SRS: A Subject Randomization System

Balasubramanian Narasimhan

Revision of *Date*

Contents

1 Introduction

SRS is a Subject Randomization System based on the paper by Pocock and Simon ([1]. It follows the development in the paper rather closely. In this vignette we show how one might use the system in designing and implementing randomizations for clinical trials.

This vignette has two parts to it. The first part goes into detail discussing some of the innards of the package. This is most meaningful to those in our Biostatistics core who may recommend this software for use in trials. The second part is more of a HOWTO for conducting a trial.

This package is written using S4 classes. No deep knowledge of S4 classes is assumed in what follows.

To use the package, we first attach it.

```
> library(SRS)
```

2 The Basic Classes

There are two main classes that most users of the package will use: `ClinicalExperiment` and `PocockSimonRandomizer`. The class `ClinicalExperiment`, as the name implies, encapsulates the characteristics of a clinical experiment. An instance of this class is used to create an instance of the

other class `PocockSimonRandomizer` so that the randomizer remains associated with a particular clinical experiment.

3 A Simple Example

3.1 A Clinical Experiment

Let us create a simple clinical experiment object after invoking the requisite package. The function `ClinicalExperiment` (as distinct from the `ClinicalExperiment` class) is available for us.

```
> expt0 <- ClinicalExperiment(number.of.factors = 3, number.of.factor.levels = c(2,
+     2, 3), number.of.treatments = 3)
```

This create an experiment with three factors and three treatments. The first factor has 2 levels, the second 2, and the third 3. If none of the arguments are specified, the default is to create a two-factor, two-treatment experiment with each factor having two levels. One can name the factors with the argument `factor.names` but default names such as $F_1, F_2, \ldots$ are provided. The levels are currently indicated by the suffixes `-1`, `-2`, etc., that are attached to the factor names; a flexible naming scheme for this might be introduced later.

It is useful to print the object to see what it contains.

```
> print(expt0)

An object of class "ClinicalExperiment"
Slot "number.of.factors":
[1] 3

Slot "factor.names":
[1] "F1" "F2" "F3"

Slot "factor.level.names":
[[1]]
[1] "1" "2"

[[2]]
[1] "1" "2"

[[3]]
[1] "1" "2" "3"


Slot "number.of.factor.levels":
[1] 2 2 3

Slot "number.of.treatments":
[1] 3

Slot "treatment.names":
[1] "Tr1" "Tr2" "Tr3"
```

Of course, in anything other than a toy setting, one actually provides some names for the factor and factor levels. We'll use this in what follows.

```
> expt <- ClinicalExperiment(number.of.factors = 3, factor.names = c("Sex",
+     "Race", "Stage"), number.of.factor.levels = c(2, 2, 3), factor.level.names = list(c("Fema
+     "Male"), c("Caucasian", "Non-caucasian"), c("I", "II", "III")),
+     number.of.treatments = 3, treatment.names <- c("Placebo",
+         "Arm1", "Arm2"))
> print(expt)

An object of class "ClinicalExperiment"
Slot "number.of.factors":
[1] 3

Slot "factor.names":
[1] "Sex"   "Race"  "Stage"

Slot "factor.level.names":
[[1]]
[1] "Female" "Male"

[[2]]
[1] "Caucasian"     "Non-caucasian"

[[3]]
[1] "I"   "II"  "III"


Slot "number.of.factor.levels":
[1] 2 2 3

Slot "number.of.treatments":
[1] 3

Slot "treatment.names":
[1] "Placebo" "Arm1"    "Arm2"
```

3.2 The PocockSimon Randomizer

Now let's create a randomizer that will work for this experiment.

```
> r.obj <- new("PocockSimonRandomizer", expt, as.integer(12345))
> print(r.obj)

An object of class "PocockSimonRandomizer"
Slot "expt":
An object of class "ClinicalExperiment"
Slot "number.of.factors":
[1] 3
```

```
Slot "factor.names":
[1] "Sex"   "Race"  "Stage"

Slot "factor.level.names":
[[1]]
[1] "Female" "Male"

[[2]]
[1] "Caucasian"     "Non-caucasian"

[[3]]
[1] "I"   "II"  "III"


Slot "number.of.factor.levels":
[1] 2 2 3

Slot "number.of.treatments":
[1] 3

Slot "treatment.names":
[1] "Placebo" "Arm1"    "Arm2"


Slot "seed":
[1] 12345

Slot "stateTable":
        Sex:Female Sex:Male Race:Caucasian Race:Non-caucasian Stage:I Stage:II
Placebo          0        0              0                  0       0        0
Arm1             0        0              0                  0       0        0
Arm2             0        0              0                  0       0        0
        Stage:III
Placebo         0
Arm1            0
Arm2            0

Slot "tr.assignments":
data frame with 0 columns and 0 rows

Slot "tr.ratios":
[1] 0.3333333 0.3333333 0.3333333

Slot "d.func":
function (x)
{
    diff(range(x))
```

```
}

Slot "g.func":
function (x)
{
    sum(x)
}

Slot "p.func":
function (overallImbalance)
{
    number.of.treatments <- length(overallImbalance)
    p.star <- 2/3
    k <- which(overallImbalance == min(overallImbalance))
    if (length(k) > 1) {
        k <- sample(k, 1)
    }
    p.vec <- rep((1 - p.star)/(number.of.treatments - 1), number.of.treatments)
    p.vec[k] <- p.star
    p.vec
}
```

Note that we don't have a helper constructor function (for no particular reason) and so we had to use the `new` function to create the object. (Indeed, that is what the `ClinicalExperiment` function does behind the scenes.)

The output of the print above indicates that there are some default settings for the randomizer. For example, the treatment ratios are all 1's indicating equal treatment preference; others such as `1 2 1` could have been specified. Note the `stateTable` slot which will summarize the margins of the factor distributions by treatment. Since no randomization has been done, the slot `tr.assignments` is empty.

Of interest are the slots named `d.func`, `g.func` and `p.func`. The `d.func` computes imbalance due to assigning each of the treatments, `g.func` computes the overall imbalance, and the `p.func` computes the probabilities of assigning each treatment based on the overall imbalance. All of these can be changed by the user. Default values for these functions are the ones described in [1].

3.3 Using the Randomizer

Now that we have defined the experiment and the randomizer, we can randomize several subjects using these classes. First some helper functions that are useful in simulations.

```
> generateId <- function(i) {
+     if (i < 0 || i > 10000) {
+         stop("generateId: Arg expected to be between 1 and 9999")
+     }
+     zero.count <- 5 - trunc(log10(i)) - 1
+     prefix <- substring(10^zero.count, 2)
+     paste("ID.", prefix, i, sep = "")
+ }
> generateRandomFactors <- function(factor.levels) {
```

```
+     unlist(lapply(factor.levels, function(x) sample(x, 1)))
+ }
```

Now, we will run a 10 randomizations and print the results.

```
> for (i in 1:10) r.obj <- randomize(r.obj, generateId(i), generateRandomFactors(expt@factor.le
> print(r.obj@tr.assignments)
```

```
            Sex          Race Stage Treatment
ID.00001   Male Non-caucasian   III      Arm2
ID.00002 Female     Caucasian    II   Placebo
ID.00003 Female     Caucasian   III      Arm1
ID.00004 Female     Caucasian    II      Arm2
ID.00005 Female Non-caucasian    II      Arm1
ID.00006   Male Non-caucasian    II   Placebo
ID.00007   Male Non-caucasian     I      Arm1
ID.00008   Male     Caucasian     I      Arm2
ID.00009 Female Non-caucasian   III   Placebo
ID.00010 Female Non-caucasian    II   Placebo
```

Just in case we are only interested in the last assigned treatment:

```
> lastRandomization(r.obj)
```

```
            Sex          Race Stage Treatment
ID.00010 Female Non-caucasian    II   Placebo
```

We can also look at the marginal distributions thus:

```
> print(r.obj@stateTable)
```

```
        Sex:Female Sex:Male Race:Caucasian Race:Non-caucasian Stage:I Stage:II
Placebo          3        1              1                  3       0        3
Arm1             2        1              1                  2       1        1
Arm2             1        2              2                  1       1        1
        Stage:III
Placebo         1
Arm1            1
Arm2            1
```

4 Customizing the Randomizer

The functions for computing imbalance, overall imbalance and probabilities can all be customized. These are best illustrated by additional examples.

4.1 A different imbalance function

Let's move away from the default range function to say the standard deviation (`sd`) function.

```
> r.obj.2 <- new("PocockSimonRandomizer", expt, as.integer(12345),
+     d.func = sd)
> print(r.obj.2@d.func)

function (x, na.rm = FALSE)
{
    if (is.matrix(x))
        apply(x, 2, sd, na.rm = na.rm)
    else if (is.vector(x))
        sqrt(var(x, na.rm = na.rm))
    else if (is.data.frame(x))
        sapply(x, sd, na.rm = na.rm)
    else sqrt(var(as.vector(x), na.rm = na.rm))
}
<environment: namespace:stats>
```

Now let's run that simulation again.

```
> for (i in 1:10) r.obj.2 <- randomize(r.obj.2, generateId(i),
+     generateRandomFactors(expt@factor.level.names))
> print(r.obj.2@tr.assignments)

            Sex          Race Stage Treatment
ID.00001   Male Non-caucasian   III      Arm2
ID.00002 Female     Caucasian    II   Placebo
ID.00003 Female     Caucasian   III      Arm1
ID.00004 Female     Caucasian    II      Arm2
ID.00005 Female Non-caucasian    II      Arm1
ID.00006   Male Non-caucasian    II   Placebo
ID.00007   Male Non-caucasian     I      Arm1
ID.00008   Male     Caucasian     I      Arm2
ID.00009 Female Non-caucasian   III   Placebo
ID.00010 Female Non-caucasian    II   Placebo
```

Now print the summaries.

```
> print(table(r.obj@tr.assignments[, "Treatment"]))

   Arm1    Arm2 Placebo
      3       3       4

> print(r.obj@stateTable)

        Sex:Female Sex:Male Race:Caucasian Race:Non-caucasian Stage:I Stage:II
Placebo          3        1              1                  3       0        3
Arm1             2        1              1                  2       1        1
Arm2             1        2              2                  1       1        1
        Stage:III
Placebo         1
Arm1            1
Arm2            1
```

4.2 Weighting factors differently

Now let's weight imbalance on factor 1 more than the others by a factor of 5. We do this by modifying the `g.func`.

```
> g.func <- function(imbalances) {
+     factor.weights <- c(5, 1, 1)
+     imbalances %*% factor.weights
+ }
> r.obj.3 <- new("PocockSimonRandomizer", expt, as.integer(12345),
+     d.func = sd, g.func = g.func)
> print(r.obj.3@g.func)

function (imbalances)
{
    factor.weights <- c(5, 1, 1)
    imbalances %*% factor.weights
}
```

Now the simulation.

```
> for (i in 1:1000) r.obj.3 <- randomize(r.obj.3, generateId(i),
+     generateRandomFactors(expt@factor.level.names))
```

Let's look at the distribution of treatments and the marginal distribution of factors.

```
> print(table(r.obj.3@tr.assignments[, "Treatment"]))

   Arm1    Arm2 Placebo
    333     334     333

> print(r.obj.3@stateTable)

        Sex:Female Sex:Male Race:Caucasian Race:Non-caucasian Stage:I Stage:II
Placebo        165      168            164                169     112      107
Arm1           165      168            162                171     107      108
Arm2           165      169            166                168     110      110
        Stage:III
Placebo       114
Arm1          118
Arm2          114
```

4.3 Unequal treatment assignments

Next, we try a simulation where we require 5:2:1 randomization. To really see the effect, we need to change the function that computes probabilities for picking each treatment based on the randomization. Let's be greedy and use the following:

```
> p.func.greedy <- function(overallImbalance) {
+     number.of.treatments <- length(overallImbalance)
```

```
+     k <- which(overallImbalance == min(overallImbalance))
+     p.vec <- rep(0, number.of.treatments)
+     p.vec[k] <- 1
+     p.vec/sum(p.vec)
+ }
```

Now, a new randomizer.

```
> r.obj.4 <- new("PocockSimonRandomizer", expt, as.integer(12345),
+     tr.ratios = c(5, 2, 1), p.func = p.func.greedy)
```

A simulation.

```
> for (i in 1:1000) r.obj.4 <- randomize(r.obj.4, generateId(i),
+     generateRandomFactors(expt@factor.level.names))
```

```
> print(table(r.obj.4@tr.assignments[, "Treatment"]))
```

```
   Arm1    Arm2 Placebo
    250     125     625
```

```
> print(r.obj.4@stateTable)
```

```
        Sex:Female Sex:Male Race:Caucasian Race:Non-caucasian Stage:I Stage:II
Placebo        312      313            309                316     206      202
Arm1           125      125            124                126      82       81
Arm2            62       63             62                 63      42       40
        Stage:III
Placebo       217
Arm1           87
Arm2           43
```

4.4 A different probability assignment

The drawback of using the greedy function in the previous example is that there is some predictability as to what the randomizer will assign based on the current state. To throw in a bit of uncertainty, we can define another function that favors the appropriate treatment heavily, but not deterministically.

```
> p.func.not.so.greedy <- function(overallImbalance) {
+     FAVORED.PROB <- 0.75
+     number.of.treatments <- length(overallImbalance)
+     k <- which(overallImbalance == min(overallImbalance))
+     if (length(k) > 1) {
+         k <- sample(k, 1)
+     }
+     p.vec <- rep((1 - FAVORED.PROB)/(number.of.treatments - 1),
+         number.of.treatments)
+     p.vec[k] <- FAVORED.PROB
+     p.vec
+ }
```

```
> r.obj.5 <- new("PocockSimonRandomizer", expt, as.integer(12345),
+     tr.ratios = c(5, 2, 1), p.func = p.func.not.so.greedy)
```

A simulation.

```
> for (i in 1:1000) r.obj.5 <- randomize(r.obj.5, generateId(i),
+     generateRandomFactors(expt@factor.level.names))

> print(table(r.obj.5@tr.assignments[, "Treatment"]))

   Arm1    Arm2 Placebo
    249     131     620

> print(r.obj.5@stateTable)

        Sex:Female Sex:Male Race:Caucasian Race:Non-caucasian Stage:I Stage:II
Placebo        301      319            295                325     210      192
Arm1           120      129            115                134      84       78
Arm2            64       67             71                 60      45       40
        Stage:III
Placebo       218
Arm1           87
Arm2           46
```

Another possibility for the probability function might be based on the actual imbalances.

```
> p.func.imbalance <- function(overallImbalance) {
+     p.vec <- overallImbalance/sum(overallImbalance)
+     p.vec
+ }
```

Of course, this assumes that the imbalances calculated are non-negative, which would be the case with range or standard deviation. But some care must be taken to ensure this is the case for arbitrary situations.

5 Notes

The current package can be used without recourse to a database for persistence. This would require the initial definition of the clinical experiment as in the example(s) above along with the randomizer. This is done once for a study on a designated computer running R to which the person assigned to do the randomization will have primary access.

Thereafter, every time a subject is to be randomized (after all the usual procedures for registration in the study) the randomization process will require merely an id for the subject and the levels of the prognostic factors of interest. The randomization is performed simply by running the code snippet

```
r.obj <- randomize(r.obj, id, c(fac1, fac2, fac3))
lastRandomization(r.obj)
```

where `r.obj` is a randomizer created as above, and `id`, `fac1`, `fac2`, `fac3`, are the study id and the associated factor levels of the subject to be randomized.

After each assignment, the person can save the R workspace so that the state is preserved. If R is invoked from the same directory again, the state is restored for subsequent randomizations. Of course, this means all the usual responsibilities for saving the workspace apply for this mode of operation.

References

[1] Stuart J. Pocock and Richard Simon. Sequential treatment assignment with balancing for prognostic factors in the controlled clinical trial. *Biometrics*, 31(1):103–115, 1975.