Package SBSA: Simplified Bayesian Sensitivity Analysis (Version 0.1.2)

Davor Čubranić Paul Gustafson

November 29, 2011

Abstract

SBSA is an R package that offers a simplified interface to Bayesian sensitivity analysis. This vignette contains a guided walkthrough of using the package to analyze a dataset. It covers calling into the package, and how the result can be checked, tuned, and analyzed.

1 Introduction

1.1 The Model

Consider a health outcome Y, exposure X, and confounders $\boldsymbol{Z} = (Z_1, \dots, Z_p)'$ and $\boldsymbol{U} = (U_1, \dots, U_q)'$. In the case of continous outcome Y, the outcome model can straightforwardly be expressed as:

$$(Y|\boldsymbol{U}, \boldsymbol{Z}, X) \sim N(\alpha_0 + \alpha_x X + \boldsymbol{\beta}_{\boldsymbol{u}}'\boldsymbol{U} + \boldsymbol{\beta}_{\boldsymbol{z}}'\boldsymbol{Z}, \sigma^2) \tag{1}$$

However, in practice the measurements of $\boldsymbol{U}$ are unavailable, whereas only noisy measurements $\boldsymbol{W}$ are available in place of $\boldsymbol{Z}$. Thus we refer to $\boldsymbol{U}$ and $\boldsymbol{Z}$ as unobserved and near-observed confounders, respectively.

The SBSA package provides functionality to estimate the parameters of this model from the observed data $(\boldsymbol{W}, Y, X)$. In this vignette, we will focus on practicalities of using the package. For full details of the model and the algorithm, please see Gustafson *et al.* [3].

1.2 Parameters

We already introduced in equation 1 parameters α_0, α_x, $\boldsymbol{\beta}_{\boldsymbol{u}}$, $\boldsymbol{\beta}_{\boldsymbol{z}}$, and σ^2. Of these, α_x is what an analyst will be most interested in, since it shows the relationship of the outcome to the exposure. But to understand the model and be able to tune the package, you should be aware of the remaining parameters.

Remember that W_j is a noisy surrogate for Z_j. If we assume that the measurement errors for the components of $\boldsymbol{Z}$ are uncorrelated with each other, we can model

$$(\boldsymbol{W}|Y,\boldsymbol{U},\boldsymbol{Z},X) \sim N_p(\boldsymbol{Z}, diag(\tau_1^2,\ldots,\tau_p^2)) \tag{2}$$

Next, we specify a normal model for the distribution of exposure and near-observed confounders as:

$$\begin{pmatrix} X \\ \boldsymbol{Z} \end{pmatrix} \sim N_{p+1}(0, \tilde{\Sigma}(\tau^2)) \tag{3}$$

where $\tilde{\Sigma} = \Sigma - diag(0,\tau_1^2,\ldots,\tau_p^2)$, and we pretend to know $\Sigma = Var(X, \boldsymbol{W}')$. (Note that Σ will have unit diagonal elements if X and $\boldsymbol{W}$ are standardized, simplifying the calculation.)

Finally, we need to link $\boldsymbol{U}$ to $(X,\boldsymbol{Z})$. We can simplify the model by assuming there is just a single unobserved confounder U as

$$\left\{ U \middle| \begin{pmatrix} X \\ \boldsymbol{Z} \end{pmatrix} \right\} \sim N(\gamma_x X + \boldsymbol{\gamma}_{\boldsymbol{z}}' \boldsymbol{Z}, c^2) \tag{4}$$

The choice of the single unobserved confounder corresponds to the situation where the investigator is concerned about the possible existence of one or more important confounders whose identities, and mutual relationships, are unknown.

2 Example

Let us work with a simulated dataset (Y, X, Z_1, Z_2, U), where the exposure X and the true confounders, $(\boldsymbol{Z}, U)$ are equi-correlated with corr=.6:

```
>   set.seed(42)
>   n <- 100
>   tmp <- sqrt(0.6) * matrix(rnorm(n), n, 4) +
+     sqrt(1 - 0.6) * matrix(rnorm(n * 4), n, 4)
>   x <- tmp[, 1]
>   z <- tmp[, 2:4]
```

while the observed outcome Y is generated according to Equation (1):

```
> y <- rnorm(n, x + z %*% rep(0.5, 3), 0.5)
```

The two near-observed confounders $\boldsymbol{Z}$ are mismeasured as $\boldsymbol{W}$ with ICC=0.7, while U is unobserved:

```
>   w <- z[, 1:2]
>   w[, 1] <- w[, 1] + rnorm(n, sd = sqrt(1/0.7 - 1))
>   w[, 2] <- w[, 2] + rnorm(n, sd = sqrt(1/0.7 - 1))
```

Finally, we standardize X and $\boldsymbol{W}$:

```
> standardize <- function(x) (x - mean(x))/sqrt(var(x))
> x.sdz <- standardize(x)
> w.sdz <- apply(w, 2, standardize)
```

Note: The package will check if these arguments have been standardized and will warn you if they have not.

2.1 Analysis

Let's use SBSA to estimate the parameters of the model given the observed data $(Y, X_{sdz}, \boldsymbol{W_{sdz}})$. There is a single entry point function in the SBSA package, `fitSBSA`. It runs MCMC for the specified number of steps, using the observed data and prior information. There is also a number of user-tunable parameters, which we will cover as we go along.

2.1.1 Describing the prior

Let's first express some prior information about the confounders. Recall that $1 - \tau_j^2$ is the ICC describing the reliability of W_j as a surrogate for Z_j. We can think of a prior under which each τ_j^2 is independently distributed as $\text{Beta}(a_j, b_j)$. If, in this example, we believe that ICC is very likely above 0.6 with mode at 0.8, we can express that via $\text{Beta}(6, 21)$ distribution:

```
> a <- 6
> b <- 21
```

To check, remember that the mode of $\text{Beta}(a, b) = \frac{a-1}{a+b-2}$, so the mode of ICC would be $1 - \frac{a-1}{a+b-2}$. For $\text{Beta}(6, 21)$, the ICC mode is then:

```
> 1 - (a - 1)/(a + b - 2)
```

```
[1] 0.8
```

Similarly, for ICC to likely be above 0.6, τ^2 should be likely to be *below* that value, which we can check via the value of the distribution function of $\text{Beta}(6, 21)$ at 0.6, or in R:

```
> pbeta(0.6, a, b)
```

```
[1] 0.9999737
```

2.1.2 Choosing sampler jumps

Having thus chosen our prior, we can move on. SBSA's algorithm uses MCMC with reparametrizing block-sampling, using the following six blocks: $(\boldsymbol{\alpha}^\star)$, $(\boldsymbol{\beta_z^\star})$, $(\boldsymbol{\tau^{2\star}})$, $(\sigma^{2\star})$, $(\boldsymbol{\gamma_z^\star})$, $(\gamma_x^\star, \beta_u^\star)$.[1] We will see this block structure both in inputs when specifying block sampler jumps to the algorithm, and in the output, where acceptance rates are reported for each block separately.

For each block, we need to specify the value of the sampler jump; 0.1 is a reasonable enough choice to try first:

```
> sampler.jump <- c(alpha = 0.1, beta.z = 0.1, sigma.sq = 0.1,
+     tau.sq = 0.1, beta.u.gamma.x = 0.1, gamma.z = 0.1)
```

[1]See Gustafson *et al.* [3] for details of reparametrization.

2.1.3 Running SBSA

We will leave all other parameters at their default settings, and run the MCMC for 20,000 steps:

```
> sbsa.fit <- fitSBSA(y, x.sdz, w.sdz, a, b, nrep = 20000,
+                         sampler.jump = sampler.jump)
```

As used above, we passed the following arguments to `fitSBSA`:

- the observed data, $(Y, X_{sdz}, \boldsymbol{W_{sdz}})$
- prior's hyperparameters `a` and `b`
- number of MCMC iterations, `nrep`

The result of SBSA, captured here in variable `sbsa.fit`, contains the estimated parameters $\boldsymbol{\alpha}$, $\boldsymbol{\beta_z}$, β_u, $\boldsymbol{\gamma_z}$, γ_x, $\boldsymbol{\tau^2}$, and σ^2, in respective elements of the output:

```
> names(sbsa.fit)

[1] "alpha"    "beta.z"   "gamma.z"  "tau.sq"   "gamma.x"
[6] "beta.u"   "sigma.sq" "acc"
```

An additional element of the output, `acc`, contains the acceptance rate of each block.

2.1.4 Tuning the acceptance rate

Before proceeding with analysis, we should do some high-level checks of the sampler's output. Let's begin by checking the acceptance count of the sampler:

```
> sbsa.fit$acc

         alpha         beta.z       sigma.sq         tau.sq
         10168          12006          14653           6709
beta.u.gamma.x        gamma.z
         17941          18428
```

Again, we can see the block-sampling structure in `acc`, where each MCMC sampling block gets a named element indicating the number of accepted updates. Checking the MCMC acceptance rate is an important first step before interpreting the results. The reason is that we want the algorithm to explore the state space efficiently, and acceptance rate is an indication of this. An acceptance rate that's either too high or too low means that the sampling is inefficient: high acceptance rate means that the chain is moving slowly and sampling largely around the current point; alternatively low acceptance rate means that proposed samples are often rejected and the chain is not moving much at all. Either way, the chain will explore the state space poorly, which we want to avoid.

What acceptance rate is “just right” is open to much debate (see [1], [2], [4]), but a rule of thumb given by Roberts and Rosenthal [5] recommends a rate between 0.15 and 0.5. So let us try to get the acceptance rate for each block to 30–40%, that is, `acc` in the 6000–8000 range. This is done by adjusting the size of the block’s sampling jump: when the acceptance rate is too low, we decrease the jump and, vice versa, when the acceptance rate is too high, we increase the jump. Keeping in mind is that changing one block’s jump may change the acceptance rate of other blocks, it’s still best to adjust the jump one block at a time until all are within the desired range.

```
> sbsa.fit <- fitSBSA(y, x.sdz, w.sdz, a, b, nrep = 20000,
+     sampler.jump = c(alpha = 0.2, beta.z = 0.1,
+         sigma.sq = 0.1, tau.sq = 0.1, beta.u.gamma.x = 0.1,
+         gamma.z = 0.1))
> sbsa.fit$acc
```

```
         alpha         beta.z       sigma.sq         tau.sq
          5161          12018          14634           6778
beta.u.gamma.x        gamma.z
         17746          18311
```

```
> sbsa.fit <- fitSBSA(y, x.sdz, w.sdz, a, b, nrep = 20000,
+     sampler.jump = c(alpha = 0.15, beta.z = 0.1,
+         sigma.sq = 0.1, tau.sq = 0.1, beta.u.gamma.x = 0.1,
+         gamma.z = 0.1))
> sbsa.fit$acc
```

```
         alpha         beta.z       sigma.sq         tau.sq
          7235          12112          14640           6607
beta.u.gamma.x        gamma.z
         17655          18371
```

```
> sbsa.fit <- fitSBSA(y, x.sdz, w.sdz, a, b, nrep = 20000,
+     sampler.jump = c(alpha = 0.15, beta.z = 0.2,
+         sigma.sq = 0.1, tau.sq = 0.1, beta.u.gamma.x = 0.1,
+         gamma.z = 0.1))
> sbsa.fit$acc
```

```
         alpha         beta.z       sigma.sq         tau.sq
          7096           7194          14669           6678
beta.u.gamma.x        gamma.z
         17842          18493
```

And so on until we reach a satisfactory acceptance rate:

```
> sbsa.fit <- fitSBSA(y, x.sdz, w.sdz, a, b, nrep = 20000,
+     sampler.jump = c(alpha = 0.15, beta.z = 0.2,
+         sigma.sq = 0.35, tau.sq = 0.1, beta.u.gamma.x = 0.8,
+         gamma.z = 1))
> sbsa.fit$acc
```

```
          alpha          beta.z        sigma.sq          tau.sq 
           7145            7283            7181            6623 
beta.u.gamma.x         gamma.z 
           7777            7829 
```

2.1.5 Checking parameter mixing

Once we have found good sampling jumps, we should check that the chains mix well by plotting parameter values as time series:

```
> mfrow <- par(mfrow = c(2, 2))
> plot(window(ts(sbsa.fit$alpha[, 1]), deltat = 30),
+     ylab = expression(alpha[0]))
> plot(window(ts(sbsa.fit$alpha[, 2]), deltat = 30),
+     ylab = expression(alpha[x]))
> plot(window(ts(sbsa.fit$beta.u), deltat = 30),
+     ylab = expression(beta[u]))
> plot(window(ts(sbsa.fit$gamma.x), deltat = 30),
+     ylab = expression(gamma[x]))
> par(mfrow = mfrow)
```

As you can see in Figure 1, the mixing appears to be fine, so we can proceed with the analysis.

2.1.6 Parameter inference

Finally, we can have a look at the estimated value of each parameter. Here, we look at α_x, but throw away the first 10,000 iterations as the burn-in:

```
> mean(sbsa.fit$alpha[10001:20000, 2])
```

```
[1] 1.141829
```

```
> sqrt(var(sbsa.fit$alpha[10001:20000, 2]))
```

```
[1] 0.7652895
```

Keep in mind that these parameters are estimated using X_{sdz} and $\boldsymbol{W_{sdz}}$, the standardized X and $\boldsymbol{W}$. In order to get them back we need to reverse the standardizing transformation:

```
> trgt <- sbsa.fit$alpha[10001:20000, 2]/sqrt(var(x))
> c(mean(trgt), sqrt(var(trgt)))
```

```
[1] 1.1381229 0.7628055
```

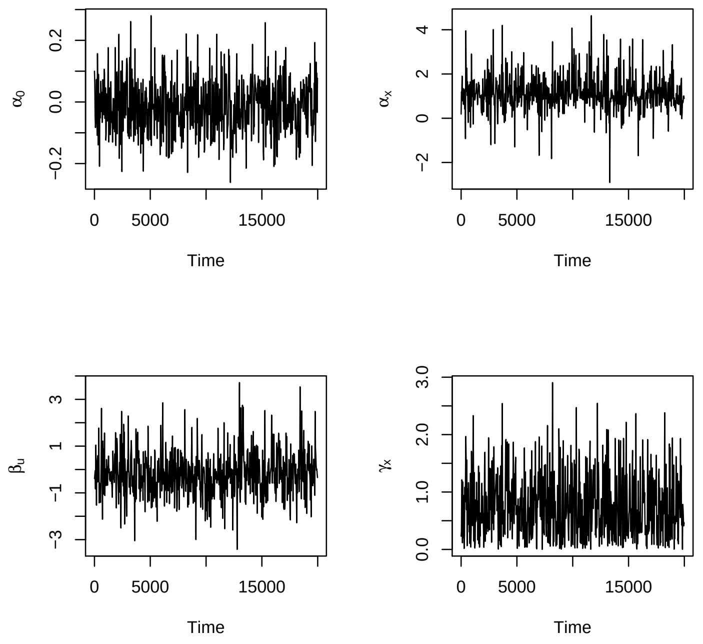

Figure 1: Parameter traces, thinned to every thirtieth sample

2.2 Handling different levels of mis-measurement

What about the case where we believe that different confounders are measured with different accuracy? In this case, $\tau_i \neq \tau_j$, and we would like the model to reflect our assumption.

In the following examples, we'll work with a modified $\boldsymbol{W}$, in which one component is mismeasured with ICC=0.7, while the other is measured more accurately, with ICC=0.95:

```
>   w <- z[, 1:2]
>   w[, 1] <- w[, 1] + rnorm(n, sd = sqrt(1/0.7 - 1))
>   w[, 2] <- w[, 2] + rnorm(n, sd = sqrt(1/0.95 - 1))
>   w.sdz <- apply(w, 2, standardize)
```

The prior can reflect our new belief about $\boldsymbol{W}$ by expressing the ICC (or rather, τ_j) of each component separately. As before, we believe that the ICC of W_1 is very likely above 0.6 with mode at 0.8, modelled via Beta(6, 21). But we now also believe that the ICC of W_2 is very likely above 0.8 with mode at 0.95, which we can model via $\tau_2^2 \sim \text{Beta}(3, 39)$:

```
> a <- c(6, 3)
> b <- c(21, 39)
```

We check our choices of a and b as before. First, the ICC mode:

```
> 1 - (a - 1)/(a + b - 2)

[1] 0.80 0.95
```

and the likelihood of ICC being above the desired value for each component:

```
> pbeta(c(0.6, 0.8), a, b)

[1] 0.9999737 1.0000000
```

Generally with the random walk Metropolis-Hastings algorithm, the best mixing is obtained if the component jump sizes scale according to the corresponding posterior standard deviations. This means that the magnitude of jump of the $\boldsymbol{\tau}^{\mathbf{2}\star}$ block sampler might also reasonably be different for each component of $\boldsymbol{\tau}^{\mathbf{2}}$. We specify per-component jump by giving a numeric vector as the `tau.sq` element of the `sampler.jump` argument, in this case using the previous jump value for the first component (0.1) and trying half the size for the second (0.05):[2]

```
> sbsa.fit <- fitSBSA(y, x.sdz, w.sdz, a, b, nrep = 20000,
+     sampler.jump = list(alpha = 0.15, beta.z = 0.2,
+         sigma.sq = 0.35, tau.sq = c(0.1, 0.05),
+         beta.u.gamma.x = 0.8, gamma.z = 1))
> sbsa.fit$acc

         alpha          beta.z        sigma.sq          tau.sq 
          6484            6043            6319            7025 
beta.u.gamma.x         gamma.z 
          8133            7976 
```

We still need to increase slightly the magnitude of jump for the $(\gamma_x^\star \beta_u^\star)$ block:

```
> sbsa.fit <- fitSBSA(y, x.sdz, w.sdz, a, b, nrep = 20000,
+     sampler.jump = list(alpha = 0.15, beta.z = 0.2,
+         sigma.sq = 0.35, tau.sq = c(0.1, 0.05),
+         beta.u.gamma.x = 0.9, gamma.z = 1))
> sbsa.fit$acc

         alpha          beta.z        sigma.sq          tau.sq 
          6550            6056            6390            6961 
beta.u.gamma.x         gamma.z 
          7033            7732 
```

[2]Note that this means `sampler.jump` now has to be a list, because its elements have differing lengths.

Finally, with all acceptance counts are still within the desired range, we move on to checking the mixing of the chains (Fig. 2):

```
> mfrow <- par(mfrow = c(2, 2))
> plot(window(ts(sbsa.fit$alpha[, 1]), deltat = 30),
+     ylab = expression(alpha[0]))
> plot(window(ts(sbsa.fit$alpha[, 2]), deltat = 30),
+     ylab = expression(alpha[x]))
> plot(window(ts(sbsa.fit$beta.u), deltat = 30),
+     ylab = expression(beta[u]))
> plot(window(ts(sbsa.fit$gamma.x), deltat = 30),
+     ylab = expression(gamma[x]))
> par(mfrow = mfrow)
```

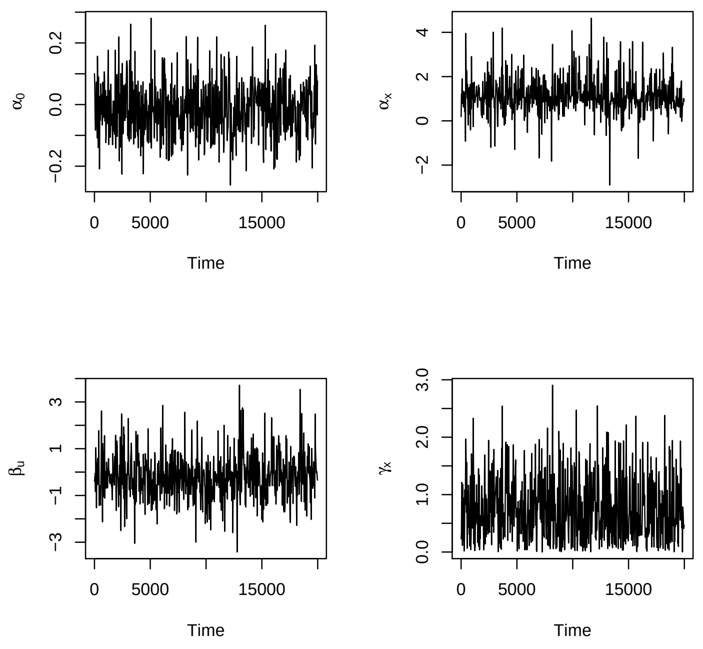

Figure 2: Parameter traces, thinned to every thirtieth sample

We can see the difference in the magnitude of error in the two components of $\boldsymbol{W}$ in the posterior density of τ^2 (Figure 3):

```
> tau.density <- kde2d(sbsa.fit$tau.sq[, 1],
+                      sbsa.fit$tau.sq[, 2],
+                      lims = c(0, max(sbsa.fit$tau),
+                                 0, max(sbsa.fit$tau)))
> filled.contour(tau.density,
+                 color.palette = function(n) grey(n:0 / n),
+                 xlab = expression({tau[1]}^2),
+                 ylab = expression({tau[2]}^2))
```

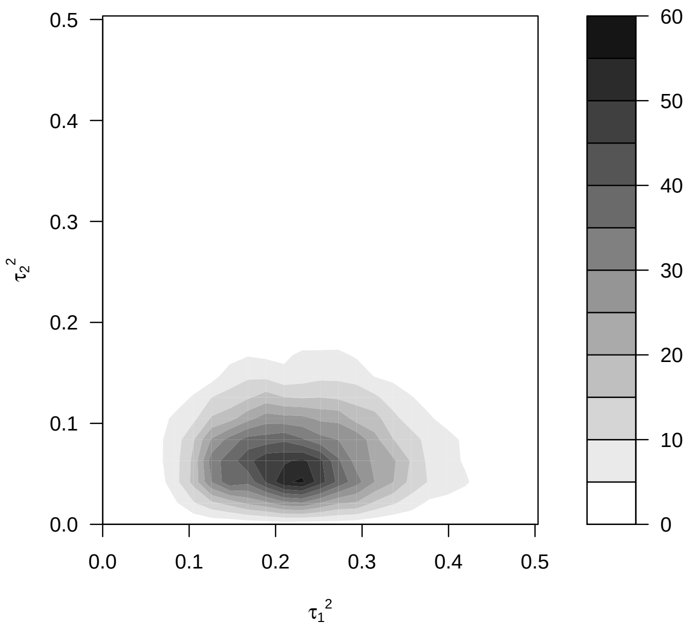

Figure 3: Contours of posterior density for τ^2

Finally, let's have a look at α_x, again throwing away the first 10,000 iterations as the burn-in:

```
> mean(sbsa.fit$alpha[10001:20000, 2])
```

```
[1] 1.097911
```

```
> sqrt(var(sbsa.fit$alpha[10001:20000, 2]))
```

```
[1] 0.7771366
```

And reversing the standardizing transformation:

```
> trgt <- sbsa.fit$alpha[10001:20000, 2]/sqrt(var(x))
> c(mean(trgt), sqrt(var(trgt)))
```

```
[1] 1.0943478 0.7746141
```

3 Conclusion

In this vignette, we guided you through a session using SBSA. We covered the basic arguments of the `fitSBSA` function, and how the result can be checked, tuned, and analyzed. There are other parameters of the algorithm that can be changed, but we refer you to the manual for full details.

In addition, the data we used in our example was continuous in Y. If your outcome variable is binary, you should pass argument `family='binary'` to `fitSBSA` function, which will switch to use a variant of the SBSA algorithm designed for binary Y. This variant does not use some of the parameters used by the algorithm for the continuous case (notably, jump for σ^2), and introduces additional ones. Once again, we refer you to the package manual for information on using the function, and to Gustafson *et al.* [3] for details of the statistical model used.

References

[1] A. Gelman, G. O. Roberts, and W. R. Gilks. (1996) Efficient Metropolis jumping rules. *Bayesian Statistics*, 5, 599–607.

[2] C. J. Geyer and E. A. Thompson. (1995) Annealing Markov chain Monte Carlo with applications to ancestral inference *Journal of the American Statistical Association*, 90, 909–920.

[3] P. Gustafson, L. C. McCandless, A. R. Levy, and S. Richardson. (2010) Simplified Bayesian Sensitivity analysis for mismeasured and unobserved confounders. *Biometrics*, 66(4):1129–1137. DOI: 10.1111/j.1541-0420.2009.01377.x

[4] G. O. Roberts, A. Gelman, and W. R. Gilks. (1997) Weak convergence and optimal scaling of random walk Metropolis algorithms. *Annual of Applied Probability*, 7, 110–120.

[5] G. O. Roberts, and J. S. Rosenthal. (2001) Optimal scaling for various Metropolis-Hastings algorithms. *Statistical Science*, 16, 351–367.