CMBBHT: Cell Means Based Bayesian Hypothesis Tests

Kyle Hardman

Package version 0.1.0, February 2017

Contents

Disclaimer

There are requirements for using this package that you as a user of this package need to be aware of. You are responsible for reading the documentation and the referenced articles. A failure to do so means that your analyses may be very wrong.

That said, the approaches used by this package are fairly general, so you are probably ok at the $p < 0.05$ threshold.

Introduction

This package performs ANOVA-like Bayesian hypothesis tests for factorial designs for which cell means or differences between cells are estimated in a Bayesian model.

You must have a factorial design in which some dependent variable is allowed to vary as a function of factor levels. Continuous independent variables are not supported.

You must have samples from the prior and posterior distributions of the parameters of interest.

This manual will first go through examples of some of the kinds of models that this package is designed to analyze. Then, basic usage of the package will be demonstrated. The steps of the procedure used to estimate Bayes factors are then shown. Finally, there are additional notes on how to use the package.

Terminology

There are two important terms that are used many times here that can be demonstrated with an example. A two-factor ANOVA model might be written (in part) as follows:

$$\mu_{ij} = \mu + \alpha_i + \beta_j + \pi_{ij}$$

Where the αs and βs are main effect parameters and the πs are interaction parameters.

Cell mean: Typically used to mean a parameter that estimates a cell mean or some measure of differences between cell means. In the ANOVA model, μ_{ij} estimates a cell mean.

Effect parameter: In the ANOVA model, the αs, βs, and πs are *effect parameters* as they are parameters related to a certain effect (a main effect or an interaction).

Example Models

What follow are some examples of the kinds of models that you might have that you can use with this package. These examples are not meant to exclude other models.

Example 1: Basic cell means model

You have a design with some number of factors that are crossed in some way to get cells of the design. In each cell of the design, you have some number of replicates/measurements of some dependent variable. Your model estimates the mean of each cell of the design. What follows is a concrete example of such a model, not a specification of what your model must be.

In each cell, i, there are K replicates of observed data, y_{ik}. They follow a normal distribution with a mean that is estimated in each cell, μ_i, and a data variance, σ_y^2.

$$y_{ik} \sim \text{Normal}(\mu_i, \sigma_y^2)$$

The cell means, μ_i, and the data variance, σ_y^2, each have some fixed prior.

$$\mu_i \sim \text{Normal}(0, 10^4)$$
$$\sigma_y^2 \sim \text{Inverse Gamma}(0.1, 0.1)$$

This design is possibly workable, but it has the issu that the prior on the cell means is uninformative. The problem with uninformative priors is discussed further here. I prefer the design in the next example, which is able to place informative priors on the quantities of interest.

Example 2: Grand mean plus Cell Offsets/Differences/Effects

In this example, the data are the same as in Example 1. The model estimates a grand mean/intercept, μ, (the interpretation of μ depends on other choices) and a cell effect, α_i, for each cell. To get the mean for each cell, you add the cell effect to the grand mean/intercept.

At the data level, this model is the same as in Example 1.

$$y_i \sim \text{Normal}(\mu_i, \sigma_y^2)$$

Now each cell mean is the sum of the intercept, μ, and the cell effect, α_i.

$$\begin{aligned}
\mu_i &= \mu + \alpha_i \\
\mu &\sim \text{Normal}(0, 10^4) \\
\alpha_i &\sim \text{Normal}(0, 2) \\
\sigma_y^2 &\sim \text{Inverse Gamma}(0.1, 0.1)
\end{aligned}$$

In this case, you can use the cell effects (αs) for analysis, ignoring the intercept. This model has an advantage over the basic cell means model in that you can place an uninformative prior on the grand mean, which means that you don't need to guess the approximate magnitude of the mean of the data. You can, however, place reasonably informative priors on the αs. The informativeness of the priors on the αs are effectively your prior beliefs about how large the differences between the cells are likely to be.

Note that some constraint on the αs is required for this model to be identifiable, otherwise the mean of the αs and the value of μ will trade off. I like to use a cornerstone parameterization in which one of the αs is set to 0. In that case, μ is an intercept. You could also use a sums-to-zero constraint, in which case μ is a grand mean, or some other constraint.

With a cornerstone parameterization, you are essentially changing the model as follows:

$$\begin{aligned}
\alpha_1 &\sim \text{Normal}(0, 0) \text{ (i.e. a point mass at 0.)} \\
\alpha_i &\sim \text{Normal}(0, 2), i \neq 1
\end{aligned}$$

Thus, different αs have different priors. I see no problem with this, you just have to be careful that when you sample from the priors on the αs that you account for the different priors. Note that you do not need to use 1 as the cornerstone condition. I have found that using the cell with the most data, if there is one, works the best.

I typically use a model like this with a cornerstone parameterization and it usually works well. Part of the value of this package is that it allows you to use a simple model like this one, that is not parameterized in terms of main effects and interactions, but after estimating the cell effects, α_i, you can test whether main effects or interactions are present.

Example 3: Full ANOVA Effects Model

In this example, you have the same data as Example 1. let's say that you have two factors called A and B. Your model estimates a grand mean (μ), a main effect of A (α_i), a main effect of B (β_j), and the interaction between A and B (π).

The data level is the same: $y_i \sim \text{Normal}(\mu_i, \sigma_y^2)$

The cell mean is calculated with: $\mu_i = \mu + \alpha_i + \beta_j + \pi_{ij}$

The components of the cell mean have the following priors.

$$\begin{aligned}
\mu &\sim \text{Normal}(0, 10^4) \\
\alpha_i &\sim \text{Normal}(0, 2) \\
\beta_j &\sim \text{Normal}(0, 2) \\
\pi_{ij} &\sim \text{Normal}(0, 2) \\
\sigma_y^2 &\sim \text{Inverse Gamma}(0.1, 0.1)
\end{aligned}$$

Notice how this model specification allows for informative priors on the effect parameters (α, β, and π) while allowing for an uninformative prior on the grand mean, μ.

If you have a model like this, you should be able to use this package fairly easily. Rather than using cell means for your analysis, you can directly use your estimates of the effect parameters, α, β, and π. To test hypotheses about the existence of effects, you can pass the prior and posterior effect parameters to `testFunction_SDDR` or `testFunction_encompassingPriors`. This will make more sense once you read farther.

Note that like Example 2, you must provide some constraints on the effect parameters in order for this model to be identifiable. This gets complicated when you have multiple factors so I won't go into it. I don't typically parameterize my models like this because of the complexity of constraining the effect parameters.

Example 4: A Complex Model

For all of the examples so far, you might be wondering why I didn't just use the `BayesFactor` package. For all of those examples, you should just use the `BayesFactor` package. Where this package becomes useful is when your model is complex and there is a fair degree of separation between the parameters and the data. An example of such a model was used by Hardman, Vergauwe, & Ricker (2017). In that model, the data are distributed as a mixture of various distributions. If we are interested in a parameter ϕ, version of the model that leaves a lot out is something like

$$\begin{aligned}
p(y_{ij}|\phi_{ij}, \theta) &= \sum_k p_k f_k(\phi_{ij}, \theta) \\
\phi_{ij} &= T(\phi_i + \phi_j) \\
\phi_i &\sim \text{Normal}(\mu_\phi, \sigma_\phi^2) \\
\phi_j &\sim \text{Cauchy}(0, 3)
\end{aligned}$$

Where y_{ij} are the data for the ith participant in the jth task condition, ϕ_{ij} is a parameter of interest, θ are a bunch of other parameters, and p_k and f_k are mixture weights and density functions for the kth component of the model. ϕ_{ij}, unfortunately, is some nonlinear transformation, $T()$, of the sum of a participant effect, ϕ_i, and an effect of task condition, ϕ_j.

Inference is focused on the effect of task condition, ϕ_j. That is where this package shines: It can test whether ϕ_j varies with task condition. All it needs are the prior and posterior samples of ϕ_j.

Basic Usage

I will work with a model like in Example 2. In that model, differences from a cornerstone condition are estimated with α_i parameters. I will use a design with two factors, called `let` and `num`. The `let` factor has three levels, `a`, `b`, and `c`. The `num` factor has two levels, `1` and `2`. The design is fully crossed.

One of the things that you will need in order to use this package is a `data.frame` that contains the factor levels. Each column is a factor and each row contains a combination of factor levels that define a cell of the design. Note that factor names and factor levels may not contain period (".") or colon (":"). The following code creates the `factors data.frame` for future use.

```
factors = data.frame(
    let = c('a', 'a', 'b', 'b', 'c', 'c'),
    num = c('1', '2', '1', '2', '1', '2')
)
```

Imagine that the cell means are 2 through 7 and that `ex2_getAlphas` samples from the prior and posterior distributions of the α parameters.

```
# Corresponds to the rows of factors
mus = c( 2,3,   4,5,   6,7 )

set.seed(143)
cms = ex2_getAlphas(mus)
```

`cms` contains two matrices, `prior` and `post`, each of which has 6 columns, each corresponding to an α parameter and 10000 rows, each being a sample from the prior or posterior. Note that the order of the columns of cell means/effects must correspond to the rows of `factors`.

```
names(cms)
```

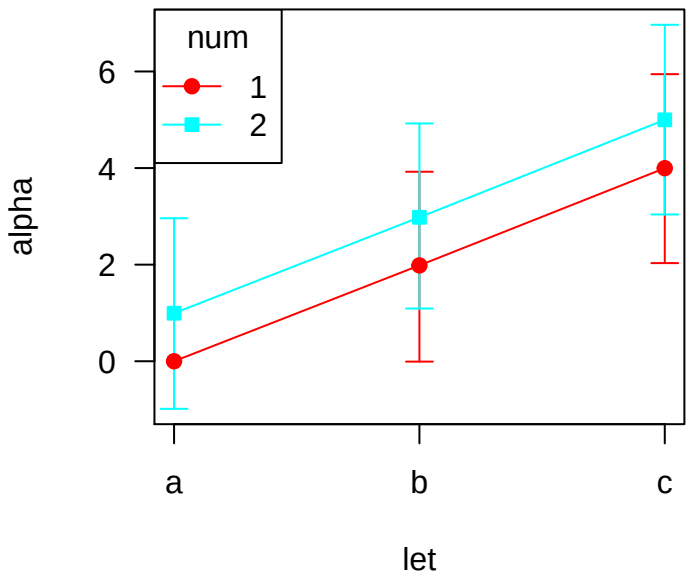
```
## [1] "prior" "post"
```

```
dim(cms$post)
```

```
## [1] 10000     6
```

A plot the effect parameters with 95% credible intervals will help with understanding what effects are present in the data.

```
library(LineChart)

#Put the data into a data.frame for plotting
cmdf = NULL
for (i in 1:ncol(cms$post)) {
    temp = data.frame(let = factors$let[i], num = factors$num[i], alpha = cms$post[,i])
    cmdf = rbind(cmdf, temp)
}

errBarFun = function(x) {
    list(eb = quantile(x, c(0.025, 0.975)),
             includesCenter = TRUE)
}

lineChart(alpha ~ let * num, cmdf, errBarType = errBarFun)
```



From looking at this plot, it seems like:

1. There is a main effect of letters.
2. There may or may not be a main effect of numbers.
3. The lines are really parallel, so there is no interaction.

Let's test these hypotheses. First, the main effect of letters. To test this hypothesis, pass the prior and posterior cell means, the `factors` `data.frame`, and the effect to test, `"let"`, to `testHypothesis`.

```
testHypothesis(cms$prior, cms$post, factors, "let")
```

```
## $bf01
## [1] 0.001394559
##
## $bf10
## [1] 717.0726
##
## $pKept
## [1] 1
##
## $success
## [1] TRUE
```

With default settings, this returns a list with four elements. As long as `success` is `TRUE`, we can examine `bf10`, which is the Bayes factor in favor of the hypothesis that there is a main effect of the `let` factor. This Bayes factor is substantial, so there is a main effect of `let`.

The test of the main effect of `num` procedes in the same way.

```
ht = testHypothesis(cms$prior, cms$post, factors, "num")
ht$bf10
```

```
## [1] 0.8127249
```

This Bayes factor is ambiguous as it is near 1. This fits with how the figure looks.

For the interaction, you provide a vector of the names of the factors in the interaction.

```
ht = testHypothesis(cms$prior, cms$post, factors, c("let", "num"))
ht$bf10
```

```
## [1] 0.1120962
```

This Bayes factor favors the null as it is well below 1, so there is no interaction. Again, this fits with the figure.

This concludes basic use of the package. With just a little setup and one function call, you are able to test hypotheses about main effects or interactions. This package, however, has more to offer, as I describe in the following sections.

Steps of the procedure

This section goes step-by-step through what needs to happen to perform a hypothesis test. The point of this section is to help you understand how this package works.

I am going to perform a test of the hypothesis that there is a main effect of the `let` factor for the same data that was just being used. The null hypothesis is that there is no main effect and the alternative hypothesis is that there is a main effect. If there is no main effect, there should be no difference between the levels of the `let` factor.

Effect Parameters

You can calculate matrices of effect parameters given cell means/differences and the `factors data.frame` with the function `getEffectParameters`. Here I calculate the posterior effects.

```
# Get effect parameters for the "let" main effect.
post_eff = getEffectParameters(cms$post, factors, "let")
post_eff[1:5,]
```

```
##          let.a      let.b     let.c
## [1,] -2.148408 -0.2254235 2.3738317
## [2,] -1.761095  0.1415843 1.6195109
## [3,] -1.998704  1.5738687 0.4248356
## [4,] -2.079807 -0.3257659 2.4055726
## [5,] -2.658869  0.2015305 2.4573386
```

Again, if there is no main effect, there should be no difference between the levels of the `let` factor and the effect parameters should be near 0.

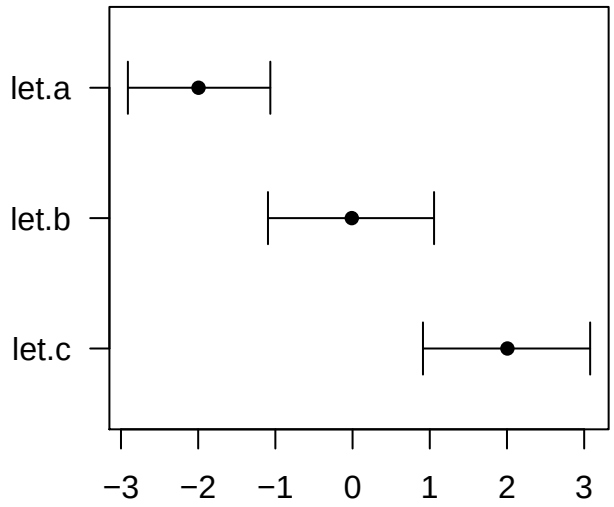
You can summarize the effect parameters with the `summarizeEffectParameters` function.

```
summary = summarizeEffectParameters(post_eff)
summary
```

```
##   effect        mean       2.5%          50%     97.5%
## 1  let.a -1.99546840 -2.9103356 -1.998292045 -1.065362
## 2  let.b -0.01007051 -1.0956426 -0.008953975  1.055700
## 3  let.c  2.00553891  0.9117691  2.009282383  3.077438
```

You can also plot the summarized effects.

```
plotEffectParameterSummary(summary)
```



As you can see, there appears to be a main effect because the effect parameters are reasonably far from one another, relative to their credible intervals.

If you don't like Bayes factors as a method of inference, you can still get something out of this package by using summaries of the posterior effect parameters. Alternately, you can group posterior effects with pairwise comparisons with `groupEffectParameters`

```
groupEffectParameters(post_eff)
```

```
##
## let.a A
## let.b   B
## let.c     C
```

By default, `groupEffectParameters` uses credible intervals of the difference between effect parameters to group the effects. You can alternately set it to use Bayes factors.

Deviance Measure

We now that have explored the effect parameters, we can move on to the next step. Again, under the null hypothesis, the effect parameters should be near to one another. Under the alternative hypothesis, the effect parameters may be more spread out. What we want is a measure of the deviance/spread of the effect parameters. The default deviance measure used by this package is the sample variance, which I have found to work well (there is no difference between sample variance and population variance in this case; it's just a scale factor).

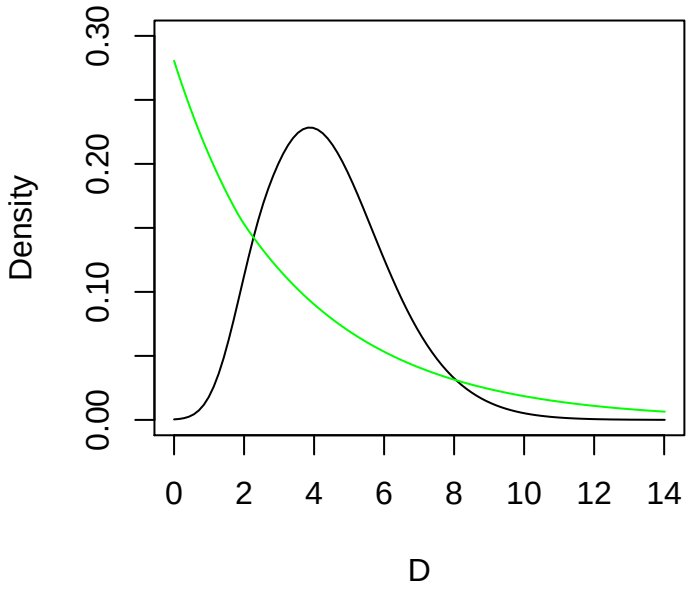
If you have a matrix of effect parameters, calculating the deviance, D, is very easy. Just `apply` the deviance function, in this case `var`, to each sample (row) from the effect parameters in `post_eff`.

```
# Calculate vector of deviances
post_D = apply(post_eff, 1, var)
```

We can do the same for the prior effects and plot the prior and posterior Ds.

```
prior_eff = getEffectParameters(cms$prior, factors, "let")
```

```
prior_D = apply(prior_eff, 1, var)

#Little helper function
estimateDDens = function(d, xs) {
    ls = polspline::logspline(d, lbound=0)
    polspline::dlogspline(xs, ls)
}

xs = seq(0, max(post_D), length.out=100)

plot(xs, estimateDDens(post_D, xs), type='l', ylim=c(0, 0.3),
         xlab="D", ylab="Density")
lines(xs, estimateDDens(prior_D, xs), col="green")
```



The posterior is black and the prior is green. As you can see from the plot, there is much more prior density at 0 than posterior density at 0.

Under the null hypothesis, the effect parameters are close to one another and D is consequently near 0. If there is more posterior density at 0 than prior density, it suggests that in the posterior the effect parameters are more clustered than the prior effect parameters, which is evidence in favor of the null hypothesis. If there is less posterior density at 0 than prior density, it suggests that the posterior effect parameters are more spread out than in the prior, which is evidence in favor of the alternative hypothesis. This is the basic logic of the Savage-Dickey density ratio.

Bayes Factor Estimation with Savage-Dickey

By default, the Bayes factor is estimated using the Savage-Dickey Density Ratio (SDDR) as suggested by Wagenmakers, Lodewyckx, Kuriyal, & Grasman (2010). The SDDR is used to test point hypotheses about

parameter values. The Bayes factor in favor of the hypothesis that some parameter $\phi = 0$ is equal, under certain conditions, to the ratio of the posterior density of ϕ at 0 to the prior density of ϕ at 0.

There are some limitations for the use of the SDDR which are discussed by Wagenmakers et al. (2010) so I will not reiterate them here. The SDDR is fairly general, so it is likely fine to use.

The following code snippet calculates the SDDR for the deviance parameter, D.

```
library(polspline)

# Estimating posterior density at 0
post_ls = logspline(post_D, lbound=0)
post_dens = dlogspline(0, post_ls)

prior_ls = logspline(prior_D, lbound=0)
prior_dens = dlogspline(0, prior_ls)

# Calculate Bayes factors with the SDDR
bf_01 = post_dens / prior_dens
bf_10 = 1 / bf_01

bf_01 # BF that D = 0 and there is no effect
```

```
## [1] 0.001394559
```

```
bf_10 # BF that D =/= 0 and there is an effect
```

```
## [1] 717.0726
```

With some extra details to make it more robust to priors with thick tails (like the Cauchy), the last few steps are done by the following function call:

```
testFunction_SDDR(prior_eff, post_eff)
```

```
## $bf01
## [1] 0.001394559
##
## $bf10
## [1] 717.0726
##
## $pKept
## [1] 1
##
## $success
## [1] TRUE
```

See this section for issues that can arise with the default density estimation approach and some possible solutions.

Conclusion

If you read this section, you now have an idea about the basic logic of how this package works. The steps shown here are all performed by the following function call.

```
ht = testHypothesis(cms$prior, cms$post, factors, "let")
ht$bf10
```

```
## [1] 717.0726
```

Notes

The Effect of Uninformative Priors

This section uses the Example 1 model, the basic cell means model. You can use the cell means for some analyses, but the priors on the cell means as in Example 1 are problematic in that kind of design. The issue is that when you directly estimate cell means, a general model must have fairly uninformative priors to the cell means in order to deal with the fact that the model does not know the approximate magnitude of the cell means. In the example, the prior variance on μ_i is 10,000 to account for the fact that μ can have very different magnitudes. This is a very uninformative prior.

To get useful Bayes factors, you should not have uninformative priors on the quantities of interest (cell means or cell effects). The logic goes as follows: An uninformative prior on cell-mean-like parameters is typically some kind of very wide almost uniform distribution. This puts excessive mass on values that are profoundly unlikely. The posterior is very unlikely to be as extreme (in aggregate) as the prior. An important step in this method is to measure the dispersion of the prior and posterior effect parameters. If the prior is extremely dispersed, the posterior is likely to be less dispersed than the prior. A hypothesis test might conclude that the posterior is less dispersed than the prior, which is evidence that there is no effect. Thus, uninformative priors result in a situation in which it is relatively easy to find evidence in favor of the null hypothesis.

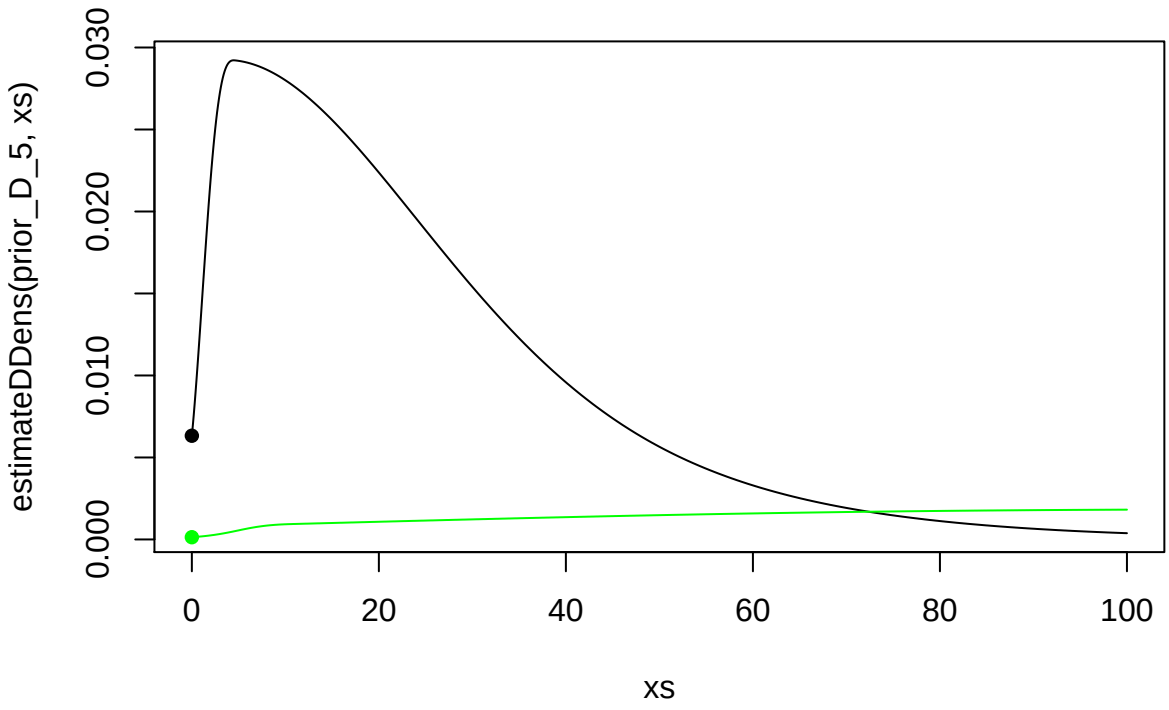
If you use prior knowledge about the approximate magnitude of the cell means, you do not need priors that are particularly uninformative, but it requires that you know the approximate magnitude of the means (if you do know, this is a legitimate use of prior knowledge). Note that if you do set the priors on cell means to be informative, you should use the same location/mean for the prior on each cell mean. The reason is that if you "guess" that there is a main effect or interaction by using different locations for different cells, then you are effectively building that effect into the prior. Thus, the null hypothesis would be not that there is no effect, but that there is a specific effect.

The code below demonstrates what happens with more and less informative priors on cell means. It uses a case where the means of the 4 cells of the design are 0, 1, 2, and 3.

```
ex1_getCMs = function(mus, posteriorSD = 1, priorMu = 0, priorSD = 3, samples = 10000) {

    if (length(priorMu) == 1) {
        priorMu = rep(priorMu, length(mus))
    }

    priorCMs = postCMs = matrix(nrow = samples, ncol=length(mus))

    for (i in 1:ncol(postCMs)) {
        priorCMs[,i] = rnorm(nrow(priorCMs), priorMu[i], priorSD)
        postCMs[,i] = rnorm(nrow(postCMs), mus[i], posteriorSD)
    }

    list(prior = priorCMs, post = postCMs)
}


fact = data.frame(f = 1:4)
mus = c( 0, 1, 2, 3 )

cm1 = ex1_getCMs(mus, priorSD = 5)

# Get effect parameters and calculate D
pf_5 = getEffectParameters(cm1$prior, fact, "f")
```

```
prior_D_5 = apply(pf_5, 1, var)


cm1 = ex1_getCMs(mus, priorSD = 20)

pf_20 = getEffectParameters(cm1$prior, fact, "f")
prior_D_20 = apply(pf_20, 1, var)
```

```
xs = seq(0, 100, 0.1)

plot(xs, estimateDDens(prior_D_5, xs), type='l')
lines(xs, estimateDDens(prior_D_20, xs), col="green")

dens5 = estimateDDens(prior_D_5, 0)
dens20 = estimateDDens(prior_D_20, 0)

points(0, dens5, pch=16)
points(0, dens20, pch=16, col="green")
```



A prior SD of 5 is black and a prior SD of 20 is green. As you can see, changing the informativeness of the prior has substantially changed the distribution of D. It has especially affected the density at 0, with the ratio `dens5 / dens20` = 46.3546209.

The following code shows what happens when you build an effect into the prior. There is a main effect, but it is the same main effect as in the prior, so the evidence is in favor of the null.

```
cm1_h1_builtIn = ex1_getCMs(mus, priorMu = mus)
testHypothesis(cm1_h1_builtIn$prior, cm1_h1_builtIn$post, fact, "f")
```

```
## $bf01
## [1] 0.3310587
##
## $bf10
## [1] 3.020613
##
## $pKept
## [1] 1
##
## $success
## [1] TRUE
```

Due to the issues with the basic cell means model, I prefer the Example 2 model or one like it. Even then, you still need to use reasonably informative priors on the cell differences, α_i.

Hierarchical Models

One example of a case that is a little complicated is when a parameter has an estimated prior as in a hierarchical model. For a model like in Example 2, it seems reasonable to place a hierarchical prior on the cell effects (αs in that model) to constrain them. For example, imagine that the prior on α_i is changed as follows:

$$\alpha_i \sim \text{normal}(0, \sigma_\alpha^2)$$
$$\sigma_\alpha^2 \sim \text{Inverse Gamma}(6, 10)$$

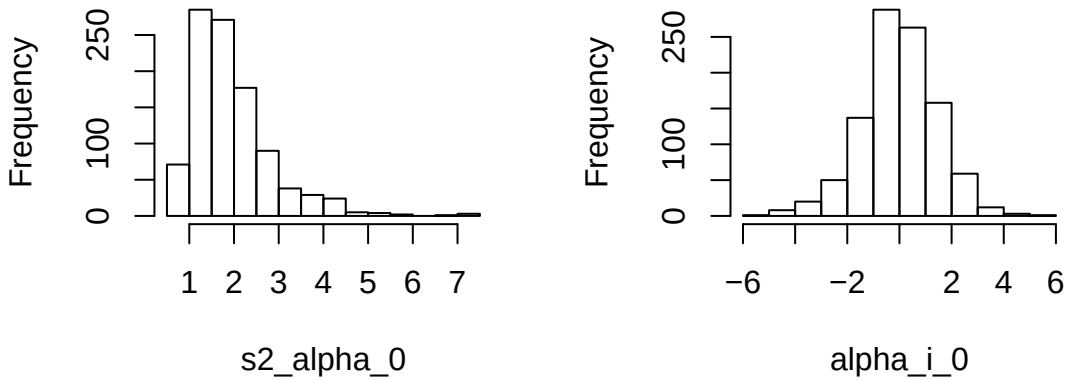
There are two things to be aware of with this setup:

First, the prior on α_i is based on prior values of σ_α^2, not posterior values. To sample from the prior on α_i, you first sample from the prior on σ_α^2 and then sample from the normal prior on α_i given the prior samples of σ_α^2. This is illustrated in the following code snippet.

```
# Sample from the prior on s2_alpha
s2_alpha_0 = MCMCpack::rinvgamma(1000, 6, 10)

# Sample form the prior on alpha_i
alpha_i_0 = rnorm(1000, 0, sqrt(s2_alpha_0))
```

Secondly, the prior on σ_α^2 should be such that the normal prior on α_i is reasonably informative. The choices of 6 and 10 for the parameters of the inverse gamma result in a mean σ_α^2 of 10 / (6 - 1) = 2 and fairly low variance, as can be seen in the left histogram below.

```
par(mfrow=c(1,2))
hist(s2_alpha_0, main="")
hist(alpha_i_0, main="")
```



As a result, the prior on α_i, on the right, is fairly informative, which is typically good. As was discussed, uninformative priors are problematic for obtaining useful Bayes factors. You should try to choose reasonably informative priors even in the more complex hierarchical settings.

Encompassing Priors Approach

Wetzels, Grasman, & Wagenmakers (2010) suggest a different method of estimating the Bayes factor than the Savage-Dickey Density Ratio. Essentially, it works nicely for interval tests and other non-point hypotheses. In addition, it does not require point density estimation, which is advantageous.

Imagine that you have two hypotheses about effect parameters, αs, that you want to compare. One model, the *encompassing* model, states that the αs are free to vary. The other model places a constraint on the αs, that they are near 0.

$$M_1 : \alpha_i$$
$$M_0 : |\alpha_i| < \epsilon$$

I will use different notation than Wetzels et al. (2010) and notate the encompassing/alternative model as M_1 and the constrained/null model as M_0.

The gist of the approach is that, under certain circumstances, the Bayes factor in favor of the null model (M_0) over the alternative model (M_1) can be expressed as the proportion of the posterior of M_1 that satisfies the constraint of M_0 divided by the proportion of the prior of M_1 that satisfies the constraint of M_0. In other words, you only need to fit M_1 and obtain its posterior. Then you retroactively apply the M_0 constraint to the prior and posterior of M_1 and see what proportion of each satisfies that constraint.

This idea is expressed in Equation 2 of Wetzels et al. (2010), which is reproduced below with some adjustments and simplifications.

$$BF_{01} = \frac{\frac{1}{m}\sum_{j=1}^{m} I_{M0}(\vec{\alpha}^{(j)}|D, M_1)}{\frac{1}{n}\sum_{k=1}^{n} I_{M0}(\vec{\alpha}^{(k)}|M_1)}$$

In this equation, the numerator is the posterior (indicated with $|D$ meaning "given data") and the denominator is the prior. $\vec{\alpha}^{(j)}$ is the jth vector of effect parameters (all of the αs) and the same for k. I_{M0} is an indicator function for the null model, M_0. It returns 1 (true) if the constraints of M_0 hold or 0 (false) if the constraints of M_0 do not hold.

To make this concrete, let's set ϵ to 1, which makes M_0 be that all of the αs must be within 1 unit of 0.

```
# The indicator function takes a vector of effect parameters and returns
# TRUE (numeric 1) if all absolute values are less than 1 and FALSE (0) otherwise
I_M0 = function(alpha_vect) {
    all(abs(alpha_vect) < 1)
}

# Calculate the numerator: The average I_M0 for the posterior
post_I = apply(post_eff, 1, I_M0)
numerator = mean(post_I)

# Calculate the denominator: The average I_M0 for the prior
prior_I = apply(prior_eff, 1, I_M0)
denominator = mean(prior_I)

bf_01 = numerator / denominator #In favor of the constrained hypothesis/model
bf_10 = 1 / bf_01 # In favor of the encompassing hypothesis/model
```

The logic of the encompassing priors approach is very much like the SDDR. The posterior proportion satisfying M_0, `numerator` = 0.0031, and the prior proportion satisfying M_0, `denominator` = 0.2022. Thus, in the prior, there is far more evidence that the αs are near 0 than in the posterior. As a result, the test finds evidence in favor of the encompassing model, M_1, which is evidence against the constraint imposed by M_0. The Bayes factor in favor of M_1, `bf_10` = 65.2258065.

Encompassing priors and SDDR are so similar, in fact, that Wetzels et al. (2010) show that as $\epsilon \to 0$, the encompassing prior approach reduces to the SDDR. Note that for small ϵ, the number of prior and posterior samples from M_1 that meet the constraints of M_0 becomes small. As a result, the estimated Bayes factor becomes noisy. As such, the SDDR is typically preferable to the encompassing prior approach for narrow intervals.

On the other hand, for the encompassing priors approach, we did not need to calculate the deviance, D, of the effect parameters. We also did not need to estimate the density of D at 0, which is often fairly low, particularly for uninformative priors, given that D can only be 0 if all αs are 0.

The encompassing priors approach also differs from the SDDR in that SDDR tests a point null, while encompassing priors tests interval nulls. This has both advantages and disadvantages. The advantage is that encompassing priors allows for a test of interval nulls. The disadvantage is that selecting the width or location of the interval is subjective and can strongly influence the results.

The encompassing priors approach can be used with `testFunction_encompassingPriors` as follows:

```
ht = testFunction_encompassingPriors(prior_eff, post_eff, I_M0)
ht$bf10
```

```
## [1] 65.22581
```

If you want pass an encompassing priors test function to `testHypothesis`, you will need to make a curried function in which the indicator function is passed to `testFunction_encompassingPriors`.

```
curriedTestFun = function(priorEffects, postEffects) {
    testFunction_encompassingPriors(priorEffects, postEffects, I_M0)
}

ht = testHypothesis(cms$prior, cms$post, factors, "let", testFunction=curriedTestFun)
ht$bf10
```

```
## [1] 65.22581
```

We can make a different constrained model just by changing the indicator function. In this case, ϵ is reduced to 0.5, which makes the constrained model more constrained.

```
narrower = function(priorEffects, postEffects) {
    I_fun = function(alpha_vect) {
        all(abs(alpha_vect) < 0.5)
    }
    testFunction_encompassingPriors(priorEffects, postEffects, I_fun)
}

ht = testHypothesis(cms$prior, cms$post, factors, "let", testFunction=narrower)
ht$bf10
```

```
## [1] Inf
```

We can also use a different kind of indicator function. Rather than making an interval around 0, we could instead use a measure of the deviance of the αs, which is conceptually more like the SDDR approach.

```
smallSd = function(priorEffects, postEffects) {
    I_fun = function(alpha_vect) {
        sd(alpha_vect) < 1
    }
    testFunction_encompassingPriors(priorEffects, postEffects, I_fun)
}

ht = testHypothesis(cms$prior, cms$post, factors, "let", testFunction=smallSd)
ht$bf10
```

```
## [1] 43.03571
```

Note that using the SDDR test function produces different, if still substantively equivalent, results.

```
ht = testHypothesis(cms$prior, cms$post, factors, "let", testFunction=testFunction_SDDR)
ht$bf10
```

```
## [1] 717.0726
```

Testing the Intercept/Grand Mean

If you want to test the value of the intercept of the model, use the `testIntercept` function. Note that this tests the intercept of whatever cell means you provide to it. Thus, if you provide cell effects like from the model in Example 2, you are testing the intercept of hose cell effects, not of the actual cell means.

The value of the intercept depends on the type of contrasts that you use. For orthogonal contrasts, the intercept is the grand mean, but it has a different value for treatment contrasts (‘contr.

```
testIntercept(cms$prior, cms$post, factors, testVal = 2.5)
```

```
## $bf01
## [1] Inf
##
## $bf10
## [1] 0
##
## $prior_pKept
## [1] 0.96
##
## $post_pKept
```

```
## [1] 0.96
##
## $success
## [1] TRUE
```

```
# You don't have to use testVal, but I do here to define the center of the interval.
intTestFun = function(prior_int, post_int, testVal) {
    I_int = function(int) {
        (testVal - 0.5) < int && int < (testVal + 0.5)
    }
    testFunction_encompassingPriors(prior_int, post_int, I_int)
}

testIntercept(cms$prior, cms$post, factors, testVal = 2.5, testFunction = intTestFun)
```

```
## $bf01
## [1] 24.54491
##
## $bf10
## [1] 0.04074164
##
## $prior_sat
## [1] 0.0334
##
## $post_sat
## [1] 0.8198
```

You can get the intercept with `getEffectParameters` by providing the special value `"(Intercept)"` for the `testedFactors` argument.

```
prior_int = getEffectParameters(cms$prior, factors, "(Intercept)")
prior_int[1:3,]
```

```
## [1]  0.637101  3.462631 -1.553516
```

Problems with Density Estimation for SDDR

When using SDDR, you must estimate the density of the prior and posterior at a point. This can be made problematic for a number of reasons. One thing that makes it problematic is when the prior has thick tails, like the Cauchy distribution. We will work with a model like in Example 2, but with the following prior on α_i.

$$\alpha_i \sim \text{Cacuchy}(0, 3)$$

```
mus = 2:7
cms = ex2_getAlphas_cauchy(mus) # Sample from prior and posterior of alphas

prior_eff = getEffectParameters(cms$prior, factors, "let")
prior_D = apply(prior_eff, 1, var)

post_eff = getEffectParameters(cms$post, factors, "let")
post_D = apply(post_eff, 1, var)

library(polspline)

# Estimating the posterior density is usually easy.
```

```
post_ls = logspline(post_D, lbound=0)
post_dens = dlogspline(0, post_ls)

# The prior density is harder when using Cauchy priors,
# because you get really large values.
# Strip off the values larger than 10 times the largest
# posterior sample.

prior_max = max(post_D) * 10
prior_kept = prior_D < prior_max

prior_ls = logspline(prior_D[ prior_kept ], lbound=0, ubound=prior_max)
prior_dens = dlogspline(0, prior_ls)

# Adjust the density to account for the removed prior samples.
prior_dens = prior_dens * mean(prior_kept)

# Calculate Bayes factors with the SDDR
bf_01 = post_dens / prior_dens
bf_10 = 1 / bf_01

bf_01 # BF that D = 0 and there is no effect
```

```
## [1] 0.005591585
```

```
bf_10 # BF that D =/= 0 and there is an effect
```

```
## [1] 178.8402
```

One thing that is important to consider is the proportion of right tail of the prior that gets cut off. The estimate of the prior density depends on what amount of the prior is actually used.

```
printPriorDensity = function(prior_max) {
    prior_ls = logspline(prior_D[ prior_D < prior_max ], lbound=0, ubound=prior_max)
    prior_dens = dlogspline(0, prior_ls)

    cat("density = ")
    cat(prior_dens)
    cat("\npKept = ")
    cat( mean(prior_D < prior_max) )
}

printPriorDensity(max(post_D) * 10^1)
```

```
## density = 0.08695192
## pKept = 0.7776
```

```
printPriorDensity(max(post_D) * 10^3)
```

```
## density = 0.06732676
## pKept = 0.977
```

```
printPriorDensity(max(post_D) * 10^5)
```

```
## density = Inf
## pKept = 0.9977
```

As you can see, as the point at which the prior is cut off (`prior_max`) changes, the prior density at 0 changes.

In addition, `pKept` changes as expected. Ideally, there would be no cutoff point. Note, however, that once the cutoff gets too large, the density estimate is no longer realistic (`Inf`). What if the top cutoff if left off?

```
tryCatch({
    logspline(prior_D, lbound=0)
}, error = function(e) {
    print(e)
})
```

```
## Warning in logspline(prior_D, lbound = 0): too much data close together
```

```
## <simpleError in oldlogspline(x, lbound = jlx): * no convergence>
```

Density estimation errors out. It is possible that a density estimation procedure specialized for estimating density at the edge of a distribution would work better, but the `polspline` package works pretty well for the most part.

This package also provides functionality to estimate the Bayes factor with the encompassing priors approach, which can avoid some of the problems with density estimation.

Modifying Test Functions

Depending on how problematic your prior and posterior distributions are, it may be difficult for `logspline` to estimate the prior and posterior densities. You may end up needing to estimate the densities in some other way. However, you may just need to change some of the default settings of the test function.

There are a few arguments of `testFunction_SDDR` that are related to some of the issues discussed in the previous section. These include `postMaxMult`, which controls where the prior is cut off relative to the maximum of the posterior, `min_pKept`, which controls how much of the prior must be kept, and `truncatePosterior`, which cuts off the same proportion of the posterior and was cut from the prior.

You can modify the default values for the arguments of `testFunction_SDDR` by creating a curried function and passing that function to the `testFunction` argument of `testHypothesis`.

```
curriedTestFunction = function(priorEffects, postEffects) {
    testFunction_SDDR(priorEffects, postEffects, postMaxMult = 3)
}

ht = testHypothesis(cms$prior, cms$post, factors, "let", testFunction = curriedTestFunction)
ht$bf10
```

```
## [1] 268.3419
```

If you need to write your own test function, just know that it takes two matrices of effect parameters. The first is the prior and the second is the posterior.

See also the Encompassing Priors section for more examples of curried test functions.

Non-Fully-Crossed/Unbalanced Designs

In short, designs that are not fully crossed (unbalanced) are supported by default, in the sense that the package does something that is correct. When dealing with unbalanced designs, however, there may be more than one correct thing to do depending on specific details about what hypothesis you want to test. You may want to do something other than what the package does by default.

This section is an exploration of why non-fully-crossed designs are weird to analyze. It concludes with some information about how to control what this package does with unbalanced designs.

Consider the following non-fully-crossed (unbalanced) design, in which there are two factors (`let` and `num`), each with three levels.

```
factors = data.frame(
    let = c('a', 'b', 'b', 'c', 'c', 'c'),
    num = c('1', '1', '2', '1', '2', '3')
)

mus    = c( 0,   0,   6,   0,   0,   0 )
```

Let's imagine that for some parameter we get the following pattern of cell means. A "." indicates a missing cell in the design.

/	1	2	3
a	0	.	.
b	0	6	.
c	0	0	0

In this design, you can conceptually think of the following effect parameters: A grand mean or intercept, main effects of the `let` factor, main effects of the numbers factor, and interaction terms.

Assume that you want to test the main effect of the `let` factor. There are two ways we can think about the main effect. The first is to consider the main effect of `let` without considering the interaction between `let` and `num`. The second is to consider the main effect in the context of the interaction. These approaches are equivalent in a fully-crossed design that uses orthogonal contrasts (the package default for fully-crossed designs), but they are not equivalent when the design is not fully crossed, in which case the contrasts cannot be orthogonal.

We will start with the first approach, examining the main effect of `let` without the interaction. I will do this by creating a design matrix and then solving for the effect parameters given the cell means.

```
X = model.matrix( ~ let, factors,
  contrasts.arg = list(let = "contr.treatment"))
X
```

```
##   (Intercept) letb letc
## 1           1    0    0
## 2           1    1    0
## 3           1    1    0
## 4           1    0    1
## 5           1    0    1
## 6           1    0    1
## attr(,"assign")
## [1] 0 1 1
## attr(,"contrasts")
## attr(,"contrasts")$let
## [1] "contr.treatment"
```

First, the design matrix, X, is created using treatment contrasts, which are valid when the design is not fully crossed. The solution for the vector of effect parameters is $\beta = (X^T X)^{-1} X^T \mu$, where μ are the cell means.

```
beta = solve(t(X) %*% X) %*% t(X) %*% mus
zapsmall(beta)
```

```
##             [,1]
## (Intercept)    0
```

```
## letb           3
## letc           0
```

We can see that the parameter corresponding to the `b` level of `let`, `letb`, is 3, while the `letc` parameter is 0. Given that treatment contrasts were used, the value for `leta` is 0, i.e. nothing is added to the `a` level of `let` beyond the intercept. Thus, the effect parameters correspond to the marginal means of 0, 3, and 0 for `a`, `b`, and `c`, respectively. If we assume very low noise in the data, there appears to be a main effect of `let`.

Now let's use the second approach and include the interaction in the design and see what happens. Let's make the design matrix, this time including both factors and their interaction.

```
X = model.matrix( ~ let * num, factors,
  contrasts.arg = list(
    let = "contr.treatment",
    num = "contr.treatment"
  )
)
X
```

```
##   (Intercept) letb letc num2 num3 letb:num2 letc:num2 letb:num3 letc:num3
## 1           1    0    0    0    0         0         0         0         0
## 2           1    1    0    0    0         0         0         0         0
## 3           1    1    0    1    0         1         0         0         0
## 4           1    0    1    0    0         0         0         0         0
## 5           1    0    1    1    0         0         1         0         0
## 6           1    0    1    0    1         0         0         0         1
## attr(,"assign")
## [1] 0 1 1 2 2 3 3 3 3
## attr(,"contrasts")
## attr(,"contrasts")$let
## [1] "contr.treatment"
##
## attr(,"contrasts")$num
## [1] "contr.treatment"
```

This design matrix, however, has too many columns (9) for the number of cells in the design (6), making it be not full rank (one outcome of which is that X^TX is uninvertable, which it needs to be). I'll take advantage of a little trick to drop excess columns.

```
q = qr(X)
kept = q$pivot[ 1:q$rank ]
X = X[ , kept ]
X
```

```
##   (Intercept) letb letc num2 num3 letb:num2
## 1           1    0    0    0    0         0
## 2           1    1    0    0    0         0
## 3           1    1    0    1    0         1
## 4           1    0    1    0    0         0
## 5           1    0    1    1    0         0
## 6           1    0    1    0    1         0
```

Now X is full rank. You can see from the columns of X that there is one interaction term, `letb:num2`. This term corresponds to the cell of the design with mean 6. We can calculate the effect parameters as before.

```
beta = solve(t(X) %*% X) %*% t(X) %*% mus
zapsmall(beta)
```

```
##             [,1]
## (Intercept)    0
## letb           0
## letc           0
## num2           0
## num3           0
## letb:num2      6
```

Now when we look at the parameter values, we can see that all of the `let` main effect parameters are 0 because all of the marginal effect of cell `b2` is accounted for by the interaction. Thus, there does not appear to be a main effect of `let`.

Here is a different example.

/	1	2	3
a	0	2	4
b	0	.	.
c	0	.	.

We can see that there is a marginal main effect of `let` when nothing else is considered. However, when the main effect of `num` is included in the design, the parameters associated with the `2` and `3` levels of the `num` factor will account for the apparent marginal `let` effect. Thus, main effects can be affected by the inclusion of other main effects.

```
factors = data.frame(
  let = c('a', 'a', 'a', 'b', 'c'),
  num = c('1', '2', '3', '1', '1')
)
mus   = c( 0,   2,   4,   0,   0)

X = model.matrix( ~ let + num, factors,
  contrasts.arg = list(
    let = "contr.treatment",
    num = "contr.treatment"
  )
)

beta = solve(t(X) %*% X) %*% t(X) %*% mus
zapsmall(beta)
```

```
##             [,1]
## (Intercept)    0
## letb           0
## letc           0
## num2           2
## num3           4
```

When working with non-fully-crossed designs, you have to choose what main effects or interactions to account for. By default, `testHypothesis` takes the first approach: Only estimate the main effect, but not interactions. More generally, it only includes terms up to and including the tested effect. For example, if you are testing two-factor interaction, it will estimate that interaction and the two associated main effects. If there is a third factor, it will not estimate the main effect of that third factor or any interactions including that third factor.

To get more control over what effects are estimated, you can provide the `dmFactors` argument to `testHypothesis` or `getEffectParameters`. Here I use a formula to specify that I want to use a design

matrix with both main effects.

```
getEffectParameters(matrix(mus, nrow=1), factors, "num", dmFactors= ~ let + num)
```

```
##      num.1 num.2 num.3
## [1,]     0     2     4
```

See also the `contrastType` argument of `testHypothesis` and `getEffectParameters`.

Subsetting Designs

What if you want to use only a subset of the cells in your design? Easy, just exclude the cells you don't want to use. You do this by dropping rows of the `factors data.frame` and by dropping the same columns of the cell means/effects. The example below will demonstrate this.

Tests of Simple Effects

A case in which you might want to subset a design is if you want to do follow-up tests of simple effects when there is an interaction present in the data.

```
factors = data.frame(
    let = c('a', 'a', 'b', 'b'),
    num = c('1', '2', '1', '2')
)

# Build in an interaction
mus = c( 1, 5, 3, 3 )

cms = ex2_getAlphas(mus)

testHypothesis(cms$prior, cms$post, factors, c("let", "num"))$bf10
```

```
## [1] Inf
```

Now that we know that there is an interaction in the design, we should not interpret main effects. Instead, we should test for main effects of one factor within each level of the other factor to determine where effects are present.

```
usedCells = (factors$let == 'a')
testHypothesis(cms$prior[, usedCells], cms$post[,usedCells], factors[usedCells,], "num")$bf10
```

```
## [1] 407.6146
```

```
usedCells = (factors$let == 'b')
testHypothesis(cms$prior[, usedCells], cms$post[,usedCells], factors[usedCells,], "num")$bf10
```

```
## [1] 0.3661002
```

Nested Design

You might have a design in which the levels of one factor are nested within the levels of another factor. For example, assume a design in which the `a` level of the `let` factor has levels 1 to 4 of the `num` factor, but level `b` of the `let` factor has levels 1 and 2 of the `num` factor. The design is such that `num` 1 has a different meaning depending on whether it was paired with `let a` or `b`. As such, you must analyze each level of `let` individually.

```r
factors = data.frame(
    let = c('a', 'a', 'a', 'a', 'b', 'b'),
    num = c('1', '2', '3', '4', '1', '2')
)

mus = c( 2, 3, 5, 5,  3, 2 )

cms = ex2_getAlphas(mus)

usedCells = (factors$let == 'a')
testHypothesis(cms$prior[, usedCells], cms$post[,usedCells], factors[usedCells,], "num")$bf10
```

```
## [1] 14.3072
```

```r
usedCells = (factors$let == 'b')
testHypothesis(cms$prior[, usedCells], cms$post[,usedCells], factors[usedCells,], "num")$bf10
```

```
## [1] 0.43428
```

References

Hardman, K. O., Vergauwe, E., & Ricker, T. J. (2017). Categorical working memory representations are used in delayed estimation of continuous colors. *Journal of Experimental Psychology: Human Perception and Performance*, *43*(1), 30–54. http://doi.org/doi:10.1037/xhp0000290

Wagenmakers, E.-J., Lodewyckx, T., Kuriyal, H., & Grasman, R. (2010). Bayesian hypothesis testing for psychologists: A tutorial on the savage-dickey method. *Cognitive Psychology*, *60*(3), 158–89. http://doi.org/10.1016/j.cogpsych.2009.12.001

Wetzels, R., Grasman, R. P., & Wagenmakers, E.-J. (2010). An encompassing prior generalization of the savage-dickey density ratio. *Computational Statistics and Data Analysis*, *54*, 2094–2102. http://doi.org/10.1016/j.csda.2010.03.016