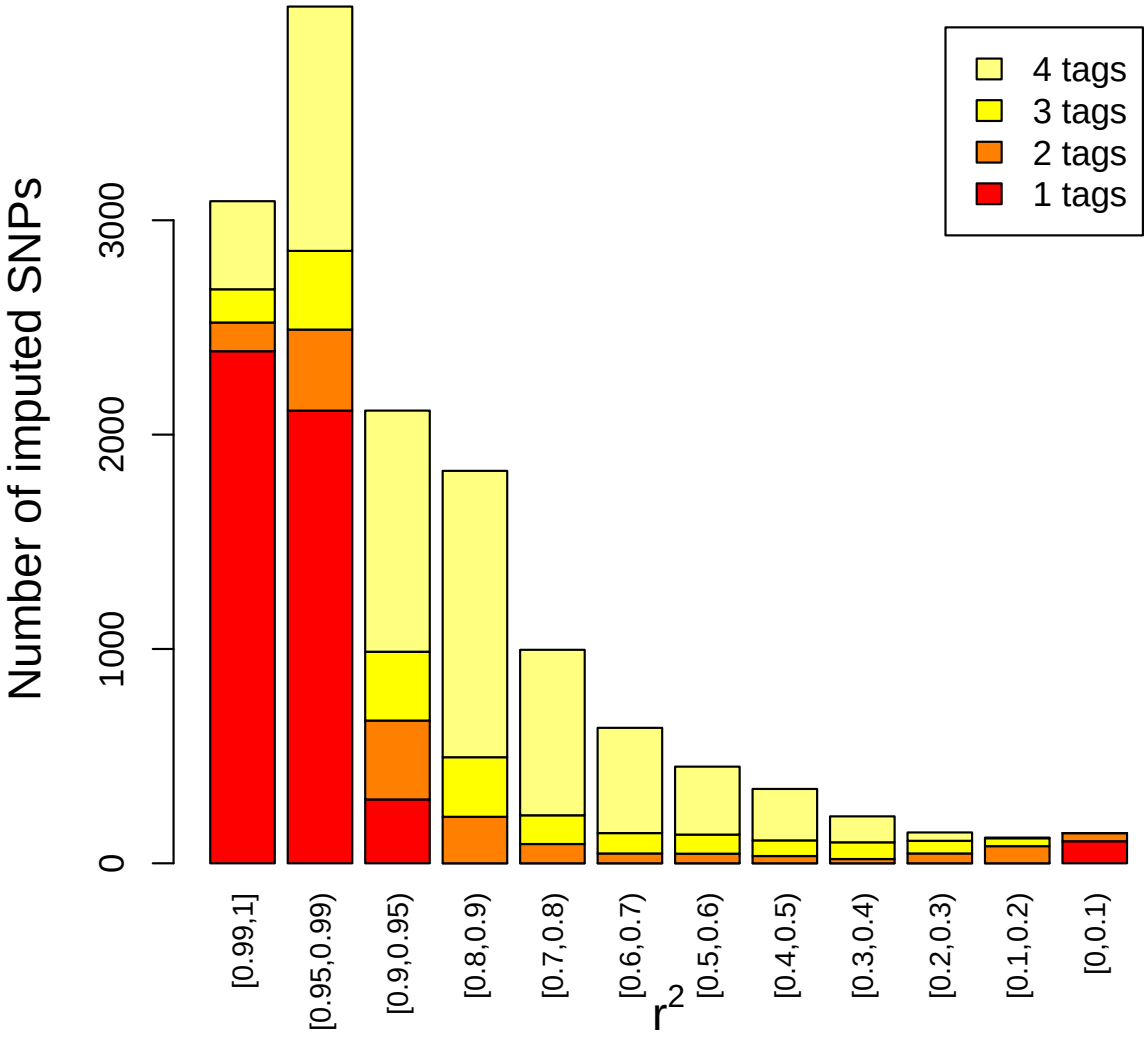
```
> summary(rules)
```

[2] For imputation from small samples, some smoothing of these haplotype frequencies would be advantageous and some ability to do this has been included. The `use.haps` argument to `snp.imputation` controls this. But invoking this option slows down the algorithm and it is not advised other than for very small sample sizes.

	SNPs used				
R-squared	1 tags	2 tags	3 tags	4 tags	<NA>
[0,0.1)	103	38	0	0	0
[0.1,0.2)	0	80	37	3	0
[0.2,0.3)	0	46	59	39	0
[0.3,0.4)	0	20	78	121	0
[0.4,0.5)	0	34	73	240	0
[0.5,0.6)	0	45	89	317	0
[0.6,0.7)	0	46	95	491	0
[0.7,0.8)	0	90	134	772	0
[0.8,0.9)	0	217	278	1336	0
[0.9,0.95)	298	368	321	1125	0
[0.95,0.99)	2112	378	367	1140	0
[0.99,1]	2389	134	155	411	0
<NA>	0	0	0	0	172

Columns represent the number of tag SNPs while rows represent grouping on R^2. The last column (headed `<NA>`) represents SNPs for which an imputation rule could not be computed, either because they were monomorphic or because there was insufficient data (as determined by the `minA` optional argument in the call to `snp.imputation`). The same information may be displayed graphically by

```
> plot(rules)
```



Carrying out the association tests

The association tests for imputed SNPs can be carried out using the function `single.snp.tests`.

```
> imp <- single.snp.tests(cc, stratum, data=subject.support,
+                         snp.data=target, rules=rules)
```

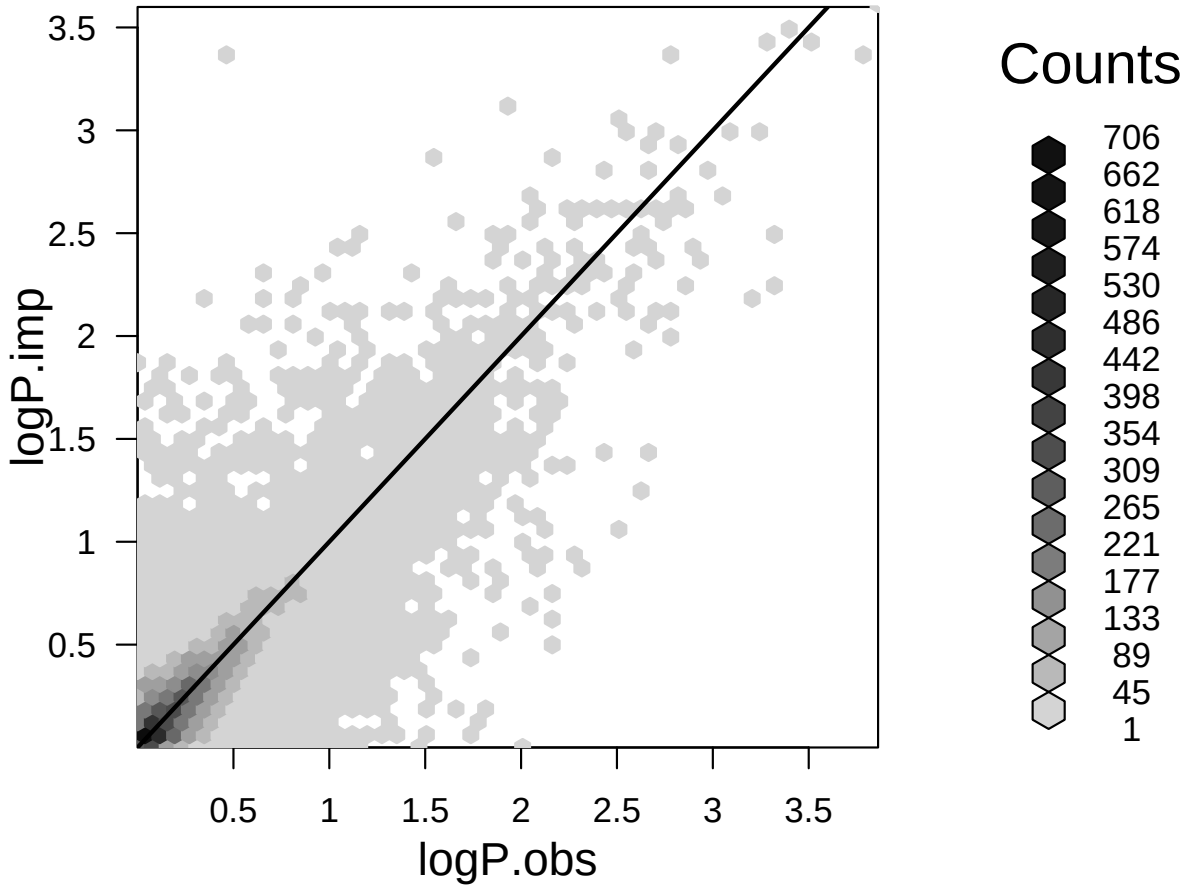
Using the observed data in the matrix `target` and the set of imputation rules stored in `rules`, the above command imputes each of the imputed SNPs, carries out 1- and 2-df

single locus tests for association, returns the results in the object `imp`. To see how successful imputation has been, we can carry out the same tests using the *true* data in `missing`:

```
> obs <- single.snp.tests(cc, stratum, data=subject.support, snp.data=lost)
```

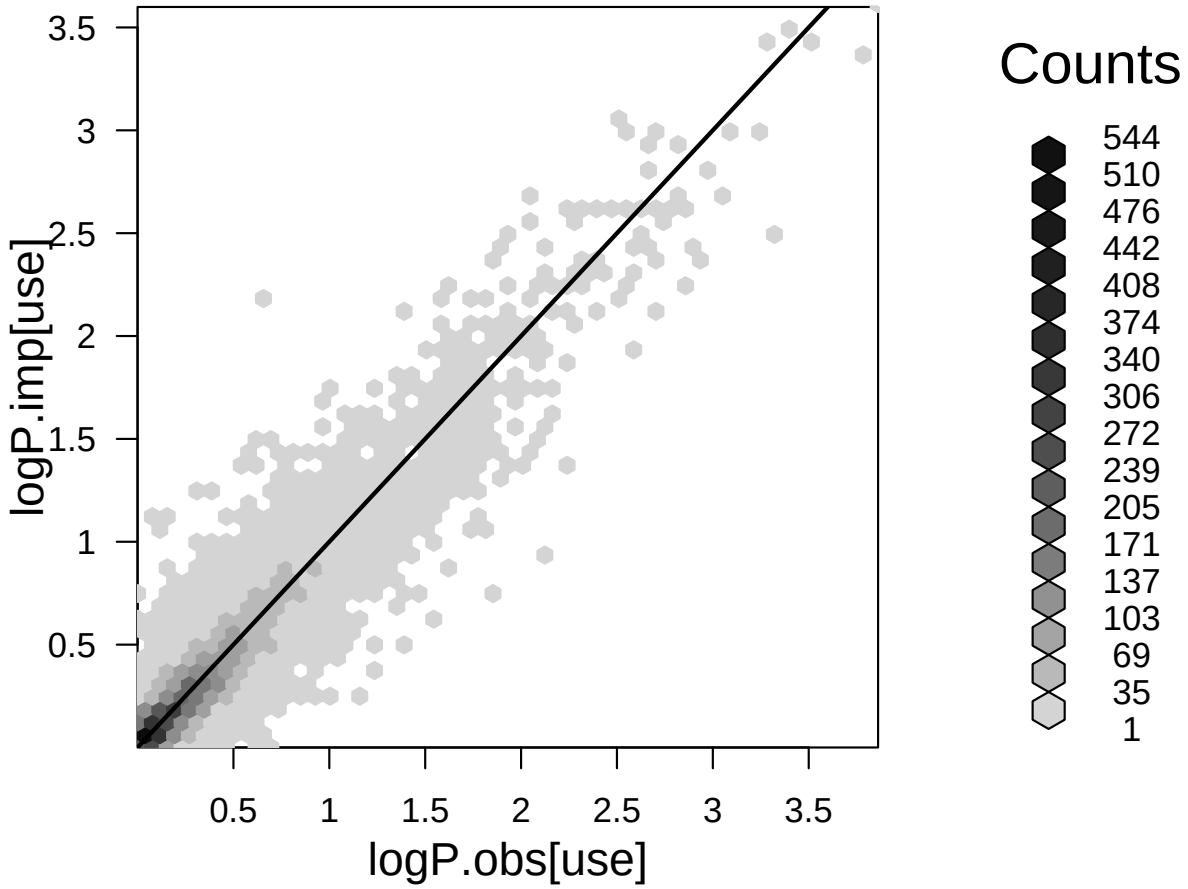
The next commands extract the p-values for the 1-df tests, using both the imputed and the true "missing" data, and plot one against the other (using the `hexbin` plotting package for clarity):

```
> logP.imp <- -log10(p.value(imp, df=1))
> logP.obs <- -log10(p.value(obs, df=1))
> hb <- hexbin(logP.obs, logP.imp, xbin=50)
> sp <- plot(hb)
> hexVP.abline(sp$plot.vp, 0, 1, col="black")
```



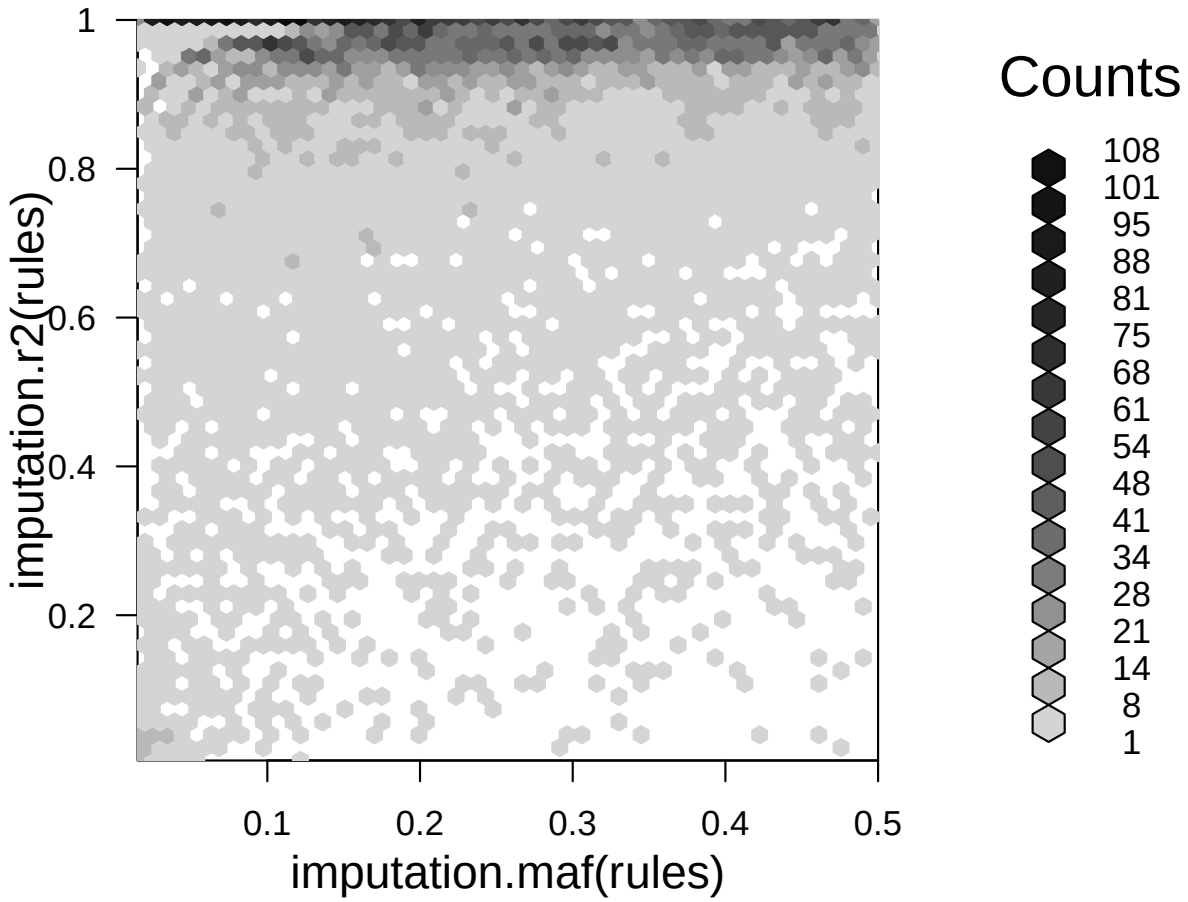
As might be expected, the agreement is rather better if we only compare the results for SNPs that can be computed with high R^2. The R^2 value is extracted from the `rules` object, using the function `imputation.r2` and used to select a subset of rules:

```
> use <- imputation.r2(rules)>0.9
> hb <- hexbin(logP.obs[use], logP.imp[use], xbin=50)
> sp <- plot(hb)
> hexVP.abline(sp$plot.vp, 0, 1, col="black")
```



Similarly, the function `imputation.maf` can be used to extract the minor allele frequencies of the imputed SNP from the `rules` object. Note that there is a tendency for SNPs with a high minor allele frequency to be imputed rather more successfully:

```
> hb <- hexbin(imputation.maf(rules), imputation.r2(rules), xbin=50)
> sp <- plot(hb)
```

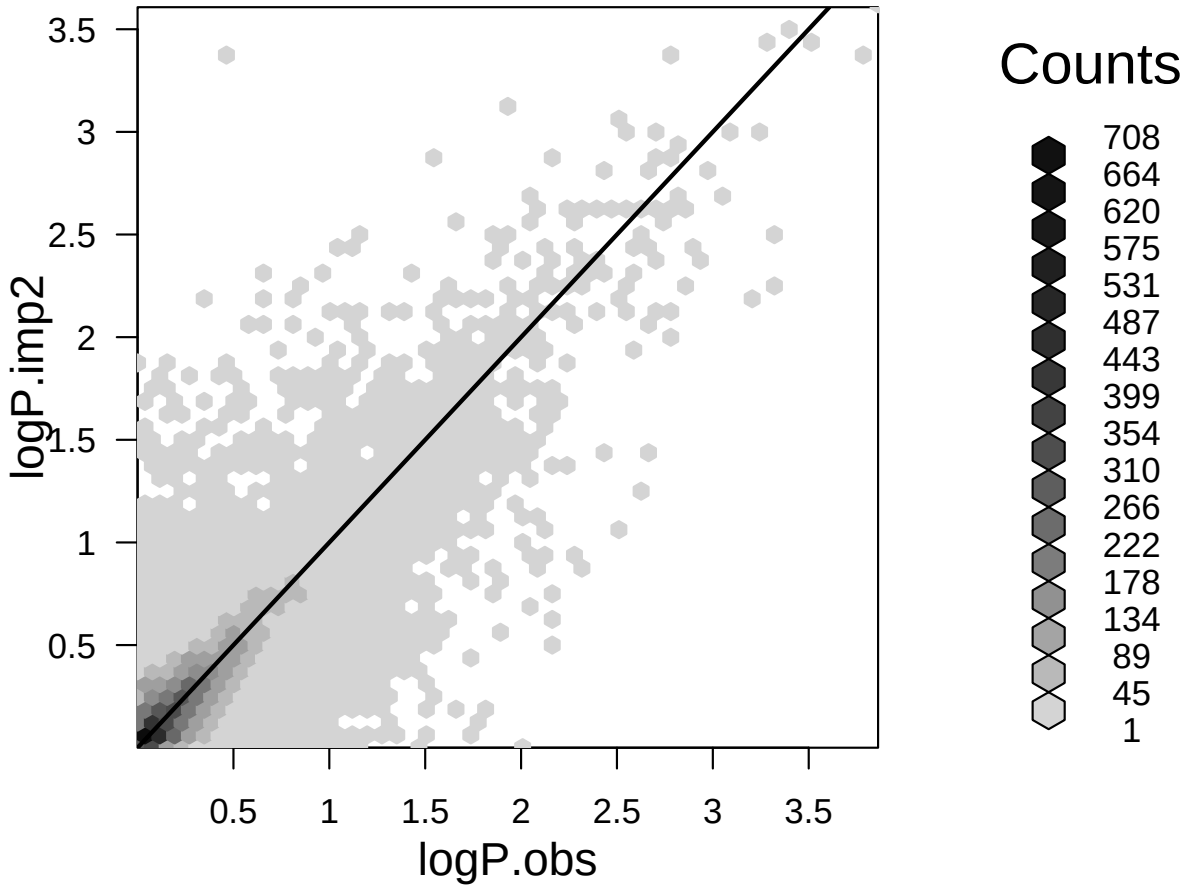


The function `snp.rhs.glm` also allows testing imputed SNPs. In its simplest form, it can be used to calculate essentially the same tests as carried out with `single.snp.tests`[3] (although, being a more flexible function, this will run somewhat slower). The next commands recalculate the 1 df tests for the imputed SNPs using `snp.rhs.tests`, and plot the results against those obtained when values are observed.

```
> imp2 <- snp.rhs.tests(cc~strata(stratum), family="binomial",
+                       data=subject.support, snp.data=target, rules=rules)
> logP.imp2 <- -log10(p.value(imp2))
```

[3]There is a small discrepancy, of the order of $(N-1):N$.

```
> hb <- hexbin(logP.obs, logP.imp2, xbin=50)
> sp <- plot(hb)
> hexVP.abline(sp$plot.vp, 0, 1, col="black")
```



Storing imputed genotypes

In the previous two sections we have seen how to (a) generate imputation rules and, (b) carry out tests on SNPs imputed according to these rules, but without storing the imputed genotypes. It is also possible to store imputed SNPs in an object of class `SnpMatrix` (or

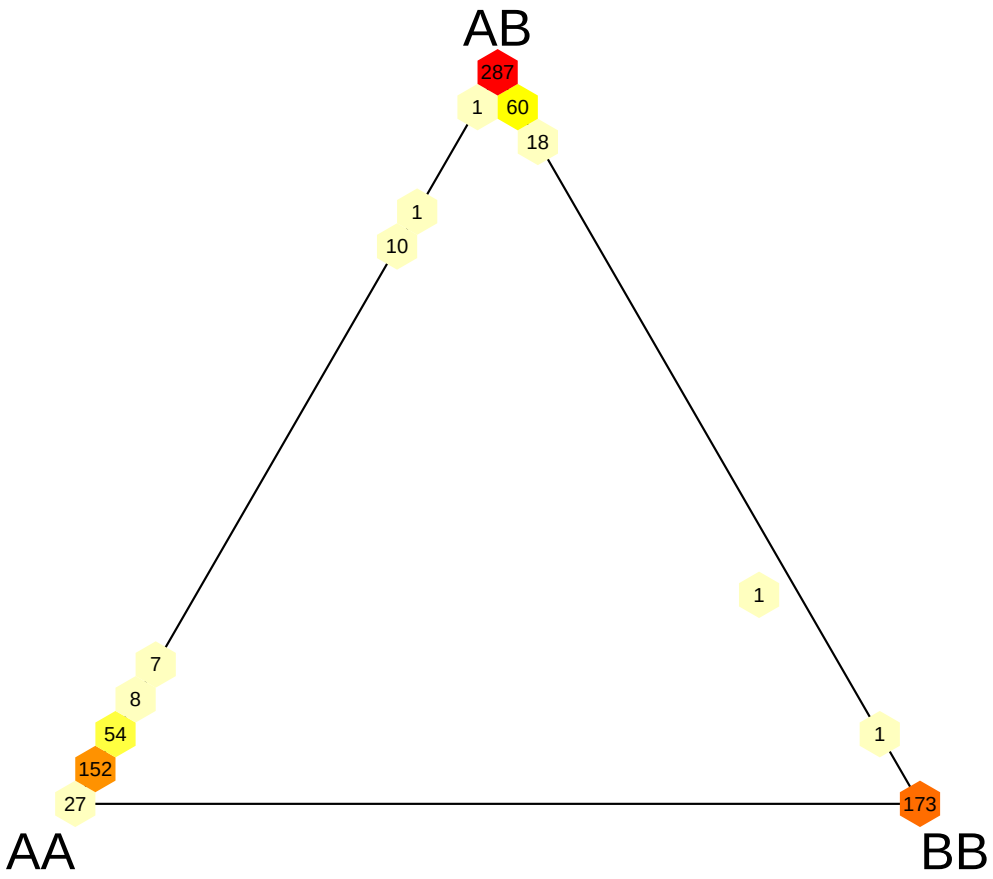
XSnpMatrix). The posterior probabilities of assignment of each individual to the three possible genotypes are stored within a one byte variable, although obviously not to full accuracy.

The following command imputes the "missing" SNPs using the "target" dataset and stores the imputed values in an object of class SnpMatrix:

```
> imputed <- impute.snps(rules, target, as.numeric=FALSE)
```

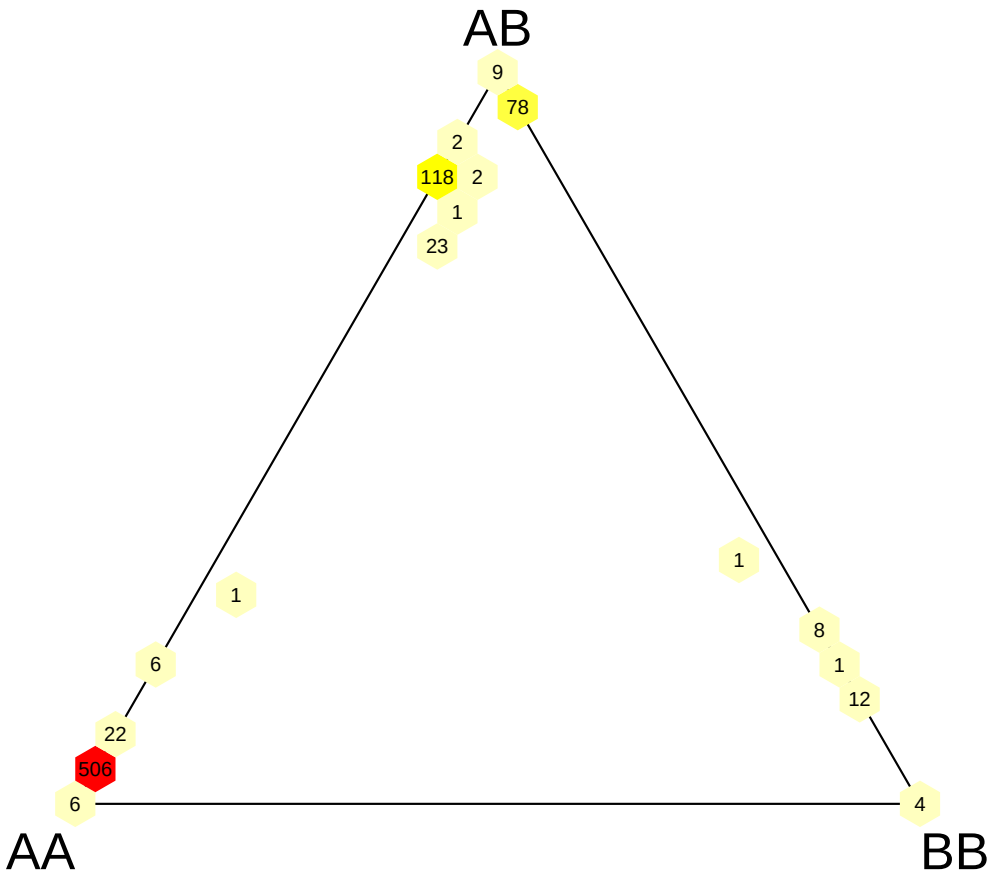
(If as.numeric were set to TRUE, the default, the resulting object would be a simple numeric matrix containing posterior expectations of the 0, 1, 2 genotype.) A nice graphical description of how snpStats stores uncertain genotypes is provided by the function plotUncertainty. This plots the frequency of the stored posterior probabilities on an equilateral triangle. The posterior probabilities are represented by the perpendicular distances from each side, the vertices of the triangle corresponding to certain assignments. Thus, the SNP rs4880568 is accurately imputed ($R^2 = 0.94$)

```
> plotUncertainty(imputed[, "rs4880568"])
```



while `rs2050968` is rather less so ($R^2 = 0.77$

```
> plotUncertainty(imputed[, "rs2050968"])
```



Tests can be carried out on these uncertainly assigned genotypes. For example

```
> imp3 <- single.snp.tests(cc, stratum, data=subject.support,
+                          snp.data=imputed, uncertain=TRUE)
```

The `uncertain=TRUE` argument ensures that uncertaing genotypes are used in the computations. This should yield nearly the same result as before. For the first five SNPs we have

```
> imp3[1:5]
```

```
             N Chi.squared.1.df Chi.squared.2.df     P.1df     P.2df
rs7909677  800       0.03221903        0.2323383 0.8575478 0.8903246
```

```
rs12773042 800       0.03495946        0.2489608 0.8516806 0.8829556
rs11253563 800       1.90998188        3.1810490 0.1669653 0.2038187
rs4881552  800       0.33317138        0.4541563 0.5637976 0.7968585
rs10904596 800       1.84110267        3.1246788 0.1748218 0.2096451
```

```
> imp[1:5]
```

```
             N     N.r2 Chi.squared.1.df Chi.squared.2.df     P.1df     P.2df
rs7909677  800 456.2311       0.01889424        0.1932950 0.8906701 0.9078760
rs12773042 800 494.5276       0.02482378        0.2128376 0.8748070 0.8990481
rs11253563 800 746.1419       1.83452359        3.1056449 0.1755942 0.2116498
rs4881552  800 776.9693       0.31397647        0.4381847 0.5752503 0.8032475
rs10904596 800 733.1828       1.80163867        3.0848916 0.1795145 0.2138574
```

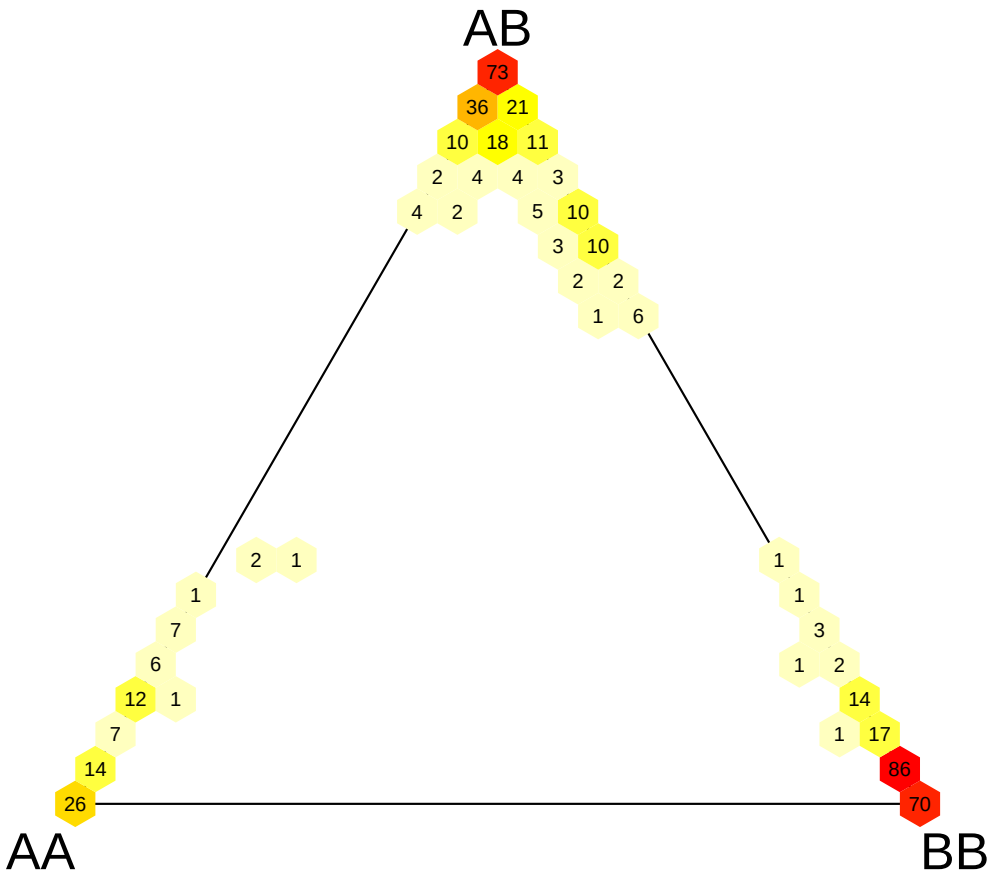
There are small discrepancies due to the genotype assignment probabilities not being stored to full accuracy. However these should have little effect on power of the tests and no effect on the type 1 error rate.

Note that the ability of `snpStats` to store imputed genotypes in this way allows alternative programs to be used to generate the imputed genotypes. For example, the file "mach1.out.mlprob.gz" (which is stored in the `extdata` sub-directory of the `snpStats` package) contains imputed SNPs generated by the MACH program, using the `-mle` and `-mldetails` options. In the following commands, we find the full path to this file, read it, and inspect one the imputed SNP in column 50:

```
> path <- system.file("extdata/mach1.out.mlprob.gz", package="snpStats")
> mach <- read.mach(path)
```

```
Reading MACH data from file /private/tmp/Rtmpnog21Q/Rinst1672e530c8393/snpStats/extdata/
Reading SnpMatrix with 500 rows and 178 columns
```

```
> plotUncertainty(mach[,50])
```



Meta-analysis

As stated at the beginning of this document, one of the main reasons that we need imputation is to perform meta-analyses which bring together data from genome-wide studies which use different platforms. The `snpStats` package includes a number of tools to facilitate this. All the tests implemented in `snpStats` are "score" tests. In the 1 df case we calculate a score defined by the first derivative of the log likelihood function with respect to the association parameter of interest at the parameter value corresponding to the null hypothesis of no

association. Denote this by U. We also calculate an estimate of its variance, also under the null hypothesis — V say. Then U^2/V provides the chi-squared test on 1 df. This procedure extends easily to meta-analysis; given two independent studies of the same hypothesis, we simply add together the two values of U and the two values of V, and then calculate U^2/V as before. These ideas also extend naturally to tests of several parameters (2 or more df tests).

In `snpStats`, the statistical testing functions can be called with the option `score=TRUE`, causing an extended object to be saved. The extended object contains the U and V values, thus allowing later combination of the evidence from different studies. We shall first see what sort of object we have calculated previously using `single.snp.tests` *without* the `score=TRUE` argument.

```
> class(imp)

[1] "SingleSnpTests"
attr(,"package")
[1] "snpStats"
```

This object contains the imputed SNP tests in our target set. However, these SNPs were observed in our training set, so we can test them. We will also recalculate the imputed tests. In both cases we will save the score information:

```
> obs <- single.snp.tests(cc, stratum, data=subject.support, snp.data=missing,
+                         score=TRUE)
> imp <- single.snp.tests(cc, stratum, data=subject.support,
+                         snp.data=target, rules=rules, score=TRUE)
```

The extended objects have been returned:

```
> class(obs)

[1] "SingleSnpTestsScore"
attr(,"package")
[1] "snpStats"

> class(imp)

[1] "SingleSnpTestsScore"
attr(,"package")
[1] "snpStats"
```

These extended objects behave in the same way as the original objects, so that the same functions can be used to extract chi-squared values, p-values etc., but several additional functions, or methods, are now available. Chief amongst these is `pool`, which combines evidence across independent studies as described at the beginning of this section. Although `obs` and `imp` are *not* from independent studies, so that the resulting test would not be valid, we can use them to demonstrate this:

```
> both <- pool(obs, imp)
> class(both)

[1] "SingleSnpTests"
attr(,"package")
[1] "snpStats"

> both[1:5]

              N     N.r2 Chi.squared.1.df Chi.squared.2.df     P.1df     P.2df
rs7909677   997 653.2311       0.07398108        0.1083750 0.7856261 0.9472545
rs12773042  998 692.5276       0.11321018        0.1427237 0.7365186 0.9311249
rs11253563  998 944.1419       1.44703256        2.2606014 0.2290047 0.3229361
rs4881552  1000 976.9693       0.33508643        1.1080666 0.5626793 0.5746275
rs10904596  995 928.1828       1.37378568        2.1663862 0.2411625 0.3385129
```

Note that if we wished at some later stage to combine the results in `both` with a further study, we would also need to specify `score=TRUE` in the call to `pool`:

```
> both <- pool(obs, imp, score=TRUE)
> class(both)

[1] "SingleSnpTestsScore"
attr(,"package")
[1] "snpStats"
```

Another reason to save the score statistics is that this allows us to investigate the *direction* of findings. These can be extracted from the extended objects using the function `effect.sign`. For example, this command tabulates the signs of the associations in `obs`:

```
> table(effect.sign(obs))

  -1    0    1
7126   32 7093
```

In this table, -1 corresponds to tests in which effect sizes were negative (corresponding to an odds ratio less than one), while +1 indicates positive effect sizes (odds ratio greater than one). Zero sign indicates that the effect was `NA` (for example because the SNP was monomorphic). Reversal of sign can be the explanation of a puzzling phenomenon when two studies give significant results individually, but no significant association when pooled. Although it is not impossible that such results are genuine, a more usual explanation is that the two alleles have been coded differently in the two studies: allele 1 in the first study is allele 2 in the second study and vice versa. To allow for this, `snpStats` provides the `switch.alleles` function, which reverses the coding of specified SNPs. It can be applied to `SnpMatrix` objects but, because allele switches are often discovered quite late on in the

analysis and recoding the original data matrices could have unforeseen consequences, the `switch.alleles` function can also be applied to the extended test output objects. This modifies the saved scores *as if* the allele coding had been switched in the original data. The use of this is demonstrated below.

```
> effect.sign(obs)[1:6]

 rs7909677 rs12773042 rs11253563  rs4881552 rs10904596  rs4880781
         1         -1          1         -1          1          1

> sw.obs <- switch.alleles(obs, 1:3)
> class(sw.obs)

[1] "SingleSnpTestsScore"
attr(,"package")
[1] "snpStats"

> effect.sign(sw.obs)[1:6]

 rs7909677 rs12773042 rs11253563  rs4881552 rs10904596  rs4880781
        -1          1         -1         -1          1          1
```