bdots

Last compiled: Thursday 16th February, 2023 at 18:04

Abstract

The Bootstrapped Differences of Timeseries (bdots) was first introduced by Oleson (and others) as a method for controlling type I error in a composite of serially correlated tests of differences between two time series curves in the context of eye tracking data. This methodology was originally implemented in R by Seedorff 2018. Here, we revist the original package, both expanding the underlying theoretical components and creating a more robust implementation.

1 Introduction

i hate introductions we will do this part last

In 2017, Oleson et al. introduced a method for detecting time-specific differences in the trajectory of outcomes between experimental groups. Particularly in the case of a densely sampled time series, the construction of evaluating differences at each point in time results in a series of highly correlated test statistics expanding the family-wise error rate, accommodated with an adjustment to the nominal alpha based on this autocorrelation. This was followed up with in 2018 with the introduction of the `bdots` package to CRAN [**?**]. Here, we introduce the second version of `bdots`, an update to the package that broadly expands the capabilities of the original.

This manuscript is not intended to serve as a complete guide for using the `bdots` package. Instead, the purpose is to showcase major changes and improvements, with those seeking a more comprehensive treatment directed to the package vignettes. Rather than taking a "compare and contrast" approach, we will first enumerate the major changes, followed by a general demonstration of the package use:

1. Major changes to underlying methodology with implications for prior users of the package
2. Simplified user interface
3. Introduction of user defined curves

4. Permit fitting for arbitrary number of groups
5. Automatic detection of paired tests based on subject identifier
6. Allows for non-homogeneous sampling of data across subjects and groups
7. Introduce formula syntax for bootstrapping difference function
8. Interactive refitting process

First, a quick review of framing the problem at hand.

We start by clearly delineating the type of problem that `bdots` has been created to solve.

Bootstrapped differences in time series We begin by assuming two or more experimental groups in which a subject response is measured over time. This may include the growth of tumors in mice or the change in the proportion of fixations over time in the context of the VWP. In either case, we assume that each of the subjects in the groups being considered has observed data of the following form:

$$y_{it} = f_{\theta_i}(t) + \epsilon_{it} \tag{1}$$

OR (as was written in the original)

$$y_{it} = f(\theta_{it}) + \epsilon_{it} \tag{2}$$

where f represents a functional mean structure, while the error structure of ϵ_{it} is open to be either IID or possess an AR(1) structure. At present, `bdots` requires that each of the subjects being compared have the same parametric function f, this is not strictly necessary and future directions of the package include accommodating non-parametric functions. While each of the subjects are required to be of the same parameteric form f_θ, each differs in their instance of their own subject-specific parameters, θ_i.

The collection of subjects' θ within a group constitutes a group distribution – bootstrapping parameters from this distribution and evaluating functions f at these values gives us an estimate of the distribution of functions. As these functions are *in time*, this in turn gives a representation of temporal changes in group characteristics. It is precisely the identification of how these temporal changes differ between groups that `bdots` seeks to [do?]

This differs from the original iteration of `bdots` (or Oleson 2017) in a critical way, however. Idk how to say this, really, but the original assumed that there was no variability between subject parameters (which I am calling the homogeneity of means assumption), in which $\theta_i = \theta_j$ for all i, j in an experimental group. The primary consequence of this was the absence of any need to estimate the variability within a group, taking into consideration only subject-specific variability (don't want to go into a lot of details because I kind of

treat this mathematically in the next section). Anyways, see Chapter 3 for all the deets on this. Finally, a full review of the context in which it was introduced is available in Seedorff et. al., 2018 [**?**].

[transition paragraph]

I really need to decide how much detail to go into here. I don't want to repeat everything from chapter 3, and even though this doesn't have to be a stand alone paper now, it will be and when it is this will be pretty damn important to convey, especially to those who used bdots previously.

[also note: however we decide to deal with dissemination of this information, and while I think 2.0.0 should definitely be released with all of this, I almost htink we should address the fitcode awkwardness before this is presented to the public]

Alrightly, on to taking a looksee at how bdots works

2 Methodology and Overview

A standard analysis using `bdots` consists of two steps: fitting the observed data to a specified parametric function, f_θ, and then using the observed variability to construct estimates of the distributions of each groups' curves. Here, we briefly detail how this is implemented in practice and introduce the new methodologies in `bdots v2`. A more comprehensive treatment of these new methods, along with their justifications, is offered in Chapter 3.

2.1 Establishing subject-level curves

We begin with the assumption that for subject i, the observed data is of the form

$$y_{it} = f_{\theta_i}(t) + \epsilon_{it}, \tag{3}$$

where ϵ_{it} can be specified to be either independent or have an AR(1) auto-correlation structure. Each subject is fit in `bdots` via `gnls`, returning an estimated set of parameters and their associated standard errors. Assuming large sample normality, we are able to construct a sampling distribution for each subject, accounting for within-subject variability. This gives us for each subject a distribution

$$\hat{\theta}_i \sim N(\theta_i, s_i^2). \tag{4}$$

2.2 Estimating Group Distributions

Once sampling distributions are created for each subject, we are prepared to begin estimating group distributions. We assume that the mean parameters for each subject come from a group level distribution, where for each subject i,

$$\theta_i \sim N(\mu_\theta, V_\theta). \tag{5}$$

This is notably in contrast to the original implementation of `bdots`; there, the authors proceeded with an assumption of homogeneity of means, with $\theta_i = \theta_j$ for each i, j. A more thorough investigation of this, along with justification for the current method, is presented in Chapter 3.

This in hand, we go about estimating the group distributions according to the following procedure: (not in love with the notation here).

1. For a group of size n, select n subjects from the group *with replacement*. This allows us to construct an estimate of V_θ.
2. For each selected subject i, draw a set of parameters from the distribution $\theta_i^* \sim N(\hat{\theta}_i, s_i^2)$. This gives us an accounting of the observed within-subject variability
3. For each resampled θ_i^*, find the bth bootstrap estimate for the group,.

$$\theta_b^* = \frac{1}{n}\sum_{i=1}^{n} \theta_i^*, \quad \text{where} \quad \theta_b^* \sim N\left(\mu_\theta, \frac{1}{n}V_\theta + \frac{1}{n^2}\sum s_i^2\right). \tag{6}$$

4. Perform steps (1.)-(3.) B times, using each θ_b^* to construct a distribution of population curves, $f_\theta(t)$.

The final population curves from (4) can be used to create estimates of the mean response and an associated standard deviation at each time point for each of the groups bootstrapped. These estimates are used both for plotting and in the construction of confidence intervals. They also can be, but do not necessarily have to be, used to construct a test statistic, which is the topic of our next section.

2.3 Hypothesis testing for statistically significant differences

We now turn our attention to the primary goal of an analysis in `bdots`, identifying time windows in which the distribution of curves of two groups differ significantly. A problem unique to the ones address by `bdots` is that of multiple testing; and especially in densely sampled time series, we must account for multiple testing while controlling the family-wise error rate (FWER). Version 2 of `bdots` provides two ways with which this can be accomplished.

2.3.1 Oleson Adjustment

Just as in the original iteration of `bdots`, we are able to construct test statistics from the bootstrapped estimates described in the previous section. These test statistics $T(t)$ can be written as

$$T(t) = \frac{(\overline{f}_1(t) - \overline{f}_2(t))}{\sqrt{\frac{1}{n_1}\text{Var}(f_1(t)) + \frac{1}{n_2}\text{Var}(f_2(t))}}, \tag{7}$$

where [...] I actually don't like that notation at all, but I can't use $\overline{p}_{1t}$ and s^2_{1t}, really, because I already used s^2_i for within-subject variance. Whatever, pin in this for now, pretend I carefully described what all of the components of this test statistic is for now.

To account for the multiple testing problem with autocorrelated test statics, we construct a nominal significance level α^* to produce the desired effective FWER α. For now I won't give any more details, it basically finds some autocorrelation parameter and uses this to create a new alpha. We then use this to determine which $T(t) < z1 - \alpha^*/2$, giving us our significant regions. neato.

2.3.2 Permutation testing

Alternatively, `bdots` provides a modified permutation test for controlling the FWER without any additional assumptions on the autocorrelated status of the errors or test statistics.

We begin by creating test statistics at each time point, similar to Equation 7. Using the fitted parameters $\hat{\theta}_i$ for each subject i, we construct subject-specific curves $f_{\hat{\theta}_i}$ and use *these* to construct population mean and standard deviations at each time point, giving population curves and standard deviations [...] (i still don't have notation i really like for this, especially in contrast to the bootstrapping step). We use these to create the observed test statistic

$$T_p(t) = [\dots] \tag{8}$$

(this is basically the same formula as Equation 6 but with absolute value and having $\overline{f}$ mean different things. whatever notation is used)

When then going about using permutations to construct a null distribution against which to compare the observed statics. We do so with the following algorithm:

1. Assign to each subject a label indicating group membership
2. Randomly shuffle the labels assigned in (1.), creating two new groups
3. Recalculate the test statistic $T_p(t)$, recording the maximum value

```
> head(mouse, n = 10)
      Volume Day Treatment ID
 1:   47.432   0         A  1
 2:   98.315   5         A  1
 3:  593.028  15         A  1
 4:  565.000  19         A  1
 5: 1041.880  26         A  1
 6: 1555.200  30         A  1
 7:   36.000   0         B  2
 8:   34.222   4         B  2
 9:   45.600  10         B  2
10:   87.500  16         B  2
```

Figure 1: Illustration of Mouse data

4. Repeat (2.)-(3.) P times

The collection of maximum values for $T(t)$ will serve as the null distribution against which to compare our observed $T(t)$. Regions in which the observed t statistic are beyond the specified α in the null distribution are then considered significant.

2.3.3 Odds and Ends

Both of the methods presented are able to accommodate paired assumptions with minor adjustments to their algorithms. In the case of the bootstrap, we simply must take care to ensure that for each iteration, the collection of subjects sampled in one group with replacement are also sampled in the other, ensuring that each bootstrapped estimate comes from the same distribution. Likewise, paired testing is implemented through permutation testing by modifying the shuffling process so that each subject has one set of observations in each of the permuted groups.

Finally, it is of note that other adjustments for FWER are offered here as were in the original implementation of `bdots`, including all of the adjustments present in the `p.adjust` function from the `stats` R package. I say this for completeness, but idk that anybody cares. Okie dokie, then, thems the methods! On to an example analysis.

3 Example Analysis

In this next section we are going to review a worked example of a typical use of the `bdots` package. We will use as our illustration a study (source?) comparing tumor growth for the 451LuBr cell line in mice data with repeated measures in five treatment groups.

Note that in Figure 1, the `Day` variable contains different values between two mice (indicated by `ID`). This reflects a new feature of `bdots`, which is the ability to fit and analyze subjects with non-homogenous time samples. Details on how to adjust how non-homogenous times are handled in a later section (they're not – I'm not sure what to do abou this yet when bootstrapping functions that are FAR outside of their observed range. I almost think the default should be a mini-max approach, creating an arbitrary range of values in the intersection of the range of all of the observed data, but with ability to specify time range directly in bootstrap function).

There are two primary functions in the `bdots` package: one for fitting the observed data to a parametric function and another for estimating group distributions and identifying time windows where they differ significantly. The first of these, `bfit`, is addressed in the next section.

3.1 Curve Fitting

The curve fitting process is performed with the `bfit` function (previously `bdotsFit`), taking the following arguments:

```
bfit(data, subject, time, y, group, curveType,, ...)
```

Figure 2: Main arguments to `bfit`, though see `help(bfit)` for additional options

(need to add/talk about an AR1 argument that fits data with this assumption or not. Really this AR1 thing is a mess. I think for the dissertation, outside of new methodology we just pretend that chapter 3 doesn't exist)

The `data` argument takes the name of the dataset being used, which should be stored in long format. `subject` is the subject identifier column in the data and should be passed as a character. It is important to note here that the identification of paired data is done automatically; in determining if two experimental groups are paired, `bdots` checks that the intersection of subjects in each of the groups are identical with the subjects in each of the groups individually. The `time` and `y` arguments are column names of the time variable and outcome, respectively. Similarly, `group` takes as an argument a character vector of each of the group columns that are meant to be fit, accommodating the fact that `bdots` is now able to fit an arbitrary number of groups at once, provided that the outcomes in each group adopt the same parametric form. This brings us to the `curveType` argument, which is addressed in the next section.

Curve functions Whereas the previous iteration of `bdots` had a separate fitting function for each parametric form (i.e., `logistic.fit` for fitting data to a four-parameter logistic), we are now able to specify the curves we wish to fit independent of the fitting function `bfit`. This is done with the `curveType` argument.

Unlike the previous arguments which took either a `data.frame` or character vector, `curveType` takes as an argument a function call, for example, `logistic()`. The motivation for this is detailed elsewhere (the appendix, maybe?), but in short, it allows the user to pass additional arguments to further specify the curve. For example, among the parametric functions included in `bdots` is now the `polynomial` function, taking as an additional argument the number of degrees we wish to use. The fit the observed data with a five parameter polynomial in `bfit`, one would then pass the argument `curveType = polynomial(degree = 5)`. Literal magic takes care of the rest. Curve functions currently included in `bdots` include `logistic()`, `doubleGauss()`, `expCurve()`, and `polynomial()`. `bfit` can also accept user-created curves; detailed vignettes for writing your own can be found with `vignette("bdots")`. (maybe i'll show going from logistic to logistic with distribution for custom functions)

We fit the mouse data to an exponential curve with `expCurve()` and using the column names found in Figure 1:

```
mouse_fit <- bfit(data = mouse, subject = "ID", time = "Day",
                  y = "Volume", group = "Treatment", curveType = expCurve())
```

Return object and generics The function `bfit` returns an object of class `bdotsObj`, inheriting from class `data.table`. As such, each row uniquely identifies one permutation of subject and group values. Included in this row are the subject identifier, group classification, summary statistics regarding the curves, and a nested `gnls` object.

```
> class(mouse_fit)
[1] "bdotsObj"   "data.table" "data.frame"

> head(mouse_fit)
   ID Treatment        fit      R2   AR1 fitCode
1:  1         A <gnls[18]> 0.97349 FALSE       3
2:  2         B <gnls[18]> 0.83620 FALSE       4
3:  3         E <gnls[18]> 0.96249 FALSE       3
4:  4         C <gnls[18]> 0.96720 FALSE       3
5:  5         D <gnls[18]> 0.76156 FALSE       5
6:  7         B <gnls[18]> 0.96361 FALSE       3
```

Figure 3: A `bfit` object inheriting from `data.frame`

The number of columns will depend on the total number of groups specified, with the subject and group identifiers always being the first columns. Following this is the `fit` column, which contains the fitted object returned from `gnls`, as well as `R2` indicating the R^2 statistic. The `AR1` column indicates whether or not the observed data was able to be fit with an AR(1) error assumption. Finally, there is the `fitCode` column, which we will describe in more detail shortly.

Several methods exist for this object, including `plot`, `summary`, and `coef`, returning a matrix of fitted coefficients obtained from `gnls`.

Fit Codes The `bdots` package was originally introduced to address a very narrow scope of problems, and the `fitCode` designation is an artifact of this original intent. Specifically, it assumed that all of the observed data was of the form given in Equation 3 where the observed time series was dense and the errors were autocorrelated. Autocorrelated errors can be specified in the `gnls` package (used internally by `bdots`) when generating subject fits, though there were times when the fitter would be incapable of converging on a solution. In that instance, the autocorrelation assumption was dropped and constructing a fit was reattempted.

R^2 proved a reliable metric for this kind of data, and preference was given to fits with an autocorrelated error structure over those without. From this, the hierarchy given in Table 1 was born. `fitCode` is a numeric summary statistic ranked from 0 to 6 detailing information about the quality of the fitted curve, constructed with the following pseudo-code:

```
AR1 <- # boolean, determines AR1 status of fit
fitCode <- 3L*(!AR1) + 1L*(R2 < 0.95)*(R2 > 0.8) + 2L*(R2 < 0.8)
```

A fit code of 6 indicates that `gnls` was unable to successfully fit the subject's data.

`bdots` today stands to accommodate a far broader range of data for which the original `fitCode` standard may no longer be reliable. The presence of autocorrelation cannot always be assumed, and users may opt for a metric other than R^2 for assessing the quality of the fits. Even the assessments of fits on a discretized scale may be something of only passing interest. Even then, however, this is how the current implementation of `bdots` categorizes the quality of its fits, with the creation of greater flexibility in this regard being a large priority for future directions. Outside of general summary information, the largest impact of this system is in the refitting process, which organizes the fits by `fitCode`. There is still flexibility in how this is handled, though we will reserve that for the relevant section. (i hate this whole section).

`fitCode`	AR(1)	R^2
0	TRUE	$R^2 > 0.95$
1	TRUE	$0.8 < R2 < 0.95$
2	TRUE	$R^2 < 0.8$
3	FALSE	$R^2 > 0.95$
4	FALSE	$0.8 < R2 < 0.95$
5	FALSE	$R^2 < 0.8$
6	NA	NA

Table 1: fit codes, though less relevant for other types of data so idk really what to do about it. nothing for the dissertation, at least

Plots and summaries Users are able to quickly summarize the quality of the fits with the `summary` method now provided.

```
> summary(mouse_fit)

bdotsFit Summary

Curve Type: expCurve
Formula: Volume ~ x0 * exp(Day * k)
Time Range: (0, 106) [31 points]

Treatment: A
Num Obs:  10
Parameter Values:
        x0          k
172.232953   0.056843
########################################
############### FITS ###################
########################################
AR1,       0.95 <= R2        -- 2
AR1,       0.80 < R2 <= 0.95 -- 1
AR1,       R2 < 0.8          -- 0
Non-AR1,   0.95 <= R2        -- 0
Non-AR1,   0.8 < R2 <= 0.95  -- 3
Non-AR1,   R2 < 0.8          -- 4
No Fit                       -- 0

[...]

All Fits
Num Obs:  42
Parameter Values:
        x0          k
102.487118   0.053662
########################################
############### FITS ###################
########################################
AR1,       0.95 <= R2        -- 4
AR1,       0.80 < R2 <= 0.95 -- 2
AR1,       R2 < 0.8          -- 0
Non-AR1,   0.95 <= R2        -- 9
Non-AR1,   0.8 < R2 <= 0.95  -- 16
Non-AR1,   R2 < 0.8          -- 11
No Fit                       -- 0
```

Figure 4: Abridged output from the summary function (missing summary information for groups B-E. Note that this includes data on the formula used, the quality of fits and mean parameter estimates by group, and a summary of all fits combined

It is also recommended that users visually inspect the quality of fits for their subjects, which includes a plot of both the observed and fit data. There are a number of options available in `?plot.bdotsObj` (they can call it with just plot but you look for help with this idk), including the option to fit the plots in base R rather than `ggplot2`. This is especially helpful when looking to quickly assess the quality of its (rather than reporting) because `ggplot2` is notoriously slow, and especially when you have 400 subjects. I haven't

actually done this yet but will shortly. Anyways, check out Figure 5 for a plot of the first four fitted subjects.

```
plot(mouse_fit[1:4, ])
```

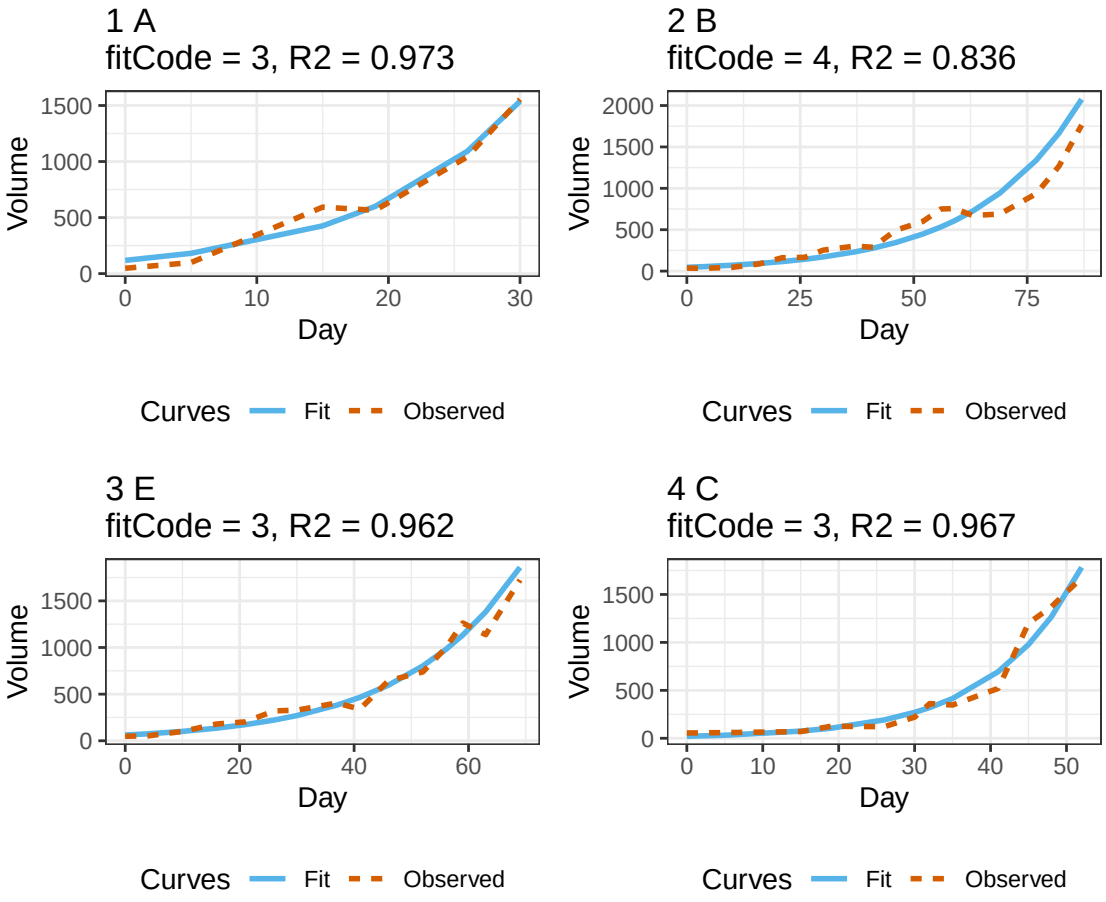

Figure 5: Plot of `mouse_fit`, using `data.table` syntax to subset to only the first four observations

3.2 Bootstrapping

Once fits have been made, we are ready to begin estimating the group distributions. This is done with the bootstrapping function, `bboot`. The number of options included in the `bboot` function have expanded to include a new formula syntax for specifying the analysis of interest as well as to include options for permutation testing. A call to `bboot` takes the following form

```
bboot(formula, bdObj, alpha, permutation = TRUE, padj = "oleson", ...)
```

Figure 6: should i caption this? It feels naked without and it feels too ritzy with

The `formula` argument is new to version 2 of `bdots` and will be discussed in the next section. As for

the remaining arguments, `bdObj` is simply the object returned from `bfit` that we wish to investigate, and `B` serves the dual role of indicating the number of bootstraps/permutations we wish to perform. `alpha` is the rate at which we wish to control the FWER. `permutation` and `padj` work in contrast to one another; when `permutation = TRUE` (the default?), the argument to `padj` is ignored. Otherwise, `padj` indicates the method to be used in adjusting the nominal `alpha` to control the FWER. By default, `padj = "oleson"`. Finally, as previously mentioned, there is no longer a need to specify if the groups are paired, and `bboot` determines this automatically based on the subject identifiers in each of the groups.

3.2.1 Formula

As the `bfit` function is now able to create fits for an arbitrary number of groups at once, we rely on a formula syntax in `bboot` to specify precisely which groups differences we wish to compare. Let`y` designate the outcome variable indicated in the `bfit` function and let `group` be one of the group column names to which our functions were fit. Further, let `val1` and `val2` be values of two of the groups in that same column. The general syntax for the `bboot` function takes the following form:

```
y ~ group(val1, val2)
```

Note that this is an *expression* in R and is written without quotation marks as in a character vector. To give a more concrete example, suppose we wished to compare the difference in tumor growth curves for A and B from the `Treatment` column in our mouse data (Figure 1). We would do so with the following syntax:

```
Volume ~ Treatment(A, B)
```

There are two special cases to consider when writing this syntax. The first is the situation that arises in the case of multiple or nested groups, the second when a difference of difference analysis is conducted. Details on both of these cases are handled in the appendix.

3.2.2 Summary and Analysis

[what gets a paragraph, what gets a subsubsection? compare this with ***subsection*** for fitting]

Let's begin first by running `bboot` using bootstrapping to compare the difference in tumor growth between treatment groups A and E in our mouse data using permutations to test for regions of significant difference.

```
mouse_boot <- bboot(Volume ~ Treatment(A, E), bdObj = mouse_fit, permutation = TRUE)
```

This returns an object of class `bdotsBootObj`. A summary method is included to display relevant information:

```
> summary(mouse_boot)

bdotsBoot Summary

Curve Type: expCurve
Formula: Volume ~ x0 * exp(Day * k)
Time Range: (0, 59) [21 points]

Difference of difference: FALSE
Paired t-test: FALSE
Difference: Treatment

FWER adjust method: Permutation
Alpha: 0.05
Significant Intervals:
     [,1] [,2]
[1,]   15   32
```

There are a few components of the summary that are worth identifying when reporting the results. In particular, note the time range provided, an indicator of if the test was paired, and which groups were being considered (noticing now it only has `Treatment`, not `A` or `E`). The last section of the summary indicates the method used, an adjusted `alphastar` if `padj` was used, and then a matrix of regions identified as being significantly different. This matrix is `NULL` is no differences were identified at the specified alpha; otherwise there is one row included for each disjointed region of significant difference.

In addition to the provided summary output, a `plot` method is available, with a list of additional options included in `help(plot.bdotsBootObj)`.

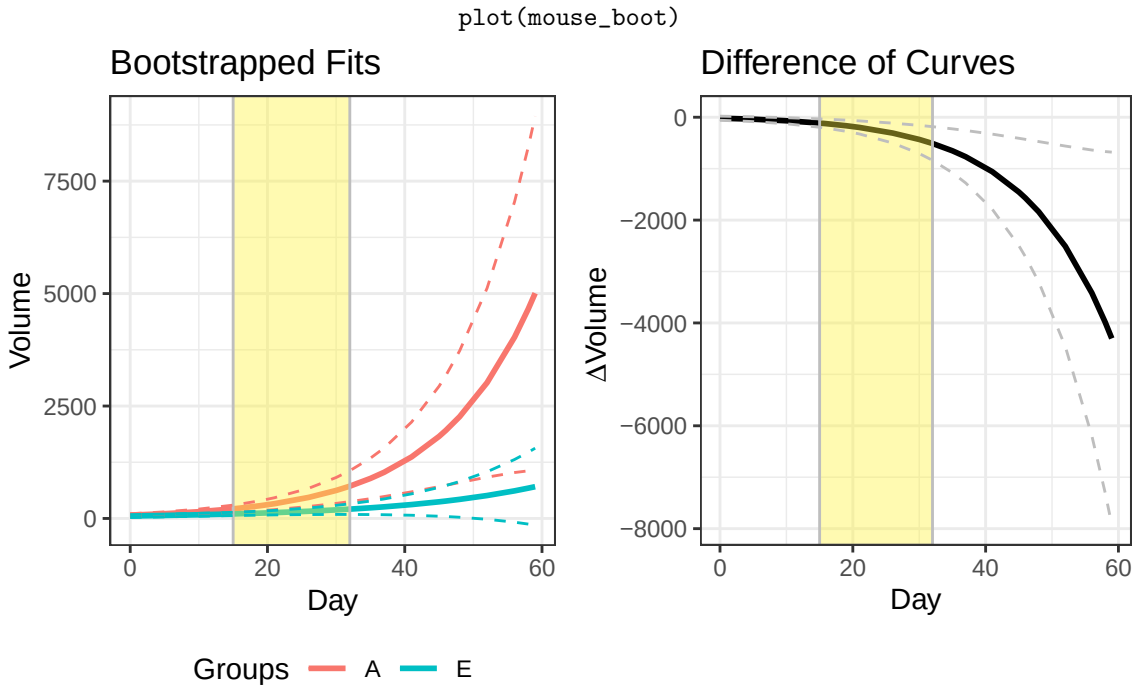

Figure 7: There are some obvious issues with time for non-homogenous samples, namely, what do we use for bootstrapping? It will be quick fix, whatever we decide, but I don't think "union of all observed times" is going to work. Here, I artificially cut it back to only 0-60

Depending on user needs, these plots can be recreated both without confidence bands or without the additional difference curve

```
plot(mouse_boot, ciBands = FALSE, plotDiffs = FALSE)
```

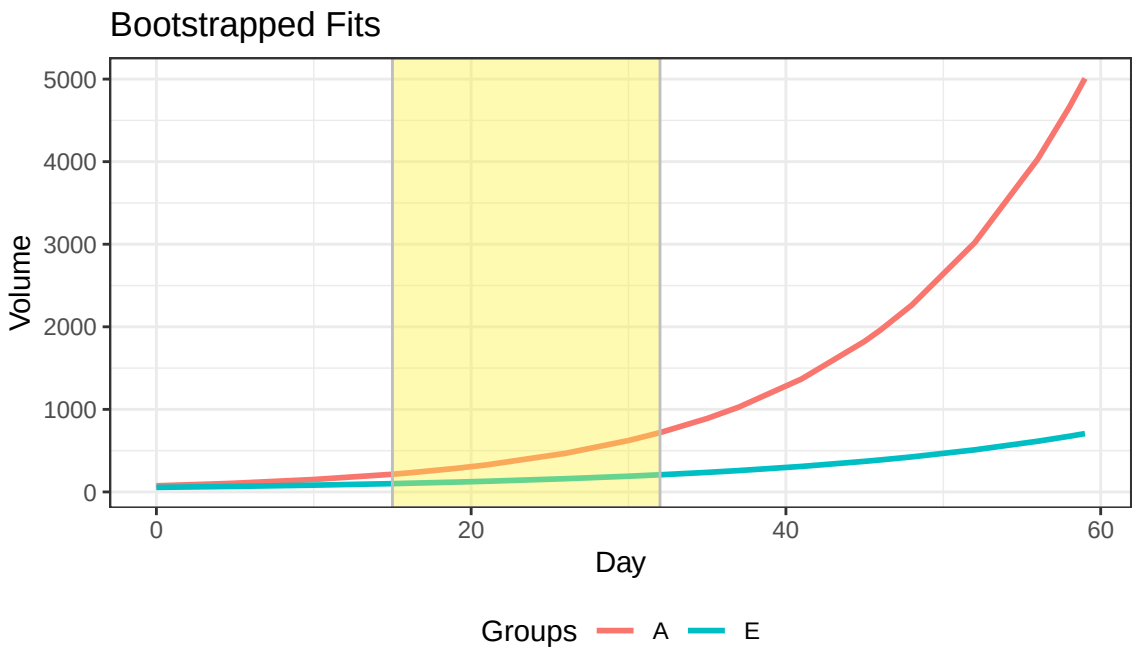

Figure 8: I think I'm going to actually not include this because who cares. I specified that they can find more options with the help function.

4 Extras

Let's do a brief tour of some of the other additions to bdots that probably doesn't warrant its own section for use

4.1 Refitting

["Better intro; I have an idea" - Patrick Breheny 1/18/31, what does better intro mean?]

There are sometimes situations in which the fitted function returned by `bfit` is a poor fit. This is largely a consequence of the sensitivity of the `nlme::gnls` function used in `bfit` for fitting the non-linear curves. Sensible starting parameters are computed as a part of the curve fitting functions (i.e., `logistic()`, but see the vignettes for more details), though these can often be improved upon. The quality of the fit can be evidenced by the `fitCode` or via a visual inspection of the fitted functions against the observations for each subject. When this occurs, there are several options available to the user, all of which are provided through the function `brefit` (previously `bdotsRefit`). `brefit` takes the following arguments:

```
brefit(bdObj, fitCode = 1L, subset = NULL, quickRefit = FALSE, paramDT = NULL)
```

The first of these arguments outside of the `bdObj` is `fitCode`, indicating the minimum fit code to be included in the refitting process. As discussed in Section 3.1, this can be sub-optimal. To add flexibility to which subjects are fit there is now the `subset` argument taking either a logical expression or collection of indices that would be used to subset an object of class `data.table` (am I explaining this clearly?) or a numeric vector with indices that the user wishes to refit.

To assist with the refitting process is the argument `quickRefit`. When set to `TRUE`, `brefit` will take the average coefficients of accepted fits within a group and use those as new starting parameters for poor fits. The new fits will be retained if they have a larger R^2 value by default (and really the only option and *technically* this isn't actually true yet but will be). When set to `FALSE`, the user will be guided through a set of prompts to refit each of the curves manually.

Finally, the `paramDT` argument allows for a `data.table` with columns for subject, group identifiers, and parameters to be passed in as a new set of starting parameters. This `data.table` requires the same format as that returned by `bdots::coefWriteout`. The use of this functionality is covered in more detail in the `bdots` vignettes and is a useful way for reproducing a `bdotsObj` from a plain text file.

When `quickRefit = FALSE`, the user is put through a series of prompts along with a series of diagnostics for each of the subjects to be refit:

```
Subject: 11
R2: 0.837
AR1: FALSE
rho: 0.9
fitCode: 4

 Model Parameters:
       x0         k
53.186497  0.051749

Actions:
1) Keep original fit
2) Jitter parameters
3) Adjust starting parameters manually
4) Remove AR1 assumption
5) See original fit metrics
6) Delete subject
99) Save and exit refitter
Choose (1-6):
```

Along with this is given a plot of the original fit, side-by-side with the suggested alternative, Figure 9.

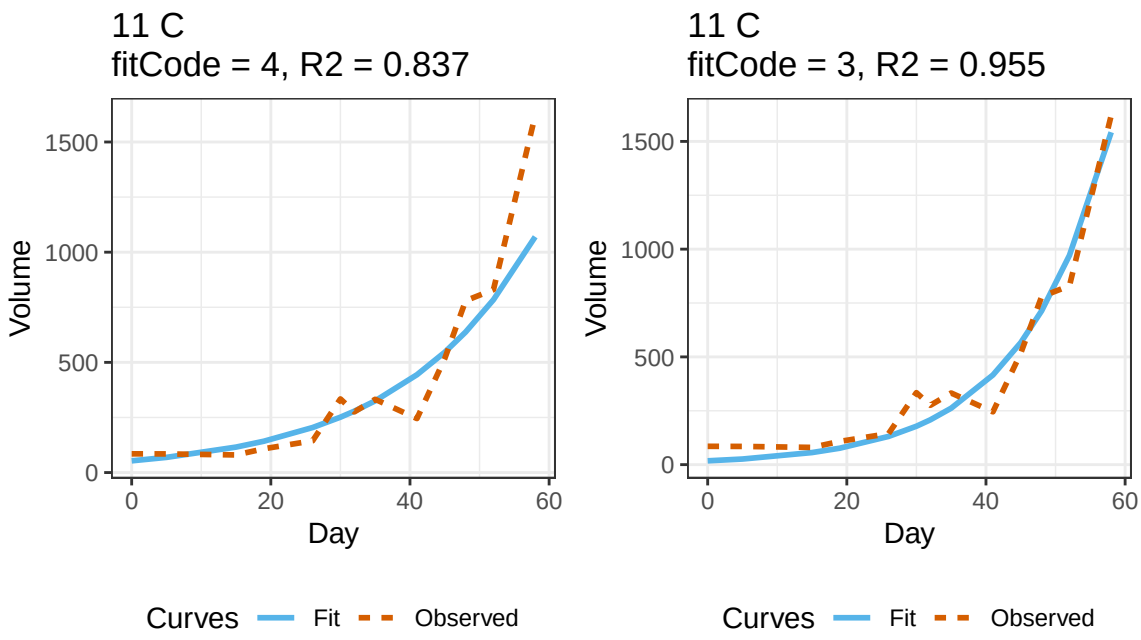

Figure 9: before and after refit: man i am good at picking new parameters

As the menu item suggests, users have the ability to end the manually refitting process early and save where they had left off. To retain previously refit items and start again at a later time, pass the first refitted object back into the refitter as such:

```
refit <- brefit(fit, ...)
refit <- brefit(refit, ...) # pass in the refitted object
```

A final note should be said regarding the option to delete a subject. As `bdots` now automatically determines if subjects are paired based on subject identifiers (necessary for calculations in significance testing), it is critical that if a subject has a poor fit in one group and must be removed that he or she is also removed from all additional groups in order to retain paired status. This can be overwritten with a final prompt in the `brefit` function before they are removed. The removal of subjects can also be done with the ancillary function, `bdRemove`, useful for removing subjects without undergoing the entire refitting process. See `help(bdRemove)` for details.

4.2 User created curves

Continue to ponder if this worth elaborating on here or appendix (or at all)

4.3 Correlations

There are sometimes cases in which we are interested in determining the correlation of a fixed attribute with group outcome responses across time . This can be done with the `bcorr` function (previously `bdotsCorr`), which takes as an argument an object of class `bdotsObj` as well as a character vector representing a column from the original dataset used in `bfit`

```
bcorr(fit, "value", ciBands, method = "pearson")
```

And I can't give an example of this with the mouse data atm because it doesn't gracefully handle nonhomogenous time samples.

This returns a thing that can be plotted. Idk, I'll see what bob wants me to say about this because truly im like so whatever

4.4 α Adjustment

Finally, we consider an extension to the `p.adjust` function, `p_adjust`, identical to `p.adjust` except that it accepts method `"oleson"` and takes additional arguments `rho`, `df`, and `cores`. `rho` determines the autocorrelation estimate for the oleson adjustment while `df` returns the degrees of freedom used to compute the original vector of t-statistics. If an estimate of `rho` isn't available, one can be computed on a vector of t-statistics using the `ar1Solver` function in `bdots`:

```
t       <- diffinv(rnorm(5))
rho     <- ar1Solver(t)
unadj_p <- pt(t, df = 10)
adj_p   <- p_adjust(unadj_p, method = "oleson",
                    df = 10, rho = rho, alpha = 0.05)
```

Doing so returns both adjusted p-values, which can be compared against the specified alpha (in this case, 0.05). Alternatively the result will also print an alphastar, a nominal alpha at which one can compare the original p-values:

```
> unadjp
[1] 0.5000000 0.0849965 0.0381715 0.1601033 0.0247453 0.0013016
> adjp
[1] 0.9201915 0.1564261 0.0702501 0.2946514 0.0455408 0.0023954
attr(,"alphastar")
[1] 0.027168
```

Here, for example, we see that the last two positions of `unadjp` have values less than `alphastar`, identifying them as significant; alternatively, we see these same two indices in `adjp` significant when compared to

```
alpha = 0.05
```

5 Discussion

[short and sweet? I hate unnecessary words]

The original implementation of `bdots` set out to address a very narrow set of problems. Previous solutions beget new opportunities, however, and it is in this space that the second iteration of `bdots` has sought to expand. Since then, the interface between user and application has been significantly revamped, creating a intuitive, reproducible workflow that is able to quickly and simply address a broader range of problems. The underlying methodology has also been improved and expanded upon, offering better control of the family-wise error rate.

While significant improvements have been made, there is room for further expansion. The most obvious of these is the need to include support for non-parametric functions, the utility of which cannot be overstated. Not only would this alleviate the need for the researcher to specify in advance a functional form for the data, it would implicitly accommodate more heterogeneity of functional forms within a group. Along with this, the current implementation is also limited in the quality-of-fit statistics used in the fitting steps to assess performance. R^2 and the presence of autocorrelation are relevant to only a subset of the types of data that can be fit, and allowing users more flexibility in specifying this metric is an active goal for future work. In all, future directions of this package will be primarily focused on user interface, non-parametric functions, and greater flexibility in fit metrics (this last sentence kind of sucks).

Appendix A - custom curves

From an R programming perspective, this is perhaps the most novel and interesting portion of the new package update. Worked use-case examples are included in the pacakge vignettes, so here we will limit discussion to the theoretical considerations when implementing it since it's actually pretty neat (I think).

Appendix X

Copy and paste hard data example here

We will illustrate use of the updated `bdots` package with a worked example, using an artificial dataset to help detail some of the newer aspects of the package. The dataset will consist of outcomes for a collection

Origin	Class	Color
foreign	car	red
		blue
	truck	red
		blue
domestic	car	red
		blue
	truck	red
		blue

Table 2: table of stuff

of vehicles, consisting of eight distinct groups. These groups will be nested in order of vehicle origin (foreign or domesetic), vehicle class (car or truck), and vehicle color (red or blue). Further, vehicles of different color but within the same origin and class groups will be considered paired observations. A table detailing the relationship of the groups is shown here:

The outcome here is simply `y` due to a lack of creativity, but the functional form assumed (and used in data generation) follows the four parameter logistic,

$$f_\theta(t) = b + \frac{p-b}{1+\exp\left(\frac{4s}{p-b}(x-t)\right)}, \tag{9}$$

where b, p, s, and x represent the baseline, peak, slope, and crossover points, respectively

The formula argument serves two functions in `bboot`: first, it specifies the collection of curves we wish to investigate the difference between, and second, it determines if we are interested in directly comparing the differences or the difference of differences between curves.

To begin, let's reintroduce the structure of the groups we have in our dataset. Recall that we have foreign and domestic cars and trucks, and each of these vehicles comes in red and blue. Recall also that the different colors of each vehicle are considered paired observations.

Beginning with a simple case, suppose we want to investigate the difference in outcome between foreign and domestic vehicles. Notionally, we would write

```
y ~ Origin(foreign, domestic).
```

Note that this involves the grouping variable, `Origin`, with the two values we are interested in comparing, `domestic` and `foreign`. With this specification, the distribution of functions considered in `domestic` include all red and blue domestic cars and trucks.

If we wanted to limit our investigation to only foreign and domestic *trucks*, we would do this by including an extra term specifying the group and the desired value. In this case,

```
y ~ Origin(foreign, domestic) + Class(truck).
```

To compare only foreign and domestic *red* trucks, we would add an additional term for color:

```
y ~ Origin(foreign, domestic) + Class(truck) + Color(red).
```

There are also instances in which we might be considered in the interaction of two groups. Although there is no native way to handle interactions in `bdots`, this can be done indirectly through the difference of differences (McMurray et al 2019, though truthfully I still don't understand why). To illustrate, suppose we are interested in understanding how the color of the vehicle differentially impacts outcome based on the vehicle class. In such a case, we might look at the difference in outcome between red cars and red trucks, and then again the difference between blue cars and blue trucks. Any difference between these two differences would give information regarding the differential impact of color between each of the two classes. This is done in `bdots` using the `diffs` synatx in the formula:

```
diffs(y, Class(car, truck)) ~ Color(red, blue)
```

Here, the *outcome* that we are considering is the difference between vehicle classes, with the outcome of interest being color. This is helpful in remembering which term goes on the LHS of the formula.

Similar as to the case before, if we wanted to limit this difference of differences investigation to only include domestic vehicles, we can do so by including an additional term:

```
diffs(y, Class(car, truck)) ~ Color(red, blue) + Origin(domestic).
```

The formula syntax was originally contrived to make comparisons within groups or within nested groups. Conceivably, however, one could be interested in making the comparison between domestic red trucks and foreign blue cars. Doing so requires a bit of a work around. Examples detailing how one might go about doing this are included in appendix B.

Appendix B - Fitting non-nested groups

(currently just copy pasted from the body of document, not edited so no need to really review)

First, there would be some function of sorts, something like `makeUniqueGroups` which would create a new group column with each permutation of previous groups being given a unique identifier. Doing this

on the vehicle example would look something like `fit <- makeuniquewhatever` resulting in the following grouping structure (for example) (and maybe you could specify group name and values who knows, kinda like factor this is just a working thought example)

Origin	Class	Color	bgroup
foreign	car	red	A
		blue	B
	truck	red	C
		blue	D
domestic	car	red	E
		blue	F
	truck	red	G
		blue	H

To then investigate differences in outcome between a foreign red car and a domestic blue truck would simply then be

$$y \sim \texttt{bgroup(A, H)}$$

yeah not like sexy or anything but whatever it would work.

A Appendix 2

```
> logistic
function (dat, y, time, params = NULL, ...)
{
    logisticPars <- function(dat, y, time, ...) {
        time <- dat[[time]]
        y <- dat[[y]]
        if (var(y) == 0) {
            return(NULL)
        }
        mini <- min(y)
        peak <- max(y)
        r <- (peak - mini)
        cross <- time[which.min(abs(0.5 * r - y))]
        q75 <- 0.75 * r + mini
        q25 <- 0.25 * r + mini
        time75 <- time[which.min(abs(q75 - y))]
        time25 <- time[which.min(abs(q25 - y))]
        tr <- time75 - time25
        tr <- ifelse(tr == 0, median(time), tr)
        slope <- (q75 - q25)/tr
        return(c(mini = mini, peak = peak, slope = slope, cross = cross))
    }
    if (is.null(params)) {
        params <- logisticPars(dat, y, time)
    }
    else {
        if (length(params) != 4)
            stop("logistic requires 4 parameters be specified for refitting")
        if (!all(names(params) %in% c("mini", "peak", "slope",
            "cross"))) {
            stop("logistic parameters for refitting must be correctly labeled")
        }
    }
    if (is.null(params)) {
        return(NULL)
    }
    y <- str2lang(y)
    time <- str2lang(time)
    ff <- bquote(.(y) ~ mini + (peak - mini)/(1 + exp(4 * slope *
        (cross - (.(time)))/(peak - mini))))
    attr(ff, "parnames") <- names(params)
    return(list(formula = ff, params = params))
}
<bytecode: 0x5591d0f862c0>
<environment: namespace:bdots>
```

```
function (dat, y, time, params = NULL, startSamp = 8, ...) {
    logisticPars <- function(dat, y, time, startSamp, ...) {
        time <- dat[[time]]
        y <- dat[[y]]
        if (var(y) == 0) {
            return(NULL)
        }
        spars <- data.table(param = c("mini", "peak", "slope", "cross"),
                    mean = c(0.115, 0.885, 0.0016, 765),
                    sd = c(0.12, 0.12, 0.00075, 85),
                    min = c(0, 0.5, 0.0009, 300),
                    max = c(0.3, 1, 0.01, 1100))
        fn <- function(p, t) {
            p[1] + (p[2] - p[1])/(1 + exp(4 * p[3] * ((p[4] - t)/(p[2] - p[1]))))
        }
        tryPars <- vector("list", length = startSamp)
        for (i in seq_len(startSamp)) {
            maxFix <- Inf
            while (maxFix > 1 | maxFix < 0.6) {
                tryPars[[i]] <- Inf
                while (any(spars[, tryPars[[i]] <= min | tryPars[[i]] >= max])) {
                  tryPars[[i]] <- spars[, rnorm(length(tryPars[[i]])) * sd + mean]
                }
                maxFix <- max(fn(tryPars[[i]], time))
            }
        }
        r2 <- vector("numeric", length = startSamp)
        for (i in seq_len(startSamp)) {
            yhat <- fn(tryPars[[i]], time)
            r2[i] <- mean((y - yhat)^2)
        }
        finalPars <- tryPars[[which.min(r2)]]
        names(finalPars) <- c("mini", "peak", "slope", "cross")
        return(finalPars)
    }
    if (is.null(params)) {
        params <- logisticPars(dat, y, time, startSamp)
    }
    # Was var(y) == 0?
    if (is.null(params)) {
        return(NULL)
    }
    y <- str2lang(y)
    time <- str2lang(time)
    ff <- bquote(.(y) ~ mini + (peak - mini)/(1 + exp(4 * slope *
        (cross - (.(time)))/(peak - mini))))
    attr(ff, "parnames") <- names(params)
    return(list(formula = ff, params = params))
}
<bytecode: 0x5591d39da1f0>
<environment: namespace:bdots>
```