Ratcliff Diffusion Model Worksheet & Assignment

Henrik Singmann

Compiled at: 2019-01-23

This document consists of two parts. The first part contains a worksheet that intends for you to get comfortable with `rtdists` for working with diffusion models in `R`. We will use the two seminar sessions to go through this worksheet together (everyone on their own time, but with us being around for questions). The worksheet should set you up to be able to do the assignment.

The second part of this document contains the instructions to the written assignment. Once you are through with the worksheet, feel free to start with the assignment. You can also ask us questions about it anytime. If you feel so, you can also first look at the assignment now. But without having went through the worksheet first, parts of the assignment might look difficult.

To get a general introduction into cognitive modeling, we recommend the recent manuscript by Wilson and Collins (2019) "Ten simple rules for the computational modeling of behavioral data" which is freely available at: https://psyarxiv.com/46mbn/
A shorter, but more advanced set of recommendations can be found in Heathcote, Brown, and Wagenmakers (2015) which is freely available at: http://www.ejwagenmakers.com/inpress/HeathcoteModelingIntro.pdf

Worksheet

Setup

For this worksheet and the assignment, we will be needing `R` package `rtdists`. So your first job is to install it:

```
install.packages("rtdists")
```

As some of you have already noticed, `rtdists` is also one of my packages. I have developed it together with some researcher who work on response time models from Australia (notably Andrew Heathcote and Scott Brown). `rtdists` contains the probability density function (PDF), cumulative probability function (CDF), quantile function, and random, number generator for the Ratcliff diffusion model.

After installing it, please attach (or load) `rtdists` and also the `tidyverse`. We do the latter to make our lives easy along the way. We also set some reasonable default for the `ggplot2` theme.

```
library("rtdists")
library("tidyverse")
theme_set(theme_bw(base_size = 15) +
            theme(legend.position="bottom",
                  panel.grid.major.x = element_blank(),
                  panel.grid.minor.x = element_blank()))
```

If you get a warning message that `rtdists` was build with an `R` version newer than the one you have (e.g., built with `3.5.2`), this reminds you that it is time to update `R` (get the latest version from `CRAN`). Maybe not now during the seminar, but afterward.

After attaching `rtdists`, you can check out the help page for the diffusion model functions:

```
?ddiffusion
```

Simulating data

Often, a researcher starts with a research question, runs an experiment to collect relevant data, and, with this data in hand, fits a model to the data. This model can be a statistical model, such as an ANOVA, or a cognitive model, such as the diffusion model. In either case, the researcher can then interpret the parameter estimates with regards to the research question. To be able to do so, she or he needs to understand what differences in parameters value actually mean: What effect do different parameter values have on the behaviour predicted by the model? One privileged way to do so is via simulation; that is, creating artificial data from the model and inspecting the obtained data.

This is very much in line with Wilson and Collins (2019), whose third rule is simply "Simulate, simulate, simulate!". They furthermore write:

> Simulate the model across the range of parameter values. Then, visualize behavior as a function of the parameters. Almost all models have free parameters. Understanding how changes to these parameters affect behavior will help you to better interpret your data and understand individual differences in fit parameters.

To simulate data from the diffusion model we can use `rdiffusion`. Let us start by simulating 500 data points from a very standard 4-parameter diffusion model. Note that this model without the between-trial variabilities is usually not called the Ratcliff diffusion model, but often just the Wiener model. Roger Ratcliff's initial contribution was to add the between-trial variability of the drift rate to the diffusion model. However, we set this to 0 for now.

To make our lives a bit easier, we create some variables that hold all parameters values we want to use for the simulation as well as the number of simulated responses we want to obtain. This makes it easy to use them later on again. We start with $v = 1.5$, $a = 1.2$, $t0 = 0.3$, and no bias (i.e., z can remain as the default value which corresponds to $a/2$).

We also set the seed before simulating the data to ensure everyone can exactly recreate the simulation results.

```
set.seed(23)
N <- 500 ## number of simulated responses
v1 <- 1.5
a1 <- 1.2
t01 <- 0.3
d1 <- rdiffusion(N, a = a1, v = v1, t0 = t01)
str(d1)
```

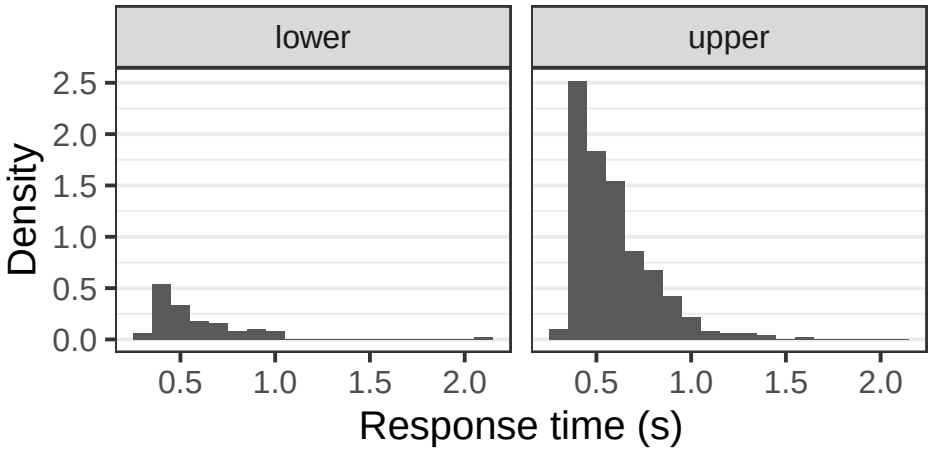
```
## 'data.frame':    500 obs. of  2 variables:
##  $ rt      : num  0.525 0.381 0.421 0.609 0.712 ...
##  $ response: Factor w/ 2 levels "lower","upper": 2 2 2 2 2 2 2 2 2 2 ...
```

As can be seen, `rdiffusion` returns a `data.frame` with two columns. The first column, `rt`, contains the simulated response time (in seconds). The second column, `response`, contains the simulated response as a factor with two levels, `upper` if the diffusion process reached the upper boundary, or `lower`. If the drift rate is positive, upper responses can be understood as correct responses and lower responses as errors.

Plot RT Data

One way to inspect this data is to plot it. As discussed before, this is generally a good approach to start any type of data analysis. Here it makes sense to plot the conditional RT distribution as a histogram:

```
ggplot(d1, aes(x = rt)) +
  geom_histogram(aes(y=stat(count/sum(count)*1/width)), binwidth = 0.1) +
  facet_wrap(~response) +
  ylab("Density") + xlab("Response time (s)")
```



Note that in this plot I used a trick, `aes(y=stat(count/sum(count)*1/width))`, which ensures that the density across panels sums to one and not within each plot (to be honest, figuring this trick out to me quite a bit of time). That is, each panel shows a "defective" histogram and only the full figure displays a proper density histogram. Be aware that this might not always be a desired behaviour. For example, should one use the same code for a plot with more than two panels, the density would also some to 1 for the whole figure, which is probably not very reasonable. Also note that this code will also work for different settings of `binwidth` or when specifying `bins` instead of `binwidth`. Here, `binwidth = 0.1` means that every bin encompasses a response time window of 0.1 seconds or 100 ms.

Plot the PDF

The decision to create "defective" histograms for each response option has the reason that this allows us easily to overlay the empirical histogram (i.e., empirical in the sense that it is based on the simulated data) with the theoretical PDF of the diffusion model. We can then visually compare data and model predictions.

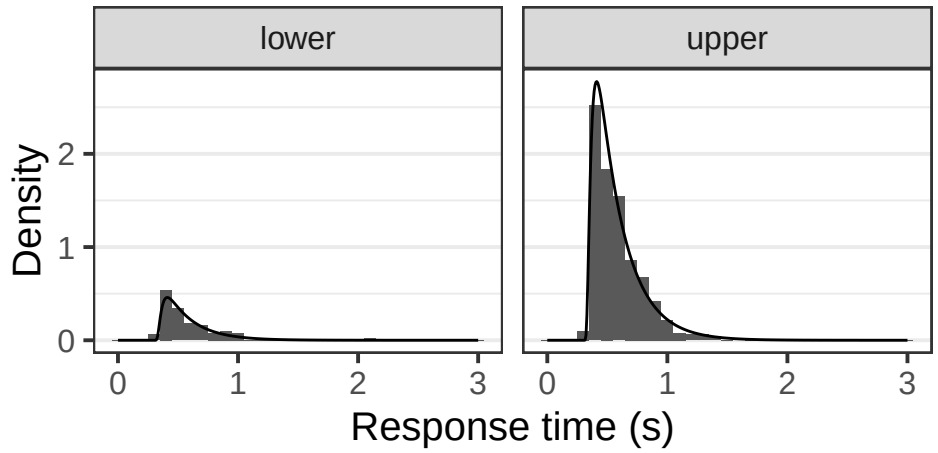
The easiest way to get this overlay is to use `ddiffusion` and to create the `x` and corresponding `y` coordinates that form the PDF via a self-written `function`. In addition to the diffusion model parameters, this function needs to have arguments to define the number of points we will be using to draw the line (which governs the resolution or precision of the line), as well as the start and end points of the line. Ideally, we can fix these to reasonable default values that we do not need to change later. The function also transform the predictions into a `tidy tibble` so we can directly plot it together with the (simulated) data. Feel free to copy and paste this code from the PDF (in case it does not copy the indentation correctly, use `CTRL`/`CMD` + `I` to auto-indent the code).

```
get_diffusion_df <- function(a, v, t0, z = a/2, sz = 0, sv = 0, st0 = 0,
                             n = 401, start = 0, end = 3) {
  rt <- seq(start, end, length.out = n)
  tibble(
    rt = rt,
    upper = ddiffusion(rt, response = "upper", a = a, v = v, t0 = t0, z = z,
                       sz = sz, sv = sv, st0 = st0),
    lower = ddiffusion(rt, response = "lower", a = a, v = v, t0 = t0, z = z,
                       sz = sz, sv = sv, st0 = st0)
  ) %>%
    gather(key = "response", value = "Density", upper, lower) %>%
    mutate(response = factor(response, levels = c("lower", "upper")))
}
```

We can then use this function to generate the theoretical density for the parameter values used to generate

the data in `d1` and then overlay the histogram with the theoretical PDF.

```
pred_d1 <- get_diffusion_df(a = a1, v = v1, t0 = t01)
ggplot(d1, aes(x = rt)) +
  geom_histogram(aes(y=stat(count/sum(count)*1/width)), binwidth = 0.1) +
  geom_line(data = pred_d1, aes(y = Density, group = 1)) +
  facet_wrap(~response) +
  ylab("Density") + xlab("Response time (s)")
```



Not too surprisingly, the theoretical PDF and the empirical PDF match quite strongly. Given that we simulated quite a bit of data (i.e., 500 data points) anything else would have also been concerning. Theoretical model predictions and random data generated from the model with the same parameter values should of course coincide in the limit (i.e., if enough data is simulated).

Compare Empirical And Theoretical Statistics

Comparing the observed and predicted RT distributions visually as done above is maybe not the most rigorous approach. It is sometimes not really clear if a difference is meaningful or not and depending on the number and size of the bins, the visual impression can change. A better way to evaluate the adequacy of a model (i.e., evaluate the model fit) is by comparing empirical and theoretical statistics summarizing the performance of both data and model.

We first start with calculating the empirical statistics. We can use some `tidyverse` magic to look at the proportion of "correct" responses and quantiles of the RT distribution. Here and in the following we use quantiles as this is what allows us easiest to compare model predictions and data. Here, we only look at the 10% quantile (i.e., fast responses or leading edge of the distribution), the 50% quantile (i.e., the median as a measure of central tendency), and the 90% quantile (i.e., slow responses or the tail of the distribution). Note that in the literature you can additionally also find comparisons of the .3 and .7 or .25 and .75 quantiles.

```
d1 %>%
  group_by(response) %>%
  summarise(acc = n()/nrow(d1),
            q10 = quantile(rt, probs = 0.1),
            q50 = quantile(rt, probs = 0.5),
            q90 = quantile(rt, probs = 0.9))
```

```
## # A tibble: 2 x 5
##   response   acc   q10   q50   q90
##   <fct>    <dbl> <dbl> <dbl> <dbl>
## 1 lower    0.156 0.365 0.494 0.889
```

```
## 2 upper    0.844 0.400 0.533 0.860
```

We see that in the simulated data the accuracy is around 84%. We also see a potentially surprising pattern when comparing RT quantiles across both responses. The differences between correct (i.e., `upper`) and errors (i.e., `lower`) responses is that the errors are somewhat faster for the fast and central quantile (.1 and .5), whereas the correct responses are somewhat faster in the tail. As mentioned in the lecture, without between-trial variabilities, these differences between error RTs and correct RTs are purely sampling noise as the model predicts equally fast quantiles as shown below.

To get the theoretical rate of responses hitting the upper boundary, we can use `pdiffusion`. If it is evaluated at `Inf` (Infinity), it tells us the maximum proportion of the corresponding response we can expect given a set of parameters.

```
pdiffusion(Inf, response = "upper",
           a = a1, v = v1, t0 = t01)
```

```
## [1] 0.858
```

Here, the difference between the predicted and simulated is not too much, a bit more than 1%. Please note that the predicted proportion of lower responses is of course 1 minus the value for the upper response.

To get the theoretical RT quantiles, we use the quantile function `qdiffusion`. We simply need to pass the desired quantile as well as the parameter values to the function. Note that we also need to set `scale_p = TRUE`. This ensures that the quantile is scaled for the maximal probability that can be reached for each response (i.e., the conditional quantile for each response) and not the overall quantile given the full distribution. This maximal probability value is of course the value we have just calculated above, so we could also scale the quantiles for the `upper` response by that value, but simply setting `scale_p = TRUE` is a bit easier.

```
qdiffusion(c(0.1, 0.5, 0.9), response = "upper",
           a = a1, v = v1, t0 = t01,
           scale_p = TRUE)
```

```
## [1] 0.383 0.522 0.875
```

```
qdiffusion(c(0.1, 0.5, 0.9), response = "lower",
           a = a1, v = v1, t0 = t01,
           scale_p = TRUE)
```

```
## [1] 0.383 0.522 0.875
```

In line with the description above, predicted RT quantiles are identical for `upper` and `lower` responses in the present case as there is no bias (i.e., $z = a/2$) and the between-trial variabilities are equal to 0. In addition, we find that there is quite a bit of agreement between theoretical and empirical quantiles.

Your Task

Your first task is now to extend this simulation to make sure the following five patterns discussed in the lecture of how the diffusion model relates parameter values and predictions hold:

- Increasing drift rate makes faster and more accurate responses
- Increasing boundary separation makes slower and more accurate responses
- Increasing non-decision time makes slower responses (no effect on accuracy)
- Increasing bias makes one response more frequent and faster, other response less frequent and slower

Note, there is not really one right answer here. You might use random number generation via `rdiffusion`, you can look at the predicted PDF via `ddifusion`, you can look at empirical or theoretical statistics (the latter with `pdiffusion` or `qdiffusion`), or you might combine any of those. The goal is more for you to play around a bit with the diffusion model parameters and the different functions to see what effect on the predicted behaviour different parameter settings have.

Fitting Data with MLE

As described above, the most common application of the diffusion model and other cognitive models is the other way around then described in the previous part. We have a fixed data set and a model and want to know which parameter values could potentially be responsible for generating this data or are most likely given the data. To do so, we fit the model to the data and this is what we want to work on now.

To fit the model, we need an index of model fit; that is, a method that allows us to evaluate how well a given set of parameter values describes the data. For example, one common way to think about model fit is in terms of the distance (or better, squared distance) between observed and predicted values. With such a measure of model fit, model fitting entails applying an algorithm that searches for the parameter values that minimize this distance. Unfortunately, in many situations it is not easy to calculate fit based on distance. For example, the diffusion model does not make predictions for individual data points, but only for response time distributions. Hence, there is no way to calculate a distance based measure in the present case.

Instead, we will be using a more general method that can be applied any time a model can be expressed in terms of a probability distribution (i.e., has an analytical PDF), maximum likelihood estimation or MLE for short. MLE uses the probability density function (PDF) for fitting. However, it uses it in a somewhat "non-standard" way. Remember, the PDF is defined as a function that, given a fixed set of parameter values, integrates to 1 across all possible data. This of course ensures that the total probability of observing any possible data is exactly 1. MLE turns this situation around. We now assume that not the parameters are fixed, but the data. MLE means to search for the parameters that maximize the PDF given fixed data. Note that in this case we call the PDF the likelihood function as it looses the property to sum to 1 across the search space.

Let's describe this again in slightly different words. MLE searches for those parameter values that maximize the likelihood function given a fixed data set. You might think that as a consequence, the resulting parameters values are those that are most likely given the data, but this would be an incorrect interpretation. Instead, the resulting parameter values (or maximum likelihood estimates) are those parameter values for which the data is most likely. As is common in frequentist statistics, we cannot actually make the inference we would like to (i.e., say that these are the most likely parameter values). Instead, we have to live with the somewhat awkward inference that the ML estimates are those for which the observed data is most likely. Also note that likelihood is a relative measure of model fit, so in contrast to the distance measure described initially, knowing the value of the likelihood function at the maximum likelihood estimate does not tell us something about how good the fit is in absolute terms. For this we need other methods.

The Likelihood Function

To use MLE, we need a function that returns the negative sum of the log-likelihoods of the data given a set of parameter values. This type of function generally needs two arguments, one for passing the parameters and one for passing the data. Here, our data consists of two vectors (response times and associated decisions), so we pass both individually (i.e., the function will have three arguments). For technical reasons that will become clear later, the parameters need to be passed as one argument, which needs to be the first argument of our function, needs to contain all parameters as a vector, and which we will call `pars`. I think it is easiest if `pars` is named to make it easy to extract the correct parameter values inside of the function (e.g., `pars["a"]` allows you to extract the value of `a` parameter from the function). An example for a `pars` that could be passed to our function is given below and called `par1`. Your first task is to create the likelihood function now, let us call it `ll_diffusion`.

Note that this function should be able to handle the special case of one of the individual likelihoods (i.e., densities for each individual data points) being equal to 0. In this case, it should simply return a really high value (e.g., `return(1e6)`) and not return the negative sum of the log of the individual likelihoods. The reason for this is that the log of 0 is negative infinity which cannot be handled by the optimization function we want to use (and likelihood values of 0 can occur for example if `t0` is larger than an observed RT). **Your**

task now is to write the likelihood function. The code below should be a start point, but you still have to fill out the details (i.e., the body of the function).

```
par1 <- c(a = 0.8, v = 1, t0 = 0.1)
ll_diffusion <- function(par, rt, response) {
  ### must return a scalar value: the negative sum of log-likelihoods
}
```

When we call this function with the following arguments (where `par1` is the vector as assigned above and `d1` is the first data we randomly generated in the previous section), the following output should be obtained if your likelihood function is correct:

```
ll_diffusion(pars = par1, rt = d1$rt, response = d1$response)
```

```
## [1] 1063
```

Finding the ML Estimate

With this function in hand, we can now use an optimization algorithm that uses numerical methods (such as gradient decent) to find the maximum likelihood estimate for our data. Here, we will be using `nlminb`, as I have had good experience with this function over the years. Similar functions are `optim` in `R` or `fminsearch` in `Matlab` and there exist a lot more in contributed `R` packages (e.g., in `nloptr` or `minqa`). One of the benefits of `nlminb` is that it allows user specified parameters bounds which we will need here (as both `a` and `t0` cannot be smaller than 0). `nlminb` then requires us to pass a set of start values for the search (e.g., our `par1` vector), the "objective" function that should be minimized (i.e., the likelihood function from above), additional arguments passed to the likelihood function, and the bounds. Note that optimization functions usually minimize and that is why our likelihood function returns the negative log-likelihood. The maximum log-likelihood estimate can then simply be obtained by minimizing the negative log-likelihood.

```
res1 <- nlminb(par1, ll_diffusion,
       rt = d1$rt, response = d1$response,
       lower = c(0, 0, -Inf))
res1
```

```
## $par
##     a     v    t0
## 1.183 1.440 0.304
##
## $objective
## [1] 5.79
##
## $convergence
## [1] 0
##
## $iterations
## [1] 22
##
## $evaluations
## function gradient
##       38       79
##
## $message
## [1] "relative convergence (4)"
```

Note that `nlminb` returns a `list`, so we can access individual elements via the `$` operator. For example the parameter estimates:

```
res1$par
```

```
##     a     v    t0
## 1.183 1.440 0.304
```

To get the log-likelihood value at the maximum, we now need to take the negative value of the objective function at its maximum:

```
-res1$objective
```

```
## [1] -5.79
```

The output also contains some diagnostic information such as the number of iterations and evaluations of the likelihood function (`nlminb` evaluates the objective function several times at each iteration to estimate the gradient at the current location). The most important bit is probably `convergence` (and maybe `message`). If `convergence` is not 0 this indicates that the algorithm is not happy with the solution it has obtained. This means that this is either not the maximum likelihood estimate (i.e., not global minimum, but a local one) or the data does not provide a maximum likelihood estimate that can be found with the current algorithm (which usually means, also not with other algorithms).

Avoiding Local Optima

To rule out the possibility that the solution is not the global optimum, but a local optimum, we usually want to run our optimization algorithm several times. However, optimization algorithms of the type discussed here are deterministic. That means, given the same start values (and other arguments) it will produce identical results. Consequently, running the code above several times will of course lead to the same results. Instead, we should run the code above several times with different initial start values. The easiest way to do so is with a function that returns random start values such as the following one.

```
get_start_values <- function() {
  c(
    a = runif(1, 0.5, 3),
    v = rnorm(1, 0, 2),
    t0 = runif(1, 0, 0.2)
  )
}
```

We can then replace the start argument in `nlminb` simply with a call to this function. If we run this function a few times and every time get essentially the same parameter values and value of the objective function, this provides quite a bit of evidence that the result is not a local optimum, but the global optimum (i.e., the maximum likelihood estimate).

```
nlminb(get_start_values(), ll_diffusion,
       rt = d1$rt, response = d1$response,
       lower = c(0, 0, -Inf))
```

```
## $par
##     a     v    t0
## 1.183 1.440 0.304
##
## $objective
## [1] 5.79
##
## $convergence
## [1] 0
##
```

```
## $iterations
## [1] 22
##
## $evaluations
## function gradient
##       30       74
##
## $message
## [1] "relative convergence (4)"
```

Here, the optimization problem is not very complicated so the obtained parameter values and value of the objective function should be the same most time the function is run. The only difference should be the number of iterations and evaluations. However, sometimes we could still get different results.

In real life examples (i.e., where we do not use generated data) the optimization problem might be a bit more complicated. Thus, it might always be a good idea to run the optimization several times. One way to do so could be by using `rerun` from the `purrr` package (which is also part of the `tidyverse`). With this, we could do something like the following:

```
res2 <- rerun(5, nlminb(get_start_values(), ll_diffusion,
                        rt = d1$rt, response = d1$response,
                        lower = c(0, 0, -Inf))) %>%
  map_dfr(~as_tibble(cbind(t(.$par),
                           logLik = -.$objective,
                           convergence = .$convergence)))
res2
```

```
## # A tibble: 5 x 5
##       a     v       t0  logLik convergence
##   <dbl> <dbl>    <dbl>   <dbl>       <dbl>
## 1  1.18  1.44 3.04e- 1   -5.79           0
## 2  1.18  1.44 3.04e- 1   -5.79           0
## 3  1.18  1.44 3.04e- 1   -5.79           0
## 4  1.18  1.44 3.04e- 1   -5.79           0
## 5  2.02  1.18 8.01e-13 -294.             0
```

The first argument to `rerun` tells how often we want to run the `nlminb` call. `rerun` then returns a `list` of `lists` returned from `nlminb`. We use some additional `purrr`, `tidyverse`, and base `R` to transform each result into one `tibble` that are then bound together. Note that I have also added a minus to the value of the objective function so the value in the `logLik` column is actually the log-likelihood value.

The returned `tibble` allows us to inspect the results across different fitting runs. Here, we can see that for 4 of the 5 runs the result is pretty much the same. However, for the result for the last run is different and the log-likelihood lower suggesting that this result is a local minimum.

We can now extract the result from the best fitting run. And even though 4 of the 5 look the same, one of them has a slightly higher log-likelihood value. We can get the corresponding index via `which.max()`.

```
which.max(res2$logLik)
```

```
## [1] 4
```

We can then easily extract the best fitting parameter values with this information.

```
mle <- res2[which.max(res2$logLik), 1:3  ]
mle
```

```
## # A tibble: 1 x 3
##       a     v    t0
```

```
##   <dbl> <dbl> <dbl>
## 1  1.18  1.44 0.304
```

Detecting Identifiability Issues

We can also check which of the five results is approximately equal to the optimum. For this, we `round` all `logLik` values to three decimal places (we could also use 2 decimal places) and then check for equality.

```
round(max(res2$logLik), 3) == round(res2$logLik, 3)
```

```
## [1]  TRUE  TRUE  TRUE  TRUE FALSE
```

We could use this code also to check if all parameters are identified; that is, whether the parameter estimates are the same given the same log-likelihood value. This seems to be the case here, but given little data or other numerical problems this does not have to be the case. One way to check for identifiability of the parameter estimates is to pick the first log-likelihood values that is approximately equal to the maximum likelihood value (and not the maximum) and compare the parameter estimates with the estimates of the best fit. If their absolute difference is larger than a reasonable threshold (say larger than .01), we would be inclined to say that this parameter is likely not identifiable given the current data set. For example, if we were to compare parameter estimates across two conditions, we would want to exclude all non-identifiable parameter estimates from such a comparison. For example, we could obtain a `tibble` that only contains the identifiable parameter estimates in the following manner.

```
which_max <- which(round(max(res2$logLik), 3) == round(res2$logLik, 3))
## exclude actual ML estimate:
which_max <- which_max[which_max != which.max(res2$logLik)]

## copy ML estimates
mle2 <- mle
## remove all estimates that are not identifiable:
mle2[abs(mle - res2[which_max[1], 1:3 ]) > 0.01] <- NA
mle2
```

```
## # A tibble: 1 x 3
##       a     v    t0
##   <dbl> <dbl> <dbl>
## 1  1.18  1.44 0.304
```

Comparing Data and Model Predictions

One interesting result from fitting the diffusion model to the generated data is that the ML parameter estimates are quite similar to, but still different from, the generating parameter values. Given that we generated random data, such a result is of course expected. **Your next task** is now to do what we would normally do when fitting a model to data and use the ML estimates to generate theoretical predictions from the model. Namely, obtain predicted response proportions using `pdiffusion` and quantiles using `qdiffusion` and compare them to the quantiles obtained from the generated data.

Fitting Multiple Participants

Often we have data from multiple participants and not only from one participant. In such a situation, we often want to fit the data for each individual participant and collect the estimates across participant for later use. Consider for example the following situation in which we generate 5 data sets, each from the same parameter values, but only with 200 data points each. To ensure reproducibility, we set the random seed before generating the data.

```
set.seed(90)
N <- 200 ## number of simulated responses
Nd <- 5  ## number of simulated data sets
v1 <- 1.5
a1 <- 1.2
t01 <- 0.3
dmulti <- map(seq_len(Nd), ~cbind(run = ., rdiffusion(N, a = a1, v = v1, t0 = t01))) %>%
  bind_rows() %>%
  as_tibble() %>%
  mutate(run = factor(run))
str(dmulti)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    1000 obs. of  3 variables:
##  $ run     : Factor w/ 5 levels "1","2","3","4",..: 1 1 1 1 1 1 1 1 1 1 ...
##  $ rt      : num  0.502 0.812 0.462 0.963 0.366 ...
##  $ response: Factor w/ 2 levels "lower","upper": 2 2 2 2 2 2 2 2 2 2 ...
```

Your task is now to fit the data of the five individual runs independently and look at the parameter estimates across runs. One way to fit the data across participants would be to use a `for`-loop. However, this requires you to create an empty `data.frame` (or `list`) that will hold the result in each iteration. An easier possibility might be to use some good old `tidyverse` magic combined with some of the code we have used above. The `tidyverse` magic I am thinking about is of course what we have used for doing a no-pooling analysis before. For example, one of the following two could be a good starting point (note that the `ungroup` after the `do` might not be necessary for your code, but for the way I wrote it, it was necessary):

```
res_multi <- dmulti %>%
  group_by(run) %>%
  do(fits = ...) %>%
  ungroup()

res_multi <- dmulti %>%
  group_by(run) %>%
  do(...) %>%
  ungroup()
```

In my solution I used quite a bit of the code from above and fitted the model five times to each individual data set and then returned the estimates from the best fitting run, but only after checking for identifiability. When doing so, I obtained the following output:

```
## # A tibble: 5 x 5
##   run   fits                a     v    t0
##   <fct> <list>          <dbl> <dbl> <dbl>
## 1 1     <tibble [5 x 5]>  1.21  1.64 0.299
## 2 2     <tibble [5 x 5]>  1.20  1.27 0.297
## 3 3     <tibble [5 x 5]>  1.20  1.66 0.292
## 4 4     <tibble [5 x 5]>  1.17  1.75 0.311
## 5 5     <tibble [5 x 5]>  1.21  1.59 0.300
```

As can be seen, the estimates of `a` and `t0` almost perfectly recover the true data generating value. For `v1` it looks a bit different. The estimates are still near the true value, but their distance is a bit further. You might want to check how this results look like if instead of simulating 200 data points per data set, only simulating data from 100 data points.

Parameter Recovery Studies

What we have done in the last step is known as a *parameter recovery study* and generally strongly recommended (in Wilson & Colin, 2019, it is rule 5, in Heathcote et al., 2015, it is section 2.2). A parameter recovery study can be seen as a best-case scenario for a model. It answers the question if the model can actually recover the data generating parameters if we simulate data from the model. If a model is not able to do that, we know that we cannot really trust the parameter values. How well a model can recover its parameter values, depends of course on the model itself, but also on the specific settings used for the recovery study. What are the true data generating parameters, how big is the N, how many parameters are considered? In a real parameter recovery study, we might of course use more than only 5 generated data set, but here it already gives us some idea.

Your last task of this worksheet is to extend the present parameter recovery study to more parameters. Generate 5 (or 10) data sets from a diffusion model in which you specify 6 diffusion model parameters instead of only 3. Specifically, I want you to simulate from the following parameter values:

```
a <- 1.5
z <- 0.7
t0 <- 0.35
sv <- 1.1
st0 <- 0.3
v <- 0.95
```

In your simulation, please also use an `N` per data set of 200. And note that you will use a new function for generating random start values as well as a new likelihood function (both now need to work with this 6 parameters). I will again provide you with the function for generating random start values.

```
get_start_values2 <- function() {
  c(
    a = runif(1, 0.5, 3),
    t0 = runif(1, 0, 0.5),
    v = rnorm(1, 0, 2),
    z = runif(1, 0.4, 0.6),
    sv = runif(1, 0, 0.5),
    st0 = runif(1, 0, 0.5)
  )
}
```

Do you feel the generating parameter values can be well recovered in this case?

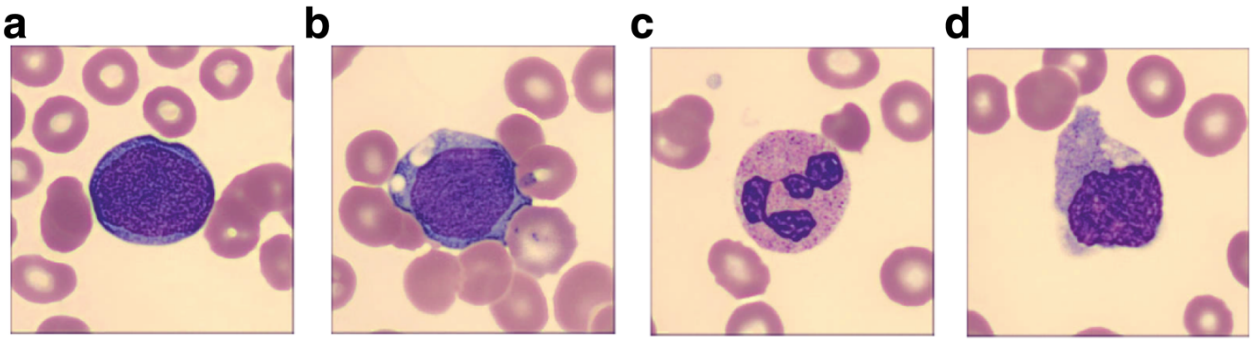


Figure 1: Sample images of blast and non-blast cells that were classified as easy and difficult. Panel a is an easy blast image, panel b is a hard blast image, panel c is an easy non-blast image, and panel d is a hard non-blast image.

Assignment

This assignment is due **Wed, March 20, 2019**. Submit your response via moodle as one file.

Medical Decision Making Data Set

File `medical_dm.csv` contains part of the data presented in Trueblood et al. (2017), also discussed in the lecture, investigating medical decision making among medical professionals (pathologists) and novices (i.e., undergraduate students). The task of participants was to judge whether pictures of blood cells show cancerous cells (i.e., blast cells) or non-cancerous cells (i.e., non-blast cells). The current data set contains 200 such decision per participant. At the beginning of the experiment, both novices and medical experts completed a training to familiarize themselves with blast cells. After that, each participant performed the main task in which they judged whether or not presented images were blast cells or non-blast cells. Among them, some of the cells were judged as easy and some as difficult trials by an additional group of experts. Figure 1 contains examples of blast cells and non-blast cells. Here, we only consider the data from the "accuracy" condition (i.e., Trueblood et al. considered additional conditions that are not part of the current data set).

```
med <- read_csv("medical_dm.csv")
glimpse(med)
```

```
## Observations: 11,000
## Variables: 9
## $ id             <chr> "002", "002", "002", "002", "002", "002", "002"...
## $ group          <chr> "experienced", "experienced", "experienced", "e...
## $ block          <dbl> 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,...
## $ trial          <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ...
## $ classification <chr> "blast", "non-blast", "non-blast", "non-blast",...
## $ difficulty     <chr> "easy", "easy", "hard", "hard", "easy", "easy",...
## $ response       <chr> "blast", "non-blast", "blast", "non-blast", "bl...
## $ rt             <dbl> 0.853, 0.575, 1.136, 0.875, 0.748, 0.706, 0.686...
## $ stimulus       <chr> "blastEasy/BL_10166384.jpg", "nonBlastEasy/1625...
```

The data contains 9 variables:

- `id`: participant identifier (note, participant identifier is not unique across groups).
- `group`: Group identifier with three levels: `"experienced"`, `"inexperienced"`, and `"novice"`. The first two levels refer to different type of medical professional that we will consider together (i.e., in your answer, please do not distinguish between `experienced` and `inexperienced` doctors).

- `block`: The block in which the trial was shown.
- `trial`: trial number.
- `classification`: true status of each image, either `"blast"` or `"non-blast"`.
- `difficulty`: difficulty of trial, either `"easy"` or `"hard"`.
- `response`: response of participant, either `"blast"` or `"non-blast"`.
- `rt`: response time in seconds.
- `stimulus`: stimulus shown.

Task Description

Your task is to analyse the data with a diffusion model using a no-pooling approach (i.e., fit the diffusion model to the data of each individual participant separately), `rtdists`, and trial-wise maximum likelihood estimation. Then, use ANOVA to compare the individual-level parameter estimates between the two groups (i.e., `experts` versus `novices`). There are two research questions here: Is the diffusion model able to describe real-life medical decision making for both experts (i.e., medical professionals) and novices? And if so, do the cognitive process captured by the diffusion model differ between experts and novices? Or said differently, how do cognitive processes underlying medical decision making differ between experts and non-experts?

As in PS 923, your answer to the research question and this assignment should have two separate sections. In the first section, write out your answer using complete sentences, as you might in a paper. This section should start with a description of the design and research question, the diffusion model used here (e.g., which and how many free parameters), and relevant aspects of the data (e.g., how many trials per participant on average, how many trials were excluded). Next, your answer should say something about the model fit or adequacy. Next, present the results comparing the parameter estimates across the two groups (`experts` versus `novices`). Remember to describe ANOVAs sufficiently. The final part should contain some sort of summary with respect to the research questions. Feel free to use headings to separate the parts in the first section in a reasonable manner. Include descriptive statistics in the text, or in tables or figures as appropriate. Tables and figures should be of publication quality (i.e., fully labelled, etc.). Integrate inferential statistics into your description of the results. **Given the correctness/appropriatness of the model and statistical analysis, the first section will play the main role for your mark. If an analysis is performed in the second section, but not reported in the first, it will usually not be considered. Do not forget to consider the research question in your answer.** The first part may be up to 2000 words long (but can of course be shorter). Please note that too many figures or tables in this part can also reduce your mark.

The second section should include the complete `R` code that you used and its output. Add potentially comments (after a `#`) to explain what the code does. The code should show all of the commands that you used, enough for me to replicate exactly what you did (I will be copying and pasting code to run it, so make sure that works). You can include figures here that you used to explore the data that you do not wish to include in the first section. I will use the second section to help identify the source of any mistakes. For practical reports and papers you would only submit the first section, and thus the first section should stand alone without the second section.

Model Specification

For your diffusion model, please estimate the same five model parameters as above: `a`, `v`, `t0`, `z`, `sv`, and `st0` (i.e., fix `sz` to 0). However, note that in contrast to all analysis performed in the worksheet, we now have two stimulus classes, `blast` or `non-blast` images. Consequently, both stimulus classes need to have independent drift rates. All other parameters should be shared across both stimulus classes. In total, the simplest possible diffusion model has 7 parameters per individual data set.

Instead of the simplest model, you can also fit and report a more complicated model. This model should have separate drift rates for easy and difficult trials, separately for each stimulus class. Such a model would have 10 parameters per individual data set.

Note that in either case, due to having two drift rates, you will have to write a new function for generating start values. In addition, you will need to modify the likelihood function, or write a new function that is a wrapper for the likelihood function that makes sure that for each data point the correct drift rate is used. For example, the function could split the data into two data sets for `blast` or `non-blast` stimuli and make sure that for each of those subsets the correct drift rate is used (note again, all other parameters should be shared across the two stimulus classes so you cannot separate the data into both stimulus classes before the fitting).

Fitting the model to all participants can be time consuming. On my laptop fitting the simple model with 7 parameters to all participants takes roughly 20 minutes with 5 fitting runs each. Fitting the more complicated model with 10 parameters takes around 40 minutes. Given this, it might not make sense to re-fit the data each time you want to work on this. To avoid this, note that you can save arbitrary `R` objects via `save()` and load them with `load()`. For example, if you have saved your fits in `res_med`, the following code can be used for saving and loading, respectively:

```
save(res_med, file = "res_med1.rda")
load("res_med1.rda")
```

Additional Considerations

Please note that before fitting the data, you should exclude too fast and too slow trials. For example, exclude all trials that are faster than 250 ms and slower than 2.5 seconds. Such a cut-off is not uncommon when applying the diffusion model to data. If you use a cut-off (which probably makes sense), make sure to describe this in your text and how many trials were excluded that way (ideally using relative frequencies and not absolute values). If you use other tricks to ensure the validity and quality of your results (such as multiple fitting runs with different random starting values to avoid local optima or checks for parameter identifiability) make sure to describe this as well.

As described above, your answer should ideally also include some sentences on the adequacy of the model. How well does it describe the data? A common way to evaluate this across participants is by plotting observed and predicted response proportions in a scatter plot and report the correlation. If the model fits the data well, the correlation should be quite high (around .8 or .9) and the points near the main diagonal. One could do the same also for the observed and predicted RT quantiles (e.g., the median). As an example, take a look Trueblood et al. (2018), Figure 4 (p. 9).

If you see additional reasonable ways how to extend this analysis and you have not yet reached 2000 words for your section 1, feel free to add such additional analyses. However, make sure they address the research question at hand.

References

- Heathcote, A., Brown, S. D., & Wagenmakers, E.-J. (2015). An Introduction to Good Practices in Cognitive Modeling. In B. U. Forstmann & E.-J. Wagenmakers (Eds.), An Introduction to Model-Based Cognitive Neuroscience (pp. 25-48). Springer New York. http://www.ejwagenmakers.com/inpress/HeathcoteModelingIntro.pdf
- Trueblood, J. S., Holmes, W. R., Seegmiller, A. C., Douds, J., Compton, M., Szentirmai, E., . Eichbaum, Q. (2018). The impact of speed and bias on the cognitive processes of experts and novices in medical image decision-making. Cognitive Research: Principles and Implications, 3(1), 28. https://doi.org/10.1186/s41235-018-0119-2
- Wilson, R. C., & Collins, A. (2019). Ten simple rules for the computational modeling of behavioral data. https://doi.org/10.31234/osf.io/46mbn