R-package "simctest"
R-class "mmctest"
A Short Introduction

Axel Gandy and Georg Hahn

March 23, 2017

This document describes briefly how to use the class "mmctest", included in the R-package "simctest". It implements the methods from "MMCTest-A Safe Algorithm for Implementing Multiple Monte Carlo Tests", based on Gandy and Hahn [2014].

The class can be used to evaluate the statistical significance of each hypothesis in a multiple testing problem.

1 Installation

The functions described in this document are included in the R-package "simctest". Please see the documentation of "simctest" on how to install the package.

2 Usage

The package is loaded by typing

```
> library(simctest)
```

This document can be accessed via

```
> vignette("simctest-mmctest-intro")
```

Documentation of the most useful commands can be obtained as follows:

```
> ? simctest
> ? mcp
> ? mmctest
```

2.1 Implementing a Monte Carlo multiple testing problem

The following is an artificial example. Implementing a Monte Carlo multiple testing problem consists of two stages.

Firstly, an interface to draw samples has to be provided. This can be done in two ways, either by implementing the generic class `mmctSamplerGeneric` or by directly providing the number m of hypotheses and a function f which generates samples. Both ways are described in the next section.

Secondly, an object of type `mmctest` has to be created. It provides a "run" method which uses the `mmctest`-object and an `mmctSamplerGeneric`-object to evaluate the multiple testing problem.

The algorithm used in class `mmctest` is the one introduced in Gandy and Hahn [2014]. The multiple testing problem is evaluated until at least one of four stopping criteria is satisfied, see below for a detailed description.

Stopped tests can be resumed with the "cont" function.

Printing an object of type `mmctest` or `mmctestres` will display the number of already rejected and non-rejected hypotheses, the number of undecided hypotheses and the total number of samples drawn up to the current stage.

2.1.1 Implementing the sampling interface

An interface for drawing new samples has to be provided for each multiple testing problem.

If new samples are simply generated by a function f, the derived class `mmctSampler` provided in `simctest` can be used as a shortcut. It works as follows: Any function f used to draw new samples has to be able to accept the arguments "ind", a vector with indices of hypotheses and a vector "n" containing the number of samples to be drawn for each hypothesis in vector "ind".

The function f has to return a vector containing the number of significant test statistics for each hypothesis specified in "ind".

For instance, passing a vector "ind" of $(2, 5)$ and a vector "n" of $(5, 10)$ as arguments means that 5 more samples are requested for the hypothesis with index 2 and 10 more for the hypothesis with index 5. The function f might need further data to evaluate the tests. Such data can be passed on to f as third argument `data`.

For instance,

```
> fun <- function(ind,n,data)
+   sapply(1:length(ind), function(i) sum(runif(n[i])<=data[ind[i]]));
```

is a function which draws samples from hypotheses having p-values given in vector `data`.

The package `mmctest` provides a shortcut which can be used to easily specify the interface. Given a function `fun` to draw samples and the number `num` of hypotheses, `fun` and `num` (and additional data `data`) can be passed on to the class `mmctSampler` which returns a derived object of the generic class `mmctSamplerGeneric`. For example,

```
> s <- mmctSampler(fun,num=500,data=c(rep(0,100),runif(400)));
```

returns an sampler interface `s` for the function `fun` defined above and 500 p-values used to draw new samples.

The class `mmctSamplerGeneric` can also directly be overwritten with an own sampler interface. Any sampler has to implement the two generic functions `getSamples` and `getNumber`:

```
> # class mmctSampler1, inherited from mmctSamplerGeneric
> setClass("mmctSampler1", contains="mmctSamplerGeneric",
+   representation=representation(data="numeric"))
> # get n[i] new samples for every index i in ind
```

```
> setMethod("getSamples", signature(obj="mmctSampler1"),
+   function(obj, ind, n) {
+     sapply(1:length(ind),
+     function(i) { return(sum(runif(n[i])<=
+     obj@data[ind[i]])); });
+   }
+ )

[1] "getSamples"

> # get number of hypotheses
> setMethod("getNumber", signature(obj="mmctSampler1"),
+   function(obj) {
+     return(length(obj@data));
+   }
+ )

[1] "getNumber"
```

In this case, the sampler will be

```
> s <- new("mmctSampler1", data=c(rep(0,100),runif(400)));
```

2.1.2 A simple run of the algorithm

After having specified the sampler, the main algorithm can be executed. This is done by creating an object of type `mmctest` using the pseudo-constructor

```
mmctest(epsilon=0.01, threshold=0.1, r=10000, h, thompson=F,
                              R=1000),
```

where `epsilon` is the overall error on the classification being correct one is willing to spend (see Gandy and Hahn [2014]) and `threshold` is the multiple testing threshold. The `MMCTest` algorithm uses a "spending sequence" which controls how the overall error probability is spent on each of the m hypotheses in each iteration (see Gandy and Hahn [2014]). The parameter `r` with default value $r = 10000$ controls after which number of samples half the error probability has been spent and can be chosen arbitrarily. The function h is the multiple testing procedure.

Thompson sampling can be used to efficiently allocate each new batch of samples per iteration, see Gandy and Hahn [2015]: it can be activated using the switch `thompson` (default is *false*), where the parameter R determines the accuracy (default value 1000 repetitions) with which weights are computed [Gandy and Hahn, 2015]. The coming subsubsections contain further details.

Any function

```
h <- function(p, threshold) ...
```

can be used as a multiple testing procedure as long as it takes a vector p of p-values and a threshold `threshold` as arguments and returns the indices of all rejected hypotheses as vector.

The Benjamini-Hochberg procedure `hBH`, its modification by Pounds and Cheng [2006] `hPC` and the Bonferroni correction `hBonferroni` are available by default:

```
> s <- mmctSampler(fun,num=500,data=c(rep(0,100),runif(400)));
> m <- mmctest(h=hBH);
```

The algorithm can now be started by calling

```
run(alg, gensample, maxsteps=list(maxit=0, maxnum=0, undecided=0,
                          elapsedsec=0))
```

which takes an object `alg` of type `mmctest`, a sampler object `gensample` to generate samples and a list `maxsteps` as stopping criterion. The list `maxsteps` can include a maximal number of iterations `maxit` after which the algorithm stops, a maximal total number of samples `maxnum` drawn until stopping, a number `undecided` of undecided hypotheses one is willing to tolerate or a time constraint `elapsedsec` in seconds.

Specifying other items in list `maxsteps` will lead to an error message and an empty list will be reset to the default list

```
list(maxit=0, maxnum=0, undecided=0, elapsedsec=0).
```

As an example, the following lines evaluate the previously created multiple testing problem `m` using the Benjamini-Hochberg procedure `hBH` and the previous sampler `s`. The algorithm stops before reaching more than a total of 1000000 samples or after all but 20 hypotheses are classified:

```
> m <- run(m, s, maxsteps=list(maxnum=1000000,undecided=20));
> m

Number of rejected hypotheses: 101
Number of non-rejected hypotheses: 380
Number of undecided hypotheses: 19
Total number of samples: 145965
```

Printing the object displays the number of already rejected and non-rejected hypotheses, the number of undecided hypotheses and the total number of samples drawn up to the current stage.

A formatted summary of the indices belonging to rejected and undecided hypotheses can be printed via `summary.mmctestres`. All indices not printed belong to non-rejected hypotheses.

```
> summary.mmctestres(m)

Number of hypotheses: 500
Indices of rejected hypotheses: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62
63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85
86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 392
Indices of undecided hypotheses: 180 193 204 209 219 291 309 322 325
343 359 400 427 457 461 464 475 492 495
All hypotheses not listed are classified as not rejected.
```

2.1.3 Continuing a run of the algorithm

Each run can be continued with the `cont` function using a new stopping criterion:

```
> m <- cont(m, steps=list(undecided=10));
> m

Number of rejected hypotheses: 107
Number of non-rejected hypotheses: 385
Number of undecided hypotheses: 8
Total number of samples: 189361
```

Here, the algorithm has been applied again to the previously stopped multiple testing problem `m`. It has been resumed until all but 10 hypotheses were classified.

As before, `maxit`, `maxnum`, `undecided` and `elapsedsec` are valid stopping criteria for parameter `steps` of function `cont`.

2.1.4 Requesting the test result

The current test result can be requested from any `mmctestres` object. Calling `testResult` of class `mmctestres` will return a list containing indices of rejected hypotheses (vector '$rejected'), nonrejected hypotheses (vector '$nonrejected') and undecided hypotheses (vector '$undecided'). For the previously continued run of object `m`, the result of the computation can be requested as follows:

```
> res <- testResult(m);
> res$undecided

[1] 180 193 291 322 400 427 461 464

> length(res$rejected)

[1] 107

> length(res$nonrejected)

[1] 385
```

In the example above, the current computation result of object `m` is stored in variable `res`. For object `m`, the algorithm has been run until all but (at least) 10 hypotheses have been classified. The indices of the undecided hypotheses as well as the number of rejected and nonrejected hypotheses (i.e. the length of the vectors containing their indices) are displayed.

2.1.5 Confidence intervals and estimates of p-values

At any stage, p-values can be estimated based on the total number of samples drawn for each hypothesis during intial or continued runs:

```
> estimate <- pEstimate(m);
> lastindex <- length(estimate);
> estimate[lastindex]
```

```
[1] 0.3382353
```

The function `pEstimate` takes an object of type `mmctest` as argument and returns a vector containing estimates of all p-values.

Similarly, the current confidence limits for the exact (Clopper-Pearson) confidence intervals can be requested:

```
> l <- confidenceLimits(m);
> l$lowerLimits[lastindex]

[1] 0.08617203

> l$upperLimits[lastindex]

[1] 0.6636927
```

The function `confidenceLimits` takes an object of type `mmctest` as argument and returns a list containing lower confidence limits (vector 'lowerLimits') and upper confidence limits (vector 'upperLimits') for each p-value.

2.1.6 Switching to QuickMMCTest

The QuickMMCTest algorithm presented in Gandy and Hahn [2015] is included as a special case of MMCTest.

To activate MMCTest, please set the parameter `thompson` in the `mmctest` constructor to TRUE.

The constructor contains a further value R (default value $R = 1000$) which controls the accuracy with which weights are computed in QuickMMCTest, see Gandy and Hahn [2015] for an explanation.

Apart from setting `thompson` to TRUE, nothing needs to be done to use QuickMMCTest.

Typically, QuickMMCTest is run using a maximal total effort and a maximal number of iterations as stopping criteria: if these two are specified in `maxit` and `maxnum` (see "A simple run of the algorithm"), QuickMMCTest spreads out the total number of samples over all `maxit` iterations and uses the weights to allocate samples.

If QuickMMCTest is run without a maximal total effort and a maximal number of iterations, it allocates the batch of samples that MMCTest would have allocated, which is geometrically increased in each iteration, see Gandy and Hahn [2014]. QuickMMCTest thus runs until the algorithm is stopped manually or the desired number of undecided hypotheses is reached.

2.1.7 Empirical rejection probabilities

When using QuickMMCTest, hypotheses can be classified after termination using three methods: `summary.mmctestres` provides sets of rejected, non-rejected and undecided hypotheses as in MMCTest, `pEstimates` provides estimated p-values computed with a pseudo-count which can be plugged into the multiple testing procedure, and finally

```
> rej <- rejProb(m)>0.5;
> rej[1]
```

```
[1] TRUE
```

provides empirical rejection probabilities for QuickMMCTest as used in Gandy and Hahn [2015].

These are obtained by drawing R sets of all the m p-values from the Beta posteriors for all p-values, by evaluating the multiple testing procedure on each set of p-values and by recording the number of times each hypothesis is rejected based on the sampled p-values.

The resulting proportion of rejections out of R repetitions are reported by the method `rejProb` (as a vector of length m, one number between 0 and 1 per hypothesis) and give an indicator of how stable the decision on each hypothesis is: i.e. proportions close to one indicate a very stable decision that a hypothesis is rejected, likewise proportions close to zero indicate non-rejections and values close to 0.5 indicate very unstable decisions.

By thresholding them against, e.g. 0.5, empirical rejections can be obtained as demonstrated in the above code example (all hypotheses with a rejection probability above 0.5 are rejected). The threshold 0.5 is arbitrary and can be replaced by higher (lower) values to be more (less) conservative.

2.1.8 An extended example

In this extended example a permutation test is used to determine if two groups A and B have equal means. This is done in $n_{groups} = 20$ cases. Each group has size 4 and both groups A and B are stored together in one row of length $n = 8$ in a matrix G.

```
> n <- 8;
> ngroups <- 20;
> G <- matrix(rep(0,n*ngroups), nrow=ngroups);
> for(j in 1:(ngroups/2)) G[j,] <- c(rnorm(n/2,mean=0,sd=0.55),rnorm(n/2,mean=1,sd=0.55));
> for(j in (ngroups/2+1):ngroups) G[j,] <- rnorm(n,mean=0,sd=3);
```

To implement this test as a Monte-Carlo test, we start by overwriting the generic class `mmctSamplerGeneric` to specify the sampler. The data stored in an `ExSampler` object is the matrix G.

```
> # class ExSampler, inherited from mmctSamplerGeneric
> setClass("ExSampler", contains="mmctSamplerGeneric",
+   representation=representation(data="matrix"))
> setMethod("getSamples", signature(obj="ExSampler"),
+   function(obj, ind, n) {
+     sapply(1:length(ind), function(i) {
+       v <- obj@data[ind[i],];
+       s <- matrix(rep(v,n[i]+1), byrow=T, ncol=length(v));
+       for(j in 1:n[i]) s[j+1,] <- sample(v);
+       means <- abs(rowMeans(s[,1:(length(v)/2)])-
+         rowMeans(s[,(length(v)/2+1):length(v)]));
+       return(sum(means>means[1]));
+     });
+   }
+ )
```

```
[1] "getSamples"
```

```
> setMethod("getNumber", signature(obj="ExSampler"),
+   function(obj) {
+     return(length(obj@data[,1]));
+   }
+ )
```

```
[1] "getNumber"
```

The `getSamples` method generates `n[i]` permutations of each row `i` in the indices vector `ind` and counts how many times the generated means exceeded the data mean (stored in row 1).

The sampler is then

```
> exsampler <- new("ExSampler", data=G);
```

As before, the multiple testing problem is set up by creating an object of type `mmctest` using `hBH` as multiple testing procedure and the `exsampler` object as sampler interface. The constructor `mmctest` uses a default threshold of 0.1.

```
> m <- mmctest(h=hBH);
> m <- run(m, exsampler, maxsteps=list(undecided=0));
```

Algorithm `mmctest` has been run until all hypotheses were classified. Based on this run, the following hypotheses are rejected:

```
> testResult(m)$rejected
```

```
[1] 1 4
```

Estimates for p-values are

```
> pEstimate(m)
```

```
 [1] 0.0006915629 0.1590909091 0.0873362445 0.0006915629 0.1454545455
 [6] 0.4130434783 0.0639032815 0.1545454545 0.0296922800 0.0708117444
[11] 0.3382352941 0.2210526316 0.6785714286 0.2941176471 0.6071428571
[16] 0.9230769231 0.7692307692 0.3913043478 0.2647058824 0.0262731690
```

To verify this result, exact p-values are computed by enumerating all permutations of each row. This is done using algorithm `QuickPerm`.

Based on the exact p-values, given by

```
> pexact
```

```
 [1] 0.00000000 0.11428571 0.08571429 0.00000000 0.08571429 0.40000000
 [7] 0.05714286 0.11428571 0.02857143 0.05714286 0.25714286 0.14285714
[13] 0.68571429 0.28571429 0.82857143 0.94285714 0.65714286 0.37142857
[19] 0.22857143 0.02857143
```

the Benjamini-Hochberg procedure at threshold 0.1 will give the following set of rejections:

```
> which(hBH(pexact, threshold=0.1))
```

```
[1] 1 4
```

References

A. Gandy and G. Hahn. MMCTest - A Safe Algorithm for Implementing Multiple Monte Carlo Tests. *Scandinavian Journal of Statistics*, 41(4):1083–1101, 2014.

A. Gandy and G. Hahn. Quickmmctest – Higher accuracy for Monte-Carlo based multiple testing. 2015. arXiv:1402.2706.

S. Pounds and C. Cheng. Robust estimation of the false discovery rate. *Bioinformatics*, 22(16):1979–1987, 2006. doi: 10.1093/bioinformatics/btl328.