Reproducibly Analysing Visual-World Eyetracking Data

Dale Barr

10/26/2022

Table of contents

Background

For this workshop, we will be reproducing data from a study by Weighall et al. (2017) on the role of sleep in learning novel words (lexical consolidation).

Unlike in the Weighall et al. article, to simplify presentation, this book will focus on the comparison between adult and child participants in the processing of existing "lexical competitors", as in the words "candy" and "candle".

Raw data

You will need to download and extract data-raw.zip into your working directory.

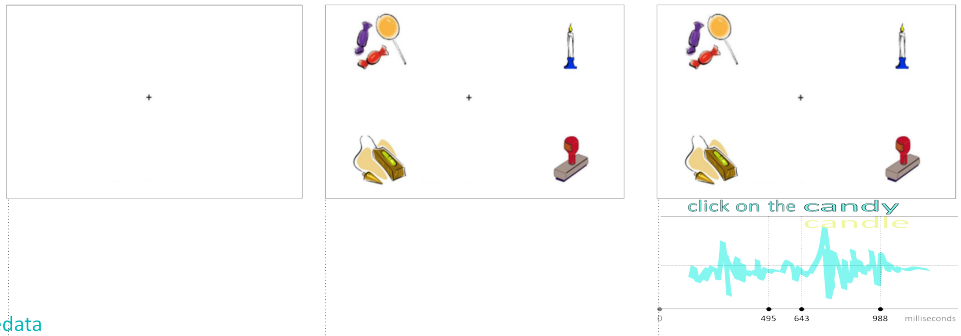
Appendix A has a figure showing the structure of the data. You will probably need to refer back to this figure many times as you work through the exercises.

Part I

Pre-processing

1 Import, epoching, and time-alignment

The overall task here is to scrape out the data we want to use from each trial (epoching) and align the frame counters for all trials to the disambiguation point for the particular audio stimulus that was played on that trial (time-alignment). In other words, the disambiguation point should be the temporal "origin" (zero point) for the timeline on each trial.



adult/sub_003.gazedata

UserDefined_1	Fixation	...	Fixation	...	Preview	...	Preview	...	StimSlide	...	StimSlide	...
ID	1	...	61		145	...	177		210		269	
TimestampSec	1317141124	...			1317141127	...			1317141128	...		
TimestampMicrosec	669529	...			77206	...			159568	...		
CursorX	-1	...			666	...			651	...		
CursorY	-1	...			521	...			526	...		
TrialId	1	...			1	...			1	...		
f_c	-268		-208		-124		-92		-59		0	

1.1 Data import

For the first part of pre-processing, we will load the eye data into our R session using functions from the **{readr}** package, which is one of many packages that is part of the **{tidyverse}** meta-package. The `.gazedata` files from the Tobii eyetracking system are in `.tsv` or **T**ab **S**eparated **V**alues format, for which we use `read_tsv()`.

Before we can perform epoching and time-alignment, we have to import and clean up the `.gazedata` files. These are 42 adult data files and 41 child data files located in the `adult` and `child` subdirectories of `data-raw/`. These files follow the naming convention `data-raw/adult/sub_XXX.gazedata` and `data-raw/child/sub_XXX.gazedata` where the

`XXX` part of the filename the unique integer identifying each subject, which corresponds to `sub_id` in the `subjects` table.

The raw gazedata files include a lot of unnecessary information. We'll need to scrape out the data that we need and convert the `XXX` value from the filename into a `sub_id` variable in the resulting table. The source files have the format below.

variable	type	description
ID	int	arbitrary value uniquely identifying each frame within subject
TETTime	dbl	(ignored)
RTTime	int	(ignored)
CursorX	int	horizontal point of gaze in pixels
CursorY	int	vertical point of gaze in pixels
TimestampSec	int	timestamp in seconds
TimestampMicrosec	int	millisecond portion of timestamp (cycles around)
XGazePosLeftEye	dbl	(ignored)
YGazePosLeftEye	dbl	(ignored)
XCameraPosLeftEye	dbl	(ignored)
YCameraPosLeftEye	dbl	(ignored)
DiameterPupilLeftEye	dbl	(ignored)
DistanceLeftEye	dbl	(ignored)
ValidityLeftEye	int	(ignored)
XGazePosRightEye	dbl	(ignored)
YGazePosRightEye	dbl	(ignored)
XCameraPosRightEye	dbl	(ignored)
YCameraPosRightEye	dbl	(ignored)
DiameterPupilRightEye	dbl	(ignored)
DistanceRightEye	dbl	(ignored)
ValidityRightEye	int	(ignored)
TrialId	int	arbitrary value uniquely identifying each trial within a subject (same as t_id)
UserDefined_1	chr	phase of the trial (Fixation, Preview, StimSlide)

1.1.1 Activity: One Subject

Read in the Tobii eyetracking data for a single subject from the datafile `data-raw/adult/sub_003.gazedata`, and convert it to the format below.

```
# A tibble: 16,658 x 7
   sub_id  t_id  f_id         sec     x     y phase
    <int> <int> <int>       <dbl> <int> <int> <chr>
```

```
 1      3     1   145 1317141127.   666   521 Preview
 2      3     1   146 1317141127.   649   442 Preview
 3      3     1   147 1317141127.   618   507 Preview
 4      3     1   148 1317141127.   645   471 Preview
 5      3     1   149 1317141127.   632   471 Preview
 6      3     1   150 1317141127.   645   536 Preview
 7      3     1   151 1317141127.   651   474 Preview
 8      3     1   152 1317141127.   643   541 Preview
 9      3     1   153 1317141127.   628   581 Preview
10      3     1   154 1317141127.   643   532 Preview
# ... with 16,648 more rows
```

Here, we have renamed `TrialId` to `t_id`, which is the name it takes throughout the rest of the database. We have also renamed `CursorX` and `CursorY` to `x` and `y` respectively. We have also renamed `ID` to `f_id` (frame id) and `UserDefined_1` to `phase`. We also exclude any frames from the phase where `UserDefined_1 == "Fixation"`, because these frames are not informative, and doing so reduces the size of the data we need to import.

Hint: Importing only those columns you need

Use the `col_types` argument to `read_tsv()` and the `cols_only()` specification.
For instance, something like:

```
read_tsv("data-raw/adult/sub_003.gazedata",
         col_types = cols_only(ID = col_integer(),
                               # [..etc]
                               ),
         #.. other args to read_tsv,
         )
```

Type `?readr::cols_only` in the console to learn more about specifying columns during data import.

Hint: Extracting the subject id number

You can use the `id` argument to `read_tsv()` to specify the name of a variable in the resulting data frame that has the filename as its value.
You can then create a new variable using `mutate()` that extracts the `XXX` substring (positions 20-22 of the string) and then converts it to an integer.

```
read_tsv("data-raw/adult/sub_003.gazedata",
         id = "filename",
         # other args to read_tsv()...
         ) %>%
  mutate(sub_id = substr(filename, 20, 22) %>% as.integer()) # %>%
  ## rest of your pipeline..
```

! Solution

```
library("tidyverse")

## make sure that your working directory is properly set!

read_tsv("data-raw/adult/sub_003.gazedata",
         col_types = cols_only(ID = col_integer(),
                               TrialId = col_integer(),
                               CursorX = col_integer(),
                               CursorY = col_integer(),
                               TimestampSec = col_integer(),
                               TimestampMicrosec = col_integer(),
                               UserDefined_1 = col_character()),
         id = "filename") %>%
  ## convert XXX to sub_id
  mutate(sub_id = substr(filename, 20, 22) %>% as.integer(),
         sec = TimestampSec + TimestampMicrosec / 1000000) %>%
  select(sub_id, t_id = TrialId, f_id = ID,
         sec, x = CursorX, y = CursorY,
         phase = UserDefined_1) %>%
  filter(phase != "Fixation")
```

1.1.2 Activity: All Subjects

Now adapt the code that you wrote above to load in *all* 83 into a single table, which should have the same format as for the data you imported for subject 3 above.

Tip

The `readr` functions like `read_tsv()` make it easy to read in multiple files. All you need to do is to provide a vector of filenames as the first argument.
For example, `read_tsv(c("file1.tsv", "file2.tsv"))` will read both `file1.tsv` and `file2.tsv` and bind together the rows imported from both files in the result.

Hint: How do I get a vector of all the files in a directory?

The `dir()` function for base R can be used to list files. Examples:

```
dir("data-raw")
```

```
[1] "adult"              "child"              "locations.csv"
[4] "screens.csv"        "speech-timings.csv" "stimuli.csv"
[7] "subjects.csv"       "trials.csv"
```

```
adults <- dir("data-raw/adult", full.names = TRUE)

adults
```

```
 [1] "data-raw/adult/sub_001.gazedata" "data-raw/adult/sub_002.gazedata"
 [3] "data-raw/adult/sub_003.gazedata" "data-raw/adult/sub_004.gazedata"
 [5] "data-raw/adult/sub_005.gazedata" "data-raw/adult/sub_006.gazedata"
 [7] "data-raw/adult/sub_007.gazedata" "data-raw/adult/sub_008.gazedata"
 [9] "data-raw/adult/sub_009.gazedata" "data-raw/adult/sub_010.gazedata"
[11] "data-raw/adult/sub_011.gazedata" "data-raw/adult/sub_012.gazedata"
[13] "data-raw/adult/sub_013.gazedata" "data-raw/adult/sub_014.gazedata"
[15] "data-raw/adult/sub_015.gazedata" "data-raw/adult/sub_016.gazedata"
[17] "data-raw/adult/sub_017.gazedata" "data-raw/adult/sub_018.gazedata"
[19] "data-raw/adult/sub_019.gazedata" "data-raw/adult/sub_020.gazedata"
[21] "data-raw/adult/sub_021.gazedata" "data-raw/adult/sub_022.gazedata"
[23] "data-raw/adult/sub_023.gazedata" "data-raw/adult/sub_024.gazedata"
[25] "data-raw/adult/sub_025.gazedata" "data-raw/adult/sub_026.gazedata"
[27] "data-raw/adult/sub_027.gazedata" "data-raw/adult/sub_028.gazedata"
[29] "data-raw/adult/sub_029.gazedata" "data-raw/adult/sub_030.gazedata"
[31] "data-raw/adult/sub_031.gazedata" "data-raw/adult/sub_032.gazedata"
[33] "data-raw/adult/sub_033.gazedata" "data-raw/adult/sub_034.gazedata"
[35] "data-raw/adult/sub_035.gazedata" "data-raw/adult/sub_036.gazedata"
[37] "data-raw/adult/sub_037.gazedata" "data-raw/adult/sub_039.gazedata"
[39] "data-raw/adult/sub_040.gazedata" "data-raw/adult/sub_041.gazedata"
[41] "data-raw/adult/sub_042.gazedata" "data-raw/adult/sub_043.gazedata"
```

❗ Solution

```
## get .gazedata filenames
adults <- dir("data-raw/adult", full.names = TRUE)
children <- dir("data-raw/child", full.names = TRUE)

edat <- read_tsv(c(adults, children),
                 col_types = cols_only(ID = col_integer(),
                                       TrialId = col_integer(),
                                       CursorX = col_integer(),
                                       CursorY = col_integer(),
                                       TimestampSec = col_integer(),
                                       TimestampMicrosec = col_integer(),
                                       UserDefined_1 = col_character()),
                 id = "filename") %>%
  mutate(sub_id = substr(filename, 20, 22) %>% as.integer(),
         sec = TimestampSec + TimestampMicrosec / 1000000) %>%
  select(sub_id, t_id = TrialId, f_id = ID,
         sec, x = CursorX, y = CursorY,
         phase = UserDefined_1) %>%
  filter(phase != "Fixation")

edat
```

```
# A tibble: 1,899,013 x 7
   sub_id  t_id  f_id         sec     x     y phase
    <int> <int> <int>       <dbl> <int> <int> <chr>
 1      1     1   272 1317113393.   628   523 Preview
 2      1     1   273 1317113393.   634   529 Preview
 3      1     1   274 1317113393.   633   519 Preview
 4      1     1   275 1317113393.   644   531 Preview
 5      1     1   276 1317113393.   637   520 Preview
 6      1     1   277 1317113393.   635   515 Preview
 7      1     1   278 1317113393.   636   519 Preview
 8      1     1   279 1317113393.   638   518 Preview
 9      1     1   280 1317113393.   642   519 Preview
10      1     1   281 1317113393.   638   518 Preview
# ... with 1,899,003 more rows
```

1.2 Epoching and time-alignment

The Tobii eyetracker recorded data at a rate of 60 Hertz (i.e., 60 frames per second, or one frame every 1/60th of a second.) For each trial, the frame counter (`ID`, which we renamed to `f_id`) starts at 1 and increments every frame. This is not very useful because we need to know when certain stimulus events occurred, and these will take place at a different frame number for every trial, depending on the timing of the speech events of the stimulus for that trial. We need to re-define the 'origin' of the eye-tracking data. In this study, we used the 'disambiguation point', which is the point in the word where the signal distinguishes between two competing lexical items (e.g., candy and candle).

adult/sub_003.gazedata

UserDefined_1	Fixation	...	Fixation	...	Preview	...	Preview	...	StimSlide	...	StimSlide	...
ID	1	...	61		145	...	177		210		269	
TimestampSec	1317141124	...			1317141127	...			1317141128	...		
TimestampMicrosec	669529	...			77206	...			159568	...		
CursorX	-1	...			666	...			651	...		
CursorY	-1	...			521	...			526	...		
TrialId	1	...			1	...			1	...		
f_c	-268		-208		-124		-92		-59		0	

As the above figure shows, each trial had three phases, a `Fixation`, `Preview`, and `StimSlide` phase, which are indexed by the variable `phase`. Playback of a soundfile with a pre-recorded speech stimulus began simultaneously with the onset of the `StimSlide` phase.

For each trial (uniquely identified by `sub_id` and `t_id`), we are going to need to do two things to time-align the eye data to the disambiguation point.

1. Find out what sound was played and the timing of the disambiguation point within that soundfile, as measured from the start of the file.
2. Figure out the frame number corresponding to the start of the `StimSlide` phase and then adjust by the amount calculated in the previous step.

1.2.1 Activity: Disambiguation Point

Create the table below from the raw data, which has information about the onset of the disambiguation point for each trial. Store the table as `origin_adj`.

You may wish to consult Appendix A to see what tables the values in the table below have been are drawn from. You'll need to import these tables into your session. All of these tables have the extension `.csv`, which indicates they are in **C**omma **S**eparated **V**alues format. The ideal way to import these files is to use `read_csv()` from the **{readr}** package.

```
# A tibble: 5,644 x 4
   sub_id  t_id sound          disambig_point
    <int> <int> <chr>                   <int>
 1      1     1 Tpelican.wav             1171
 2      1     2 Tpumpkin.wav             1079
 3      1     3 pencil.wav                810
 4      1     4 paddle.wav                881
 5      1     6 Tbalcony.wav             1012
 6      1     7 Tnapkin.wav              1069
 7      1    11 Tflamingo.wav            1150
 8      1    13 Tangel.wav               1036
 9      1    14 Tparachute.wav           1046
10      1    16 Tmushroom.wav            1062
# ... with 5,634 more rows
```

! Solution

```
trials <- read_csv("data-raw/trials.csv",
                   col_types = "iiiiii")

stimuli <- read_csv("data-raw/stimuli.csv",
                    col_types = "iiciccc")

speech <- read_csv("data-raw/speech-timings.csv",
                   col_types = "ciii")

origin_adj <- trials %>%
  inner_join(stimuli, "iv_id") %>% # to get `sound`
  select(sub_id, t_id, sound) %>%
  inner_join(speech, "sound") %>% # to get the timings
  select(-article, -noun)
```

1.2.2 Activity: Onset of StimSlide

Now let's do part 2, where we find the value of `f_id` for the first frame of eyedata for each trial following the onset of the `StimSlide` phase. We should have a table that looks like the one below, with one row for each trial, and where `f_ss` is the value of `f_id` for the earliest frame in the `StimSlide` phase.

```
# A tibble: 7,385 x 3
   sub_id  t_id  f_ss
    <int> <int> <int>
 1      1     1   338
 2      1     2   729
 3      1     3  1124
 4      1     4  1443
 5      1     5  1795
 6      1     6  2300
 7      1     7  2593
 8      1     8  3348
 9      1     9  3874
10      1    10  4331
# ... with 7,375 more rows
```

! Solution

```
## figure out the f_id for the earliest StimSlide frame
origin_frames <- edat %>%
  filter(phase == "StimSlide") %>%
  group_by(sub_id, t_id) %>%
  summarise(f_ss = min(f_id),
            .groups = "drop")

origin_frames
```

1.2.3 Activity: Combine origins

Now that we have the first frame of `StimSlide` and the adjustment we have to make in milliseconds for the disambiguation point, combine the tables and calculate `f_z`, which will represent the "zero points" in frames for each trial. Store the resulting table in `origins`.

```
# A tibble: 5,643 x 5
   sub_id  t_id  f_ss disambig_point   f_z
    <int> <int> <int>          <int> <int>
 1      1     1   338           1171   408
 2      1     2   729           1079   794
 3      1     3  1124            810  1173
 4      1     4  1443            881  1496
 5      1     6  2300           1012  2361
 6      1     7  2593           1069  2657
 7      1    11  4699           1150  4768
 8      1    13  5395           1036  5457
 9      1    14  5893           1046  5956
10      1    16  6811           1062  6875
# ... with 5,633 more rows
```

Hint: How to convert milliseconds to frames of eye data

There are 60 frames per second, so 60 frames per 1000 milliseconds.
So to convert from milliseconds to frames:
`f_z = 60 * ms / 1000`
For example, if you have 500 ms, then 60 * 500 / 1000 = 30.

Solution

```
origins <- origin_frames %>%
  inner_join(origin_adj, c("sub_id", "t_id")) %>%
  mutate(f_z = round(f_ss + 60 * disambig_point / 1000) %>%
           as.integer()) %>%
  select(-sound)
```

1.2.4 Activity: Time-align

Now we're ready to calculate a new frame index on our eye data (`edat`), `f_c`, which is centered on the zero point, `f_z`. The resulting table should be called `epdat` and have the following structure.

```
# A tibble: 1,341,405 x 7
   sub_id  t_id  f_id   f_z   f_c     x     y
    <int> <int> <int> <int> <int> <int> <int>
```

```
 1      1     1   272   408  -136   628   523
 2      1     1   273   408  -135   634   529
 3      1     1   274   408  -134   633   519
 4      1     1   275   408  -133   644   531
 5      1     1   276   408  -132   637   520
 6      1     1   277   408  -131   635   515
 7      1     1   278   408  -130   636   519
 8      1     1   279   408  -129   638   518
 9      1     1   280   408  -128   642   519
10      1     1   281   408  -127   638   518
# ... with 1,341,395 more rows
```

! Solution

```
epdat <- edat %>%
  inner_join(origins, c("sub_id", "t_id")) %>%
  mutate(f_c = f_id - f_z) %>%
  select(sub_id, t_id, f_id, f_z, f_c, x, y)
```

1.3 Save the data

We've reached a stopping point. We'll want to save the epoched data so that we can use that as our starting point for the next preprocessing stage. We'll remove the variables `f_id` and `f_z` because we no longer need them. We'll also keep 1.5 seconds (90 frames) before and after the disambiguation point for each trial.

```
## if we haven't made a "data-derived" directory, do so now
if (!dir.exists("data-derived")) dir.create("data-derived")

epdat %>%
  filter(f_c >= -90L, f_c <= 90L) %>%
  select(-f_id, -f_z) %>%
  saveRDS(file = "data-derived/edat-epoched.rds")
```

2 Mapping gaze to areas of interest

At this point we have epoched our eyetracking data, resulting in the `edat-epoched.rds` file which looks like so:

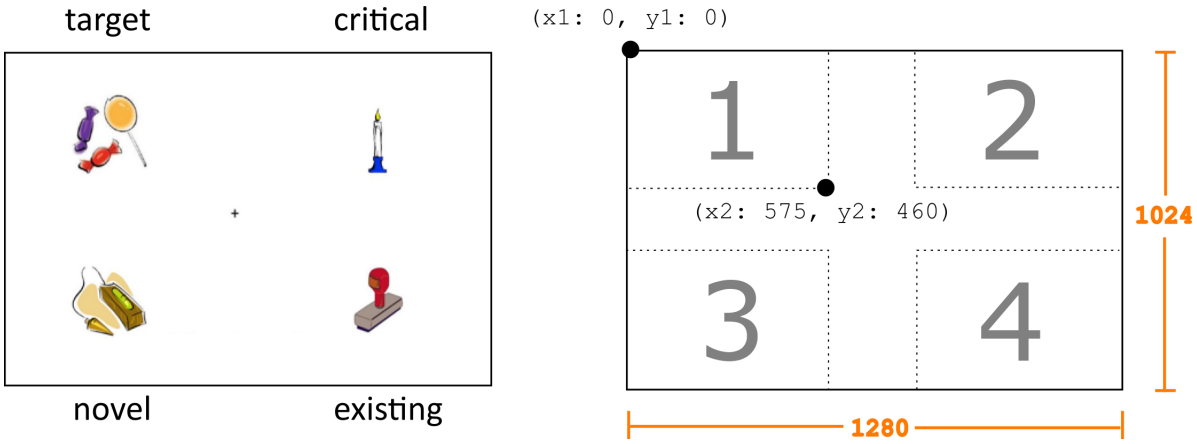
```
library("tidyverse")

edat <- read_rds("data-derived/edat-epoched.rds")

edat
```

```
# A tibble: 941,815 x 5
   sub_id  t_id   f_c     x     y
    <int> <int> <int> <int> <int>
 1      1     1   -90  1010   209
 2      1     1   -89  1010   214
 3      1     1   -88  1020   216
 4      1     1   -87  1014   217
 5      1     1   -86  1027   221
 6      1     1   -85  1027   220
 7      1     1   -84  1018   218
 8      1     1   -83  1019   216
 9      1     1   -82  1024   219
10      1     1   -81  1019   216
# ... with 941,805 more rows
```

We know **when** people are looking relative to the disambiguation point for the trial (`f_c`), and we know **where** they are looking, because we have the (x, y) coordinates. But we yet don't know **which image they are looking at** on each frame. So we have to map the two-dimensional gaze coordinates onto the coordinates of the images that was displayed on a given trial.



We know what pictures were shown on each trial from the data in the `screens` table (from `data-raw/screens.csv`).

```
screens <- read_csv("data-raw/screens.csv",
                    col_types = "iicc")
```

The table looks like so.

```
# A tibble: 1,024 x 4
    s_id   loc role     bitmap
   <int> <int> <chr>    <chr>
 1     1     3 critical bacon.bmp
 2     1     4 existing EDsandcastle.bmp
 3     1     2 novel    ND_104.bmp
 4     1     1 target   baker.bmp
 5     2     3 critical penny.bmp
 6     2     4 existing EDsandcastle.bmp
 7     2     2 novel    ND_104.bmp
 8     2     1 target   baker.bmp
 9     3     2 critical beetle.bmp
10     3     4 existing EDcaptain.bmp
# ... with 1,014 more rows
```

variable	type	description
s_id	int	arbitrary value uniquely identifying each display screen
loc	int	arbitrary integer identifying each rectangle
role	chr	image's role in the set (target, critical, existing novel)
bitmap	chr	name of bitmap file

The `loc` variable is a number that refers to the four quadrants of the screen where the images appeared. We can get the pixel coordinates representing the top left and bottom right corners of each rectangle from the `locations` table.

```
locations <- read_csv("data-raw/locations.csv",
                      col_types = "iiiii")
```

```
# A tibble: 4 x 5
    loc    x1    y1    x2    y2
  <int> <int> <int> <int> <int>
1     1     0     0   575   460
2     2   704     0  1279   460
3     3     0   564   575  1023
4     4   704   564  1279  1023
```

variable	type	description
loc	int	arbitrary integer identifying each rectangle
x1	int	horizontal coordinate of top-left corner in pixels
y1	int	vertical coordinate of top-left corner in pixels
x2	int	horizontal coordinate of bottom-right corner in pixels
y2	int	vertical coordinate of bottom-right corner in pixels

2.1 Image locations for each trial

2.1.1 Activity: Get coordinates

We want to combine the data from `screens` and `locations` with trial info to create the following table, which we will use later to figure out what image was being looked at (if any) on each frame of each trial. Save this information in a table named `aoi` (for **A**rea **O**f **I**nterest). You might need to reference Appendix A to see how to get `sub_id` and `t_id` into the table.

```
# A tibble: 22,576 x 8
   sub_id  t_id  s_id role        x1    y1    x2    y2
    <int> <int> <int> <chr>    <int> <int> <int> <int>
 1      1     1   183 critical   704   564  1279  1023
 2      1     1   183 existing     0   564   575  1023
 3      1     1   183 novel        0     0   575   460
 4      1     1   183 target     704     0  1279   460
 5      1     2   194 critical     0   564   575  1023
```

```
 6      1     2   194 existing   704     0  1279   460
 7      1     2   194 novel        0     0   575   460
 8      1     2   194 target     704   564  1279  1023
 9      1     3    33 critical   704     0  1279   460
10      1     3    33 existing     0   564   575  1023
# ... with 22,566 more rows
```

! Solution

We can get `sub_id` and `t_id` from `trials`. But to get there from `screens`, we need to get the item version (`iv_id`) from `stimuli`. We can connect `screens` to `stimuli` through the screen id (`s_id`).

```
trials <- read_csv("data-raw/trials.csv",
                   col_types = "iiiiii")

stimuli <- read_csv("data-raw/stimuli.csv",
                    col_types = "iiciccc")

aoi <- trials %>%
  select(sub_id, t_id, iv_id) %>%
  inner_join(stimuli, "iv_id") %>%
  inner_join(screens, "s_id") %>%
  inner_join(locations, "loc") %>%
  select(sub_id, t_id, s_id, role, x1, y1, x2, y2)
```

As a check, we should have four times the number of rows as `trials` (5644), because there should be four areas of interest for each trial. We can use `stopifnot()` to make our script terminate if this condition is not satisfied.

```
stopifnot( nrow(aoi) == 4 * nrow(trials) )
```

2.2 Identifying frames where the gaze cursor is within an AOI

What we need to do now is look at the (`x`, `y`) coordinates in `edat` and see if they fall within the bounding box for each image in the `aoi` table for the corresponding trial.

2.2.1 Activity: Create `frames_in`

There are different ways to accomplish this task, but an effective strategy is just to join the eyedata (`edat`) to the `aoi` table and retain any frames where the `x` coordinate of the eye gaze is within the `x1` and `x2` coordinates of the rectangle, and the `y` coordinate is within the `y1` and `y2` coordinates. Because our AOIs do not overlap, the gaze can only be within a single AOI at a time.

Name the resulting table `frames_in`.

> Hint
>
> Some code to get you started.
>
> ```
> edat %>%
>   inner_join(aoi, c("sub_id", "t_id")) # %>%
>   ## filter(...)
> ```

```
# A tibble: 759,311 x 4
   sub_id  t_id   f_c role
    <int> <int> <int> <chr>
 1      1     1   -90 target
 2      1     1   -89 target
 3      1     1   -88 target
 4      1     1   -87 target
 5      1     1   -86 target
 6      1     1   -85 target
 7      1     1   -84 target
 8      1     1   -83 target
 9      1     1   -82 target
10      1     1   -81 target
# ... with 759,301 more rows
```

> Solution
>
> ```
> frames_in <- edat %>%
>   inner_join(aoi, c("sub_id", "t_id")) %>%
>   filter(x >= x1, x <= x2,
>          y >= y1, y <= y2) %>%
>   select(sub_id, t_id, f_c, role)
> ```

2.2.2 Activity: Create `frames_out`

Create a table `frames_out` containing only those frames from `edat` where the gaze fell outside of any of the four image regions, and label those with the role `(blank)`. Use the `anti_join()` function from `dplyr` to do so.

The resulting table should have the format below.

```
# A tibble: 182,504 x 4
   sub_id  t_id   f_c role
    <int> <int> <int> <chr>
 1      1     1   -69 (blank)
 2      1     1   -68 (blank)
 3      1     1   -66 (blank)
 4      1     1   -49 (blank)
 5      1     1    -9 (blank)
 6      1     1    -8 (blank)
 7      1     1    -7 (blank)
 8      1     1    -6 (blank)
 9      1     1    -5 (blank)
10      1     1    -4 (blank)
# ... with 182,494 more rows
```

Hint: Show me an example of `anti_join()`

```
table_x <- tibble(letter = c("A", "B", "C", "D", "E"),
                  number = c(1, 2, 3, 4, 5))

table_x
```

```
# A tibble: 5 x 2
  letter number
  <chr>   <dbl>
1 A           1
2 B           2
3 C           3
4 D           4
5 E           5
```

```
table_y <- tibble(letter = c("C", "D", "E"),
                  number = c(3, 4, 99))

table_y
```

```
# A tibble: 3 x 2
  letter number
  <chr>   <dbl>
1 C           3
2 D           4
3 E          99
```

```
## which rows in table_x are not in table_y?
anti_join(table_x, table_y, c("letter", "number"))
```

```
# A tibble: 3 x 2
  letter number
  <chr>   <dbl>
1 A           1
2 B           2
3 E           5
```

! Solution

```
frames_out <- edat %>%
  select(sub_id, t_id, f_c) %>%
  anti_join(frames_in, c("sub_id", "t_id", "f_c")) %>%
  mutate(role = "(blank)")
```

A good test to do at this point is to make sure that all 941,815 rows of `edat` have been assigned to either `frames_in` or `frames_out`.

```
stopifnot( nrow(edat) == (nrow(frames_in) + nrow(frames_out)) ) # TRUE
```

2.2.3 Activity: Combine into `pog`

Combine `frames_in` and `frames_out` into a single table by concatenating the rows. Sort the rows so by `sub_id`, `t_id`, and `f_c`, and convert `role` into type `factor` with levels in this order:

`target`, `critical`, `existing`, `novel`, and `(blank)`. The resulting table should be called `pog` and have the format below.

```
# A tibble: 941,815 x 4
   sub_id  t_id   f_c role
    <int> <int> <int> <fct>
 1      1     1   -90 target
 2      1     1   -89 target
 3      1     1   -88 target
 4      1     1   -87 target
 5      1     1   -86 target
 6      1     1   -85 target
 7      1     1   -84 target
 8      1     1   -83 target
 9      1     1   -82 target
10      1     1   -81 target
# ... with 941,805 more rows
```

How do I concatenate two tables?

Use the `bind_rows()` function from **{dplyr}**.

Solution

We might want to check that `role` has been defined properly.

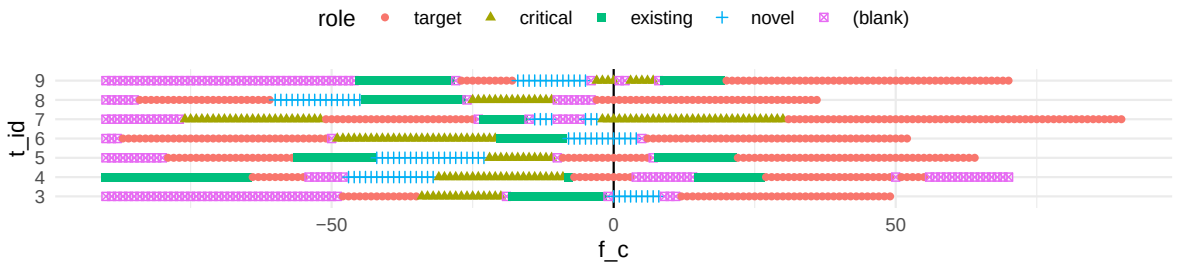
```
pog %>%
  pull(role) %>%
  levels()
```

```
[1] "target"   "critical" "existing" "novel"    "(blank)"
```

2.3 Dealing with trial dropouts

We want to be able to use the data in `pog` to calculate probabilities of gazing at regions over time. However, we are not ready to do this yet.

If we look at the first seven trials from subject 3, we can see that there is a problem, because the trials end at different times, due to variation in response time. If we plot the resulting data, we will have fewer and fewer data points as we progress through the trial.



A solution to this is to make each time series "cumulative to selection", which means padding frames after the trial ends with artificial looks to the object that was selected. In other words, we pretend that the subject remained fixated on the selected object after clicking.

But before we do this, we should double check that trials also **start** at the same frame (-90). Once we pass this sanity check we can pad frames at the end.

```
start_frame <- edat %>%
  group_by(sub_id, t_id) %>%
  summarise(min_f_c = min(f_c), # get the minimum frame per trial
            .groups = "drop") %>%
  pull(min_f_c) %>%
  unique() # what are the unique values?

## if the value is the same for every trial, there should be
## just one element in this vector
stopifnot( length(start_frame) == 1L )

start_frame
```

```
[1] -90
```

2.3.1 Activity: Selected object

Which object was selected on each trial? The `trials` table tells us which location was clicked (1, 2, 3, 4) but not which object. We need to figure out which object was clicked by retrieving that information from the `screens` table. The result should have the format below.

```
# A tibble: 5,644 x 3
   sub_id  t_id role
    <int> <int> <chr>
 1      1     1 target
```

```
 2      1     2 target
 3      1     3 target
 4      1     4 target
 5      1     6 target
 6      1     7 target
 7      1    11 target
 8      1    13 target
 9      1    14 target
10      1    16 target
# ... with 5,634 more rows
```

! Solution

```
## which object was selected on each trial?
selections <- trials %>%
  inner_join(stimuli, "iv_id") %>%
  inner_join(screens, c("s_id", "resploc" = "loc")) %>%
  select(sub_id, t_id, role)
```

Now that we know what object was selected, we want to pad trials up to the latest frame in the dataset, which we determined during epoching as frame 90 (that is, 1.5 seconds after the disambiguation point).

We will use the `crossing()` function (from **{tidyr}**) to create a table with all combinations of the rows from `selections` with frames `f_c` from 0 to 90. Then, in the next activity, we will use `anti_join()` to pull out the combinations that are missing from `pog`, and use them in padding.

```
all_frames <- crossing(selections, tibble(f_c = 0:90))

all_frames
```

```
# A tibble: 513,604 x 4
   sub_id  t_id role     f_c
    <int> <int> <chr>  <int>
 1      1     1 target     0
 2      1     1 target     1
 3      1     1 target     2
 4      1     1 target     3
 5      1     1 target     4
 6      1     1 target     5
```

```
 7      1     1 target     6
 8      1     1 target     7
 9      1     1 target     8
10      1     1 target     9
# ... with 513,594 more rows
```

2.3.2 Activity: Pad frames

Use `anti_join()` to find out which frames in `all_frames` are missing from `pog`. Concatenate these frames onto `pog`, storing the result in `pog_cts`. The resulting table should have a variable `pad` which is `FALSE` if the frame is an original one, and `TRUE` if it was added through the padding procedure. Sort the rows of `pog_cts` by `sub_id`, `t_id`, and `f_c`. The format is shown below.

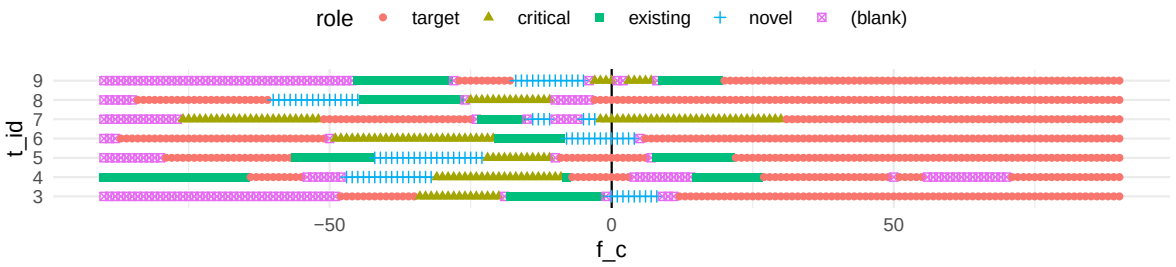
```
# A tibble: 1,021,288 x 5
   sub_id  t_id   f_c role   pad
    <int> <int> <int> <chr>  <lgl>
 1      1     1   -90 target FALSE
 2      1     1   -89 target FALSE
 3      1     1   -88 target FALSE
 4      1     1   -87 target FALSE
 5      1     1   -86 target FALSE
 6      1     1   -85 target FALSE
 7      1     1   -84 target FALSE
 8      1     1   -83 target FALSE
 9      1     1   -82 target FALSE
10      1     1   -81 target FALSE
# ... with 1,021,278 more rows
```

! Solution

One thing that may have happened in the process above is that `role` is no longer a factor. So let's convert it back before we finish.

```
pog_cts2 <- pog_cts %>%
  mutate(role = fct_relevel(role, c("target", "critical",
                                    "existing", "novel", "(blank)")))
```

Now let's double check that the padding worked by looking again at some trials from subject 3.



Looks good. Now let's save all our hard work so that we can use `pog_cts2` as a starting point for analysis.

```
saveRDS(pog_cts2, "data-derived/pog_cts.rds")
```

Part II

Visualization and Analysis

3 Plot probabilities

In the last chapter, we completed data preprocessing and saved the resulting data to as an R binary RDS file, `pog_cts.rds`. In this chapter, we will import the data and use it to recreate some of the figures in Weighall et al. (2017).

First, let's load in **{tidyverse}** and then import the point-of-gaze data.

```
library("tidyverse")

pog_cts <- read_rds("data-derived/pog_cts.rds")
```

As usual, the first thing we should do is have a look at our data.

```
# A tibble: 1,021,288 x 5
   sub_id  t_id   f_c role   pad
    <int> <int> <int> <fct>  <lgl>
 1      1     1   -90 target FALSE
 2      1     1   -89 target FALSE
 3      1     1   -88 target FALSE
 4      1     1   -87 target FALSE
 5      1     1   -86 target FALSE
 6      1     1   -85 target FALSE
 7      1     1   -84 target FALSE
 8      1     1   -83 target FALSE
 9      1     1   -82 target FALSE
10      1     1   -81 target FALSE
# ... with 1,021,278 more rows
```

The data has `sub_id` and `t_id` which identify individual subjects and trials-within-subjects, respectively. But we are missing iformation about what group the subject belongs to (adult or child) and what experimental condition each trial belongs to.

3.1 Merge eye data with information about group and condition

3.1.1 Activity: Get trial condition

The first step is to create `trial_cond`, which has information about the group that each subject belongs to, the competitor type (existing or novel), and the condition (the identity of the critical object). The information we need is distributed across the `subjects`, `trials`, and `stimuli` tables (see Appendix A). Create `trial_cond` so that the resulting table matches the format below.

```
# A tibble: 5,644 x 5
   sub_id group  t_id ctype crit
    <int> <chr> <int> <chr> <chr>
 1      1 adult     1 novel competitor-day2
 2      1 adult     2 novel competitor-day1
 3      1 adult     3 exist competitor
 4      1 adult     4 exist competitor
 5      1 adult     6 novel untrained
 6      1 adult     7 novel competitor-day1
 7      1 adult    11 novel untrained
 8      1 adult    13 novel competitor-day2
 9      1 adult    14 novel untrained
10      1 adult    16 novel untrained
# ... with 5,634 more rows
```

! Solution

```
trials <- read_csv("data-raw/trials.csv",
                   col_types = "iiiiii")

stimuli <- read_csv("data-raw/stimuli.csv",
                    col_types = "iiciccc")

subjects <- read_csv("data-raw/subjects.csv",
                     col_types = "ic")

trial_cond <- trials %>%
  inner_join(stimuli, "iv_id") %>%
  inner_join(subjects, "sub_id") %>%
  select(sub_id, group, t_id, ctype, crit)
```

3.2 Plot probabilities for existing competitors

We want to determine the probability of looking at each image type at each frame in each condition. We will do this first for the existing competitors. Note there were two conditions here, indexed by `crit`: `competitor` and `unrelated`, corresponding to whether the critical image was a competitor or an unrelated item.

3.2.1 Activity: Probs for `exist` condition

From `trial_cond`, include only those trials where `ctype` takes on the value `exist`, combine with `pog_cts`, and then count the number of frames in each region for every combination of the levels of group (`adult`, `child`) and `crit` (`competitor`, `unrelated`). The resulting table should have the format below, where `Y` is the number of frames for each combination. While you're at it, convert `f_c` to milliseconds (`1000 * f_c / 60`). Call the resulting table `count_exist`.

Hint: Counting things

Use the `count()` function from **{dplyr}**. Take note of the `.drop` argument to deal with possible situations where there are zero observations. For example:

```
pets <- tibble(animal = factor(rep(c("dog", "cat", "ferret"), c(3, 2, 0)),
                               levels = c("dog", "cat", "ferret")))

pets
```

```
# A tibble: 5 x 1
  animal
  <fct>
1 dog
2 dog
3 dog
4 cat
5 cat
```

```
pets %>%
  count(animal)
```

```
# A tibble: 2 x 2
  animal     n
  <fct>  <int>
1 dog        3
2 cat        2
```

```r
pets %>%
  count(animal, .drop = FALSE)
```

```
# A tibble: 3 x 2
  animal     n
  <fct>  <int>
1 dog        3
2 cat        2
3 ferret     0
```

```
# A tibble: 3,620 x 6
   group crit         f_c role         Y     ms
   <chr> <chr>      <int> <fct>    <int>  <dbl>
 1 adult competitor   -90 target      54 -1500
 2 adult competitor   -90 critical    55 -1500
 3 adult competitor   -90 existing    55 -1500
 4 adult competitor   -90 novel       75 -1500
 5 adult competitor   -90 (blank)    181 -1500
 6 adult competitor   -89 target      59 -1483.
 7 adult competitor   -89 critical    60 -1483.
 8 adult competitor   -89 existing    58 -1483.
 9 adult competitor   -89 novel       74 -1483.
10 adult competitor   -89 (blank)    169 -1483.
# ... with 3,610 more rows
```

! Solution

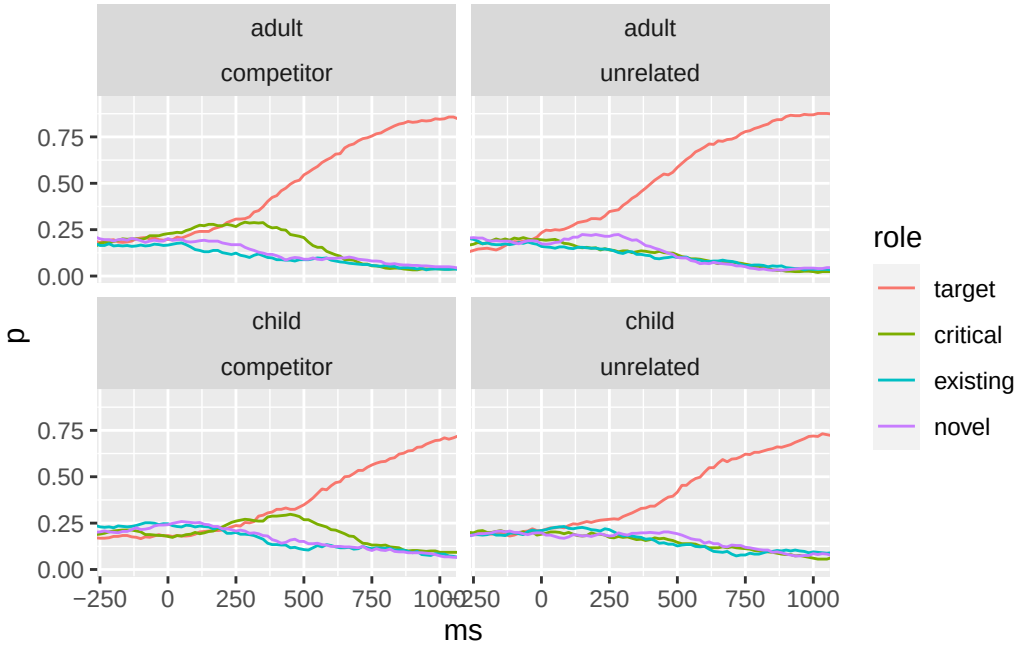
```r
count_exist <- trial_cond %>%
  filter(ctype == "exist") %>%
  inner_join(pog_cts, c("sub_id", "t_id")) %>%
  count(group, crit, f_c, role, name = "Y", .drop = FALSE) %>%
  mutate(ms = 1000 * f_c / 60)
```

To calculate the probability for each value of `role`, we need to calculate the number of opportunities for each combination of `group`, `crit`, and `f_c`, storing this in `N`. We do this using a windowed mutate, grouping the data before adding `N` for each group. We can then calculate the probability as `p = Y / N`.

```r
prob_exist <- count_exist %>%
  group_by(group, crit, f_c) %>%
  mutate(N = sum(Y), p = Y / N) %>%
  ungroup()
```

Now we are ready to plot.

```r
ggplot(prob_exist %>% filter(role != "(blank)"),
       aes(ms, p, color = role)) +
  geom_line() +
  facet_wrap(group ~ crit, nrow = 2)  +
  coord_cartesian(xlim = c(-200, 1000))
```



4 Polynomial regression

For more information about these approaches, see Barr (2008) and Mirman, Dixon, and Magnuson (2008). It is also possible to use Generalized Additive Mixed Models (GAMMs), which can more easily accommodate arbitrary wiggly patterns and asymptotes, but that is beyond the current scope of this textbook.

```
library("tidyverse")

pog <- read_rds("data-derived/pog_cts.rds")
```

```
# A tibble: 1,021,288 x 5
   sub_id  t_id   f_c role   pad
    <int> <int> <int> <fct>  <lgl>
 1      1     1   -90 target FALSE
 2      1     1   -89 target FALSE
 3      1     1   -88 target FALSE
 4      1     1   -87 target FALSE
 5      1     1   -86 target FALSE
 6      1     1   -85 target FALSE
 7      1     1   -84 target FALSE
 8      1     1   -83 target FALSE
 9      1     1   -82 target FALSE
10      1     1   -81 target FALSE
# ... with 1,021,278 more rows
```

4.1 Binning data

We are going to follow the Mirman, Dixon, and Magnuson (2008) approach. What we want to do first is to model the shape of the curve for existing competitors and see if it differs across children and adults.

We will perform separate by-subject and by-item analysis. The reason why this is needed is that we have to first aggregate the data in order to deal with the frame-by-frame dependencies. A common approach is to aggregate frames into 50 ms bins (i.e. each having 3 frames).

The general formula for binning data is:

`bin = floor( (frame + binsize/2) / binsize ) * binsize`

To bin things up into bins of 3 frames each, it would be

`bin = floor( (frame + 3/2) / 3) * 3`

To get a sense for how this formula works, try it out in the console.

```
sample_frames <- -10:10

rbind(frame = sample_frames,
      bin = floor( (sample_frames + 3/2) / 3) * 3)
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
frame  -10   -9   -8   -7   -6   -5   -4   -3   -2    -1     0     1     2
bin     -9   -9   -9   -6   -6   -6   -3   -3   -3     0     0     0     3
      [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21]
frame     3     4     5     6     7     8     9    10
bin       3     3     6     6     6     9     9     9
```

Why add half a bin?

Shifting frames forward by half of the binsize gives us more accurate bin numbering. To see why, consider the unshifted version to our shifted version above.

```
## unshifted version
rbind(frame = sample_frames,
      bin = floor(sample_frames / 3) * 3)
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13]
frame  -10   -9   -8   -7   -6   -5   -4   -3   -2    -1     0     1     2
bin    -12   -9   -9   -9   -6   -6   -6   -3   -3    -3     0     0     0
      [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21]
frame     3     4     5     6     7     8     9    10
bin       3     3     3     6     6     6     9     9
```

Note that in the shifted version, the bin name corresponds to the median frame contained in the bin, whereas in the unshifted version, it corresponds to the first frame in the bin. For instance, bin 0, contains -1, 0, and 1 in the shifted version; in the unshifted version, it contains 0, 1, and 2.

4.1.1 Activity: Calculating bins

Following the above logic, add the variables `bin` and `ms` (time in milliseconds for the corresponding bin) to the `pog` table. Save the result as `pog_calc`.

```
# A tibble: 1,021,288 x 7
   sub_id  t_id   f_c role   pad     bin    ms
    <int> <int> <int> <fct>  <lgl> <int> <int>
 1      1     1   -90 target FALSE   -30  -500
 2      1     1   -89 target FALSE   -30  -500
 3      1     1   -88 target FALSE   -29  -483
 4      1     1   -87 target FALSE   -29  -483
 5      1     1   -86 target FALSE   -29  -483
 6      1     1   -85 target FALSE   -28  -466
 7      1     1   -84 target FALSE   -28  -466
 8      1     1   -83 target FALSE   -28  -466
 9      1     1   -82 target FALSE   -27  -450
10      1     1   -81 target FALSE   -27  -450
# ... with 1,021,278 more rows
```

! Solution

```
pog_calc <- pog %>%
  mutate(bin = floor((f_c + 3/2) / 3) %>% as.integer(),
         ms = as.integer(1000 * bin / 60))
```

4.1.2 Activity: Count frames in bins

For the analysis below, we're going to focus on the existing competitors (`ctype == "exist"`). Link the `pog_calc` data to information about subjects and conditions (`crit`) to create the following table, where `Y` is the number of frames observed for the particular combination of `sub_id`, `group`, `crit`, `ms`, and `role`. Save the resulting table as `pog_subj_y`.

```
# A tibble: 50,630 x 6
   sub_id group crit          ms role         Y
    <int> <chr> <chr>      <int> <fct>    <int>
 1      1 adult competitor  -500 target       6
 2      1 adult competitor  -500 critical     4
 3      1 adult competitor  -500 existing     4
```

```
 4      1 adult competitor  -500 novel        6
 5      1 adult competitor  -500 (blank)      0
 6      1 adult competitor  -483 target       7
 7      1 adult competitor  -483 critical     5
 8      1 adult competitor  -483 existing     8
 9      1 adult competitor  -483 novel        8
10      1 adult competitor  -483 (blank)      2
# ... with 50,620 more rows
```

! Solution

```
subjects <- read_csv("data-raw/subjects.csv",
                     col_types = "ic")

trials <- read_csv("data-raw/trials.csv",
                   col_types = "iiiiii")

stimuli <- read_csv("data-raw/stimuli.csv",
                    col_types = "iiciccc")

pog_subj_y <- pog_calc %>%
  inner_join(subjects, "sub_id") %>%
  inner_join(trials, c("sub_id", "t_id")) %>%
  inner_join(stimuli, c("iv_id")) %>%
  filter(ctype == "exist") %>%
  count(sub_id, group, crit, ms, role,
        name = "Y", .drop = FALSE)
```

4.1.3 Activity: Compute probabilities

Now add in variables `N`, the total number of frames for a given combination of `sub_id`, `group`, `crit`, and `ms`, and `p`, which is the probability (`Y / N`). Save the result as `pog_subj`.

```
# A tibble: 50,630 x 8
   sub_id group crit          ms role         Y     N      p
    <int> <chr> <chr>      <int> <fct>    <int> <int>  <dbl>
 1      1 adult competitor  -500 target       6    20 0.3
 2      1 adult competitor  -500 critical     4    20 0.2
 3      1 adult competitor  -500 existing     4    20 0.2
 4      1 adult competitor  -500 novel        6    20 0.3
```

```
 5      1 adult competitor  -500 (blank)      0    20 0
 6      1 adult competitor  -483 target       7    30 0.233
 7      1 adult competitor  -483 critical     5    30 0.167
 8      1 adult competitor  -483 existing     8    30 0.267
 9      1 adult competitor  -483 novel        8    30 0.267
10      1 adult competitor  -483 (blank)      2    30 0.0667
# ... with 50,620 more rows
```

Hint

Recall what we did back in the plotting chapter, when creating `probs_exist` (a windowed mutate). You'll need to do something like that again here.

Solution

```
pog_subj <- pog_subj_y %>%
  group_by(sub_id, group, crit, ms) %>%
  mutate(N = sum(Y),
         p = Y / N) %>%
  ungroup()
```

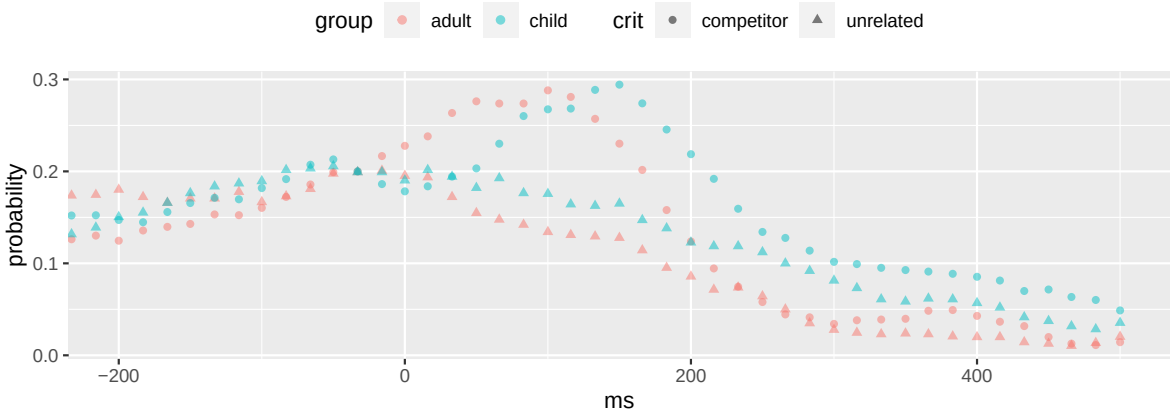
4.2 Plot mean probabilities

4.2.1 Activity: Mean probabilities

Let's now compute the mean probabilities for looks to the critical object across groups (adults, children) and condition (competitor, unrelated). First calculate the table `pog_means` below, then use it to create the graph below.

```
# A tibble: 244 x 4
   group crit          ms probability
   <chr> <chr>      <int>       <dbl>
 1 adult competitor  -500       0.137
 2 adult competitor  -483       0.140
 3 adult competitor  -466       0.131
 4 adult competitor  -450       0.129
 5 adult competitor  -433       0.135
 6 adult competitor  -416       0.137
 7 adult competitor  -400       0.135
```

```
 8 adult competitor  -383       0.138
 9 adult competitor  -366       0.137
10 adult competitor  -350       0.129
# ... with 234 more rows
```



! Solution

```
pog_means <- pog_subj %>%
  filter(role == "critical") %>%
  group_by(group, crit, ms) %>%
  summarize(probability = mean(p),
            .groups = "drop")
```

```
ggplot(pog_means, aes(ms, probability,
                      shape = crit, color = group)) +
  geom_point(alpha = .5) +
  coord_cartesian(xlim = c(-200, 500)) +
  theme(legend.position = "top")
```

4.3 Polynomial regression

Our task now is to fit the functions shown in the above figure using orthogonal polynomials. To avoid asymptotes, we will limit our analysis to 200 to 500 ms window, which is where the function seems to be changing.

The first thing we will do is prepare the data, adding in deviation-coded numerical predictors for `group` (`G`) and `crit` (`C`).

We will load in the R packages **{lme4}** for fitting linear mixed-effects models, and **{polypoly}** for working with orthogonal polynomials.

```
# if you don't have it, type
# install.packages("polypoly") # in the console
library("polypoly")
library("lme4")

pog_prep <- pog_subj %>%
  filter(role == "critical", ms >= -200) %>%
  mutate(G = if_else(group == "child", 1/2, -1/2),
         C = if_else(crit == "competitor", 1/2, -1/2))

## check that we didn't make any errors
pog_prep %>%
  distinct(group, crit, G, C)
```

```
# A tibble: 4 x 4
  group crit           G     C
  <chr> <chr>      <dbl> <dbl>
1 adult competitor  -0.5   0.5
2 adult unrelated   -0.5  -0.5
3 child competitor   0.5   0.5
4 child unrelated    0.5  -0.5
```

```
pog_3 <- pog_prep %>%
  poly_add_columns(ms, degree = 3) %>%
  select(sub_id, group, G, crit, C, ms, p, ms1, ms2, ms3)
```

```
mod_3 <- lmer(p ~ (ms1 + ms2 + ms3) * G * C +
                ((ms1 + ms2 + ms3) * C || sub_id),
              data = pog_3)

summary(mod_3)
```

```
Linear mixed model fit by REML ['lmerMod']
Formula: p ~ (ms1 + ms2 + ms3) * G * C + ((1 | sub_id) + (0 + ms1 | sub_id) +
    (0 + ms2 | sub_id) + (0 + ms3 | sub_id) + (0 + C | sub_id) +
```

```
    (0 + ms1:C | sub_id) + (0 + ms2:C | sub_id) + (0 + ms3:C |      sub_id))
   Data: pog_3

REML criterion at convergence: -14758.4

Scaled residuals:
    Min      1Q  Median      3Q     Max
-3.2565 -0.5914 -0.0358  0.5113  5.4038

Random effects:
 Groups   Name        Variance Std.Dev.
 sub_id   (Intercept) 0.001249 0.03534
 sub_id.1 ms1         0.054835 0.23417
 sub_id.2 ms2         0.027294 0.16521
 sub_id.3 ms3         0.026226 0.16194
 sub_id.4 C           0.003698 0.06081
 sub_id.5 ms1:C       0.128879 0.35900
 sub_id.6 ms2:C       0.106783 0.32678
 sub_id.7 ms3:C       0.061623 0.24824
 Residual             0.005815 0.07625
Number of obs: 7138, groups:  sub_id, 83

Fixed effects:
             Estimate Std. Error t value
(Intercept)  0.133722   0.003983  33.569
ms1         -0.360730   0.026378 -13.676
ms2         -0.188137   0.019077  -9.862
ms3          0.134597   0.018736   7.184
G            0.027162   0.007967   3.409
C            0.030741   0.006915   4.445
ms1:G        0.108453   0.052756   2.056
ms2:G       -0.035381   0.038154  -0.927
ms3:G       -0.088315   0.037473  -2.357
ms1:C        0.058203   0.041147   1.414
ms2:C       -0.193833   0.037774  -5.131
ms3:C        0.045101   0.029710   1.518
G:C          0.001996   0.013831   0.144
ms1:G:C      0.068046   0.082295   0.827
ms2:G:C      0.082218   0.075548   1.088
ms3:G:C     -0.127579   0.059420  -2.147
```

```
Correlation matrix not shown by default, as p = 16 > 12.
Use print(x, correlation=TRUE)  or
    vcov(x)        if you need it
```

It converged! Before we get too excited, plot the model fitted values against the observed values to assess the quality of the fit.

We need data to feed in to the `predict()` function in order to generate our fitted values. We'll use `pog_means` for this purpose, adding in all of the predictors we need for the model, and restricting the range.

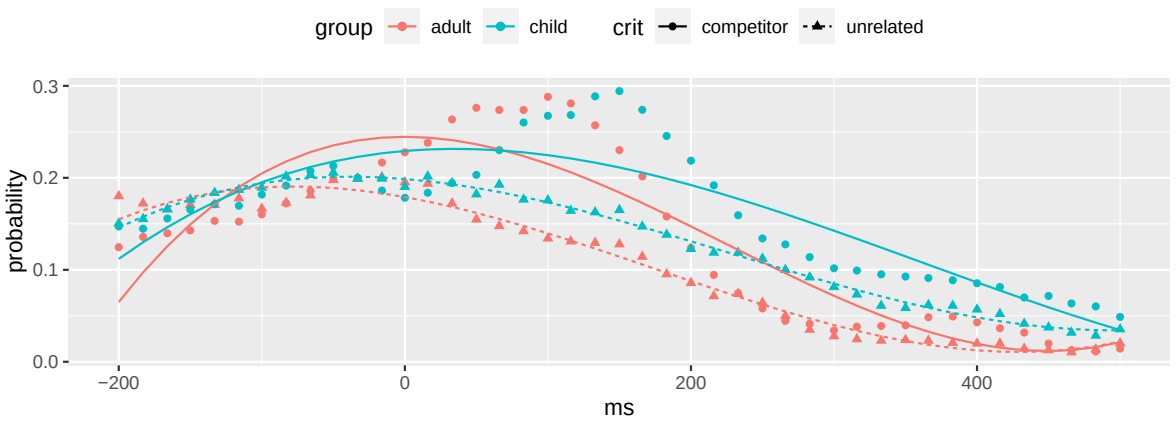
```
pog_new <- pog_means %>%
  filter(ms >= -200) %>%
  mutate(G = if_else(group == "child", 1/2, -1/2),
         C = if_else(crit == "competitor", 1/2, -1/2))
```

Now we are ready to feed it into `predict()` to generate fitted values. Note that we want to make predictions for the "typical" subject with random effects of zero, which requires setting `re.form = NA` for the `predict()` function. See `?predict.merMod` for details. We use `newdata = .` to send the current data from our pipeline as the argument for `newdata`.

```
fits_3 <- pog_new %>%
  poly_add_columns(ms, degree = 3) %>%
  mutate(fitted = predict(mod_3, newdata = .,
                          re.form = NA))
```

Now we plot the fitted values (lines) against observed (points).

```
ggplot(fits_3,
       aes(ms, probability,
           shape = crit, color = group)) +
  geom_point() +
  geom_line(aes(y = fitted, linetype = crit)) +
  theme(legend.position = "top")
```



Not good. We might want to try a higher order model. Alternatively, we can restrict the range further to get rid of asymptotic elements in the later part of the window. Let's try the latter first because that's fairly easy.

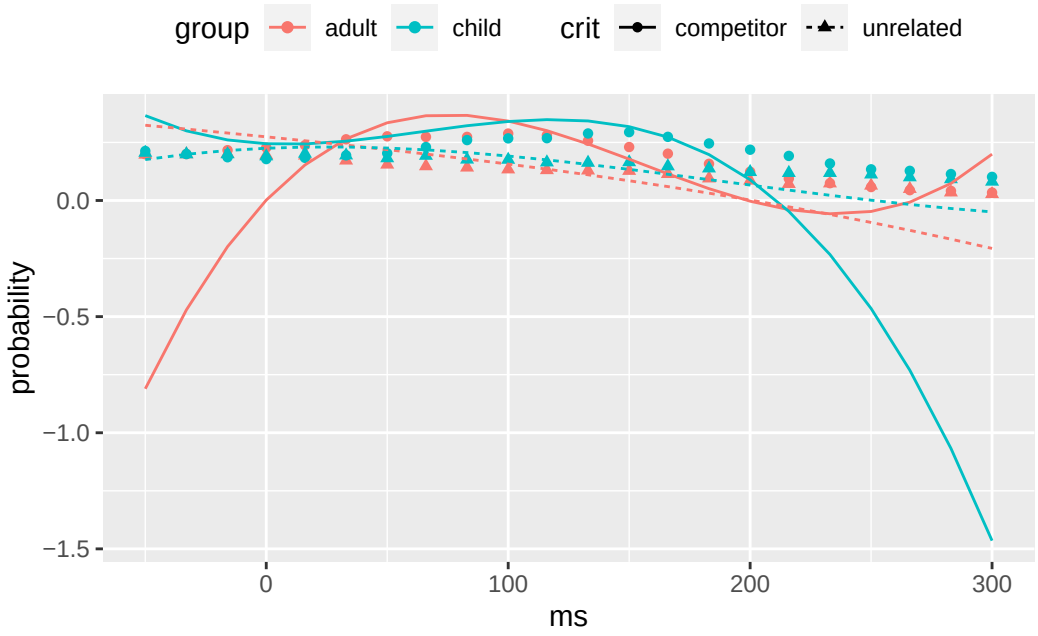
```
pog_3b <- pog_3 %>%
  filter(between(ms, -50L, 300L))

## refit with a different dataset
mod_3b <- update(mod_3, data = pog_3b)
```

Generate fitted values from the new model and plot.

```
fits_3b <- pog_new %>%
  filter(between(ms, -50L, 300L)) %>%
  poly_add_columns(ms, degree = 3) %>%
  mutate(fitted = predict(mod_3b, newdata = .,
                          re.form = NA))

ggplot(fits_3b,
       aes(ms, probability,
           shape = crit, color = group)) +
  geom_point() +
  geom_line(aes(y = fitted, linetype = crit)) +
  theme(legend.position = "top")
```



Well, that is even worse.

4.3.1 Activity: Quintic model

A cubic is really not enough. Try to fit a quintic (5th order) function on the reduced data range (-50 ms to 300 ms). Use the `bobyqa` optimizer to get `lmer()` to converge (`control = lmerControl(optimizer = "bobyqa")`), and fit it with `REML=FALSE`.

Then, follow the example above to assess the quality of fit using a plot.

! Solution

```
pog_5 <- pog_prep %>%
  filter(between(ms, -50L, 300L)) %>%
  poly_add_columns(ms, degree = 5)

mod_5 <- lmer(p ~ (ms1 + ms2 + ms3 + ms4 + ms5) * G * C +
                ((ms1 + ms2 + ms3 + ms4 + ms5) * C || sub_id),
              data = pog_5, REML=FALSE,
              control = lmerControl(optimizer = "bobyqa"))
```

Now evaluate the fit.

```
fits_5 <- pog_new %>%
  filter(between(ms, -50L, 300L)) %>%
  poly_add_columns(ms, degree = 5) %>%
  mutate(fitted = predict(mod_5, newdata = .,
                          re.form = NA))

ggplot(fits_5,
       aes(ms, probability,
           shape = crit, color = group)) +
  geom_point() +
  geom_line(aes(y = fitted, linetype = crit)) +
  theme(legend.position = "top")
```

OK, that's a fit that we can be happy with.

Let's have a look at the model output.

```
summary(mod_5)
```

```
Linear mixed model fit by maximum likelihood  ['lmerMod']
Formula: p ~ (ms1 + ms2 + ms3 + ms4 + ms5) * G * C + ((1 | sub_id) + (0 +
    ms1 | sub_id) + (0 + ms2 | sub_id) + (0 + ms3 | sub_id) +
    (0 + ms4 | sub_id) + (0 + ms5 | sub_id) + (0 + C | sub_id) +
    (0 + ms1:C | sub_id) + (0 + ms2:C | sub_id) + (0 + ms3:C |
    sub_id) + (0 + ms4:C | sub_id) + (0 + ms5:C | sub_id))
   Data: pog_5
Control: lmerControl(optimizer = "bobyqa")

     AIC      BIC   logLik deviance df.resid
 -9233.6  -9004.1   4653.8  -9307.6     3615

Scaled residuals:
    Min      1Q  Median      3Q     Max
-3.2631 -0.4883 -0.0013  0.4694  4.4521

Random effects:
 Groups    Name        Variance Std.Dev.
 sub_id    (Intercept) 0.002499 0.04999
 sub_id.1  ms1         0.046758 0.21624
 sub_id.2  ms2         0.017703 0.13305
 sub_id.3  ms3         0.014331 0.11971
 sub_id.4  ms4         0.005942 0.07709
 sub_id.5  ms5         0.006101 0.07811
 sub_id.6  C           0.007937 0.08909
 sub_id.7  ms1:C       0.113327 0.33664
 sub_id.8  ms2:C       0.075215 0.27425
 sub_id.9  ms3:C       0.061012 0.24701
 sub_id.10 ms4:C       0.020677 0.14380
 sub_id.11 ms5:C       0.021688 0.14727
 Residual              0.002148 0.04635
Number of obs: 3652, groups:  sub_id, 83

Fixed effects:
             Estimate Std. Error t value
(Intercept)  0.167871   0.005541  30.295
ms1         -0.209290   0.024008  -8.718
ms2         -0.121173   0.015042  -8.056
ms3          0.006115   0.013625   0.449
ms4          0.046584   0.009195   5.066
ms5          0.003498   0.009299   0.376
G            0.026723   0.011082   2.411
C            0.054710   0.009899   5.527
```

```
ms1:G        0.148235   0.048016   3.087
ms2:G        0.006309   0.030084   0.210
ms3:G       -0.070349   0.027249  -2.582
ms4:G        0.018576   0.018390   1.010
ms5:G        0.006853   0.018597   0.368
ms1:C        0.010683   0.037648   0.284
ms2:C       -0.191025   0.030953  -6.171
ms3:C        0.006511   0.028053   0.232
ms4:C        0.099358   0.017347   5.728
ms5:C       -0.003818   0.017695  -0.216
G:C         -0.009915   0.019798  -0.501
ms1:G:C      0.144761   0.075295   1.923
ms2:G:C      0.046990   0.061907   0.759
ms3:G:C     -0.156586   0.056106  -2.791
ms4:G:C      0.009317   0.034695   0.269
ms5:G:C      0.061278   0.035390   1.732


Correlation matrix not shown by default, as p = 24 > 12.
Use print(x, correlation=TRUE)  or
    vcov(x)        if you need it
```

Now let's use model comparison to answer our question: do the time-varying components for lexical competition vary across children and adults?

```
mod_5_drop <-
  update(mod_5,
         . ~ . -ms1:G:C -ms2:G:C -ms3:G:C -ms4:G:C -ms5:G:C)

anova(mod_5, mod_5_drop)
```

```
Data: pog_5
Models:
mod_5_drop: p ~ ms1 + ms2 + ms3 + ms4 + ms5 + G + C + (1 | sub_id) + (0 + ms1 | sub_id) + (0
mod_5: p ~ (ms1 + ms2 + ms3 + ms4 + ms5) * G * C + ((1 | sub_id) + (0 + ms1 | sub_id) + (0 +
           npar     AIC     BIC logLik deviance  Chisq Df Pr(>Chisq)
mod_5_drop   32 -9229.0 -9030.5 4646.5  -9293.0
mod_5        37 -9233.6 -9004.1 4653.8  -9307.6 14.653  5    0.01195 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

There are further things we could potentially do with this model, including performing model comparison on the time-varying components. One thing we probably should do would be to repeat all the above steps, but treating items as a random factor instead of subjects.

One issue with polynomial regression is that the complexity of the model is likely to give rise to convergence problems. One strategy is to estimate the parameters using re-sampling techniques, which we'll learn about in the next chapter.

Before we do that, let's save `pog_subj`, because we'll need it for the next set of activities.

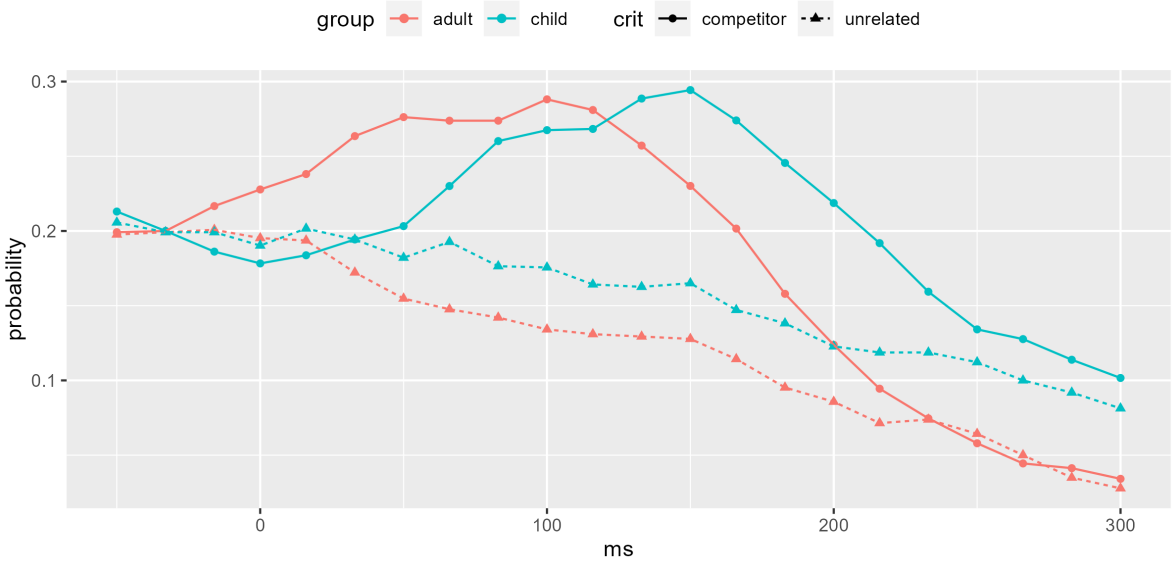
```
pog_subj %>%
  saveRDS(file = "data-derived/pog_subj.rds")
```

5 Cluster-permutation analysis

Cluster-permutation analysis was first developed for statistical problems in fMRI (Bullmore et al. 1999) and EEG/MEG research (Maris and Oostenveld 2007). It developed as a means of controlling the false positive rates for numerous tests across electrode, voxel, and time, without incurring the catastrophic hit to power that would occur using conventional (Bonferroni-type) correction methods. Usually you would use this approach when you more are interested in **when** an effect arises than in the overall shape of effects across an analysis window. However, I would be remiss not to mention a recent article which takes a critical view on its ability to establish the locus of effects in time (Sassenhagen and Draschkow 2019). But it is still useful for establishing a time range around which 'something is happening' even if it doesn't allow us to express uncertainty around the boundaries of that time range (which would be even more useful).

5.1 Our task

What we are going to do is run a cluster-permutation analysis on the data below, to see when the group-by-competition interaction is likely to be reliable.



For the analysis, we'll need two development packages (only available on github): **{exchangr}** and **{clusterperm}**. The former does exchangeable permutations, and the latter more specifically for this kind of analysis.

We're going to use the data `pog_subj` that we created in the last chapter and saved in the `data-derived/` subdirectory. We're going to be performing ANOVAs on the data using `aov()`, so we'll need to define our independent variables as factors.

```
library("tidyverse")
library("exchangr") # remotes::install_github("dalejbarr/exchangr")
library("clusterperm") # remotes::install_github("dalejbarr/clustperm")

pog_subj <- read_rds("data-derived/pog_subj.rds") %>%
  filter(role == "critical",
         between(ms, -200, 300)) %>%
  select(-role, -Y, -N) %>%
  mutate(group = factor(group),
         crit = factor(crit),
         sub_id = factor(sub_id))
```

Let's take a moment to understand the `aov()` function from base R. Imagine that instead of having multiple frames, we wanted to run an ANOVA on a single time point, say at 150 ms. We have a mixed design (group = between-subjects; crit = within-subjects), so the way we would do it is as shown below.

```
pog_150 <- pog_subj %>%
  filter(ms == 150L) %>%
  select(-ms)

mod_aov <- aov(p ~ group * crit + Error(sub_id / crit),
               pog_150)

summary(mod_aov)
```

```
Error: sub_id
          Df Sum Sq Mean Sq F value  Pr(>F)
group      1 0.1067 0.10669   7.584 0.00727 **
Residuals 81 1.1395 0.01407
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Error: sub_id:crit
           Df Sum Sq Mean Sq F value   Pr(>F)
crit        1 0.5552  0.5552  27.098 1.43e-06 ***
group:crit  1 0.0075  0.0075   0.366    0.547
Residuals  81 1.6595  0.0205
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

5.1.1 Activity: `aov_by_bin()`

The `aov_by_bin()` function from **`{clusterperm}`** will run that same ANOVA at every bin in the dataset. Try it, plugging in the same formula from above. Save the result as `orig_result`.

! Solution

```
orig_result <- aov_by_bin(pog_subj, ms,
                          p ~ group * crit + Error(sub_id / crit))

orig_result
```

```
# A tibble: 93 x 4
      ms effect        stat      p
   <int> <chr>        <dbl>  <dbl>
 1  -200 group       0.0329 0.857
 2  -200 crit       -3.44   0.0672
 3  -200 group:crit -2.66   0.107
 4  -183 group       0.0456 0.831
 5  -183 crit       -2.18   0.143
 6  -183 group:crit -0.654  0.421
 7  -166 group      -0.189  0.665
 8  -166 crit       -1.11   0.295
 9  -166 group:crit -0.219  0.641
10  -150 group      -0.535  0.467
# ... with 83 more rows
```

The function `aov_by_bin()` returns the variable `stat`, which is a signed `F` statistic, and is positive or negative depending on the direction of the effect. It also returns `p`, which is the p-value for the effect at the corresponding bin.

A 'cluster' is defined as a set of temporally-adjacent bins where all of the test statistics have the same signs, and the p-values are all less than alpha (where alpha is the false positive level,

usually .05). We look for these temporally adjacent bins for each main effect or interaction.

In this case we have two main effects (`group` and `crit`) and one interaction (`group:crit`), and we can detect clusters for each one of these.

5.1.2 Activity: Detect clusters

The **`{clusterperm}`** package provides a function to do this, `detect_clusters_by_effect()`, which is fed the output of `aov_by_bin()`. Try applying this function to `orig_result`, and save the result as `clusters`.

! Solution

```
clusters <- orig_result %>%
  detect_clusters_by_effect(effect, ms, stat, p)

clusters
```

```
# A tibble: 3 x 5
  effect        b0    b1  sign   cms
  <chr>      <int> <int> <dbl> <dbl>
1 group        150   300    -1 169.
2 crit          33   216     1 284.
3 group:crit    33    66     1  21.5
```

`b0` and `b1` tell you the start and end frames for each cluster; `sign` gives you the direction of the effect, and `cms` gives you the "cluster mass statistic", which is the summed test statistics for the entire cluster.

5.2 Deriving null-hypothesis distributions through resampling

A permutation test has proceeds according to the following steps:

1. perform an analysis on the original data and store the resulting test statistic;
2. generate a null-hypothesis distribution for the test statistic by randomly permuting labels, re-running the analysis, and storing the test statistic many times;
3. compare the original test statistic to the distribution of statistics you generated in step 2 to determine how unlikely your original test statistic is under the null hypothesis.

> ⚠ Warning
>
> The logic of permutation tests is appealing, but you can really get into trouble very easily trying to apply it to multilevel data. There is a critical assumption that you need to ensure is honored, which is that **all exchanges you make when generating the null-hypothesis distribution are legitimate exchanges under H0**. To state this differently, each randomly re-labeled dataset should be one that **could** have existed had the experiment gone differently, but didn't.
> The functions in **{exchangr}** are there to help you meet this assumption, but they should not be used without understanding exactly what they do.

Let's see how easy it is to go wrong when exchanging labels on multilevel data. We decide that we want to randomly re-label adults and children in order to test the effect of `group` and/or the `group-by-crit` interaction. Run the `count()` function on the original data in `pog_subj` to see what is what.

```
pog_subj %>%
  count(sub_id, group)
```

```
# A tibble: 83 x 3
   sub_id group     n
   <fct>  <fct> <int>
 1 1      adult    62
 2 2      adult    62
 3 3      adult    62
 4 4      adult    62
 5 5      adult    62
 6 6      adult    62
 7 7      adult    62
 8 8      adult    62
 9 9      adult    62
10 10     adult    62
# ... with 73 more rows
```

Each subject was either an adult or a child, and has 62 observations. Now let's imagine we applied `shuffle()` without thinking, as an attempt to reattach the labels across subjects

```
set.seed(62) # so you get the same random result as me

pog_shuffled_bad <- pog_subj %>%
  shuffle(group)
```

```
pog_shuffled_bad
```

```
# A tibble: 5,146 x 5
   sub_id group crit          ms     p
   <fct>  <fct> <fct>      <int> <dbl>
 1 1      adult competitor  -200 0.2
 2 1      child competitor  -183 0.333
 3 1      adult competitor  -166 0.467
 4 1      adult competitor  -150 0.5
 5 1      child competitor  -133 0.467
 6 1      child competitor  -116 0.367
 7 1      adult competitor  -100 0.3
 8 1      child competitor   -83 0.367
 9 1      adult competitor   -66 0.4
10 1      child competitor   -50 0.4
# ... with 5,136 more rows
```

```
pog_shuffled_bad %>%
  count(sub_id, group)
```

```
# A tibble: 166 x 3
   sub_id group     n
   <fct>  <fct> <int>
 1 1      adult    33
 2 1      child    29
 3 2      adult    37
 4 2      child    25
 5 3      adult    28
 6 3      child    34
 7 4      adult    21
 8 4      child    41
 9 5      adult    38
10 5      child    24
# ... with 156 more rows
```

So now we can see the problem: in our re-labeled group, subject 1 is both adult and child! We have violated the exchangeability of the labels under the null hypothesis, creating an impossible dataset, and turning a between-subject factor into a within-subject factor. Any null-hypothesis distribution created from this manner of shuffling will be garbage.

What we need to do is to "nest" the 62 observations into a `list-column` before we do the shuffling, using the `nest()` function from `{tidyr}`. Once we've done this, then we can run the shuffle function, and then unnest the data back into it's original form.

5.2.1 Activity: Build a `nest()`

Let's try using the `nest()` function to create the data below from `pog_subj`. For guidance, look at the examples in the documentation (type `?nest` in the console). Save the result as `pog_nest`. Then try to `unnest()` the data.

```
# A tibble: 83 x 3
   sub_id group data
   <fct>  <fct> <list>
 1 1      adult <tibble [62 x 3]>
 2 2      adult <tibble [62 x 3]>
 3 3      adult <tibble [62 x 3]>
 4 4      adult <tibble [62 x 3]>
 5 5      adult <tibble [62 x 3]>
 6 6      adult <tibble [62 x 3]>
 7 7      adult <tibble [62 x 3]>
 8 8      adult <tibble [62 x 3]>
 9 9      adult <tibble [62 x 3]>
10 10     adult <tibble [62 x 3]>
# ... with 73 more rows
```

! Solution

```
pog_nest <- pog_subj %>%
  nest(data = c(-sub_id, -group))
```

```
## back to its original format
pog_nest %>%
  unnest(cols = c(data))
```

```
# A tibble: 5,146 x 5
   sub_id group crit          ms     p
   <fct>  <fct> <fct>      <int> <dbl>
 1 1      adult competitor  -200 0.2
 2 1      adult competitor  -183 0.333
 3 1      adult competitor  -166 0.467
```

```
 4 1      adult competitor  -150 0.5
 5 1      adult competitor  -133 0.467
 6 1      adult competitor  -116 0.367
 7 1      adult competitor  -100 0.3
 8 1      adult competitor   -83 0.367
 9 1      adult competitor   -66 0.4
10 1      adult competitor   -50 0.4
# ... with 5,136 more rows
```

OK, now that we've figured out how to nest data, we can apply `shuffle()` to the nested data and then unnest. We'll write a function to do this called `shuffle_ml()`. This will work with all the **`{clusterperm}`** functions as long as we name the first argument `.data`.

```
shuffle_ml <- function(.data) {
  .data %>%
    nest(data = c(-sub_id, -group)) %>%
    shuffle(group) %>%
    unnest(data)
}
```

Let's try it out and verify that it works as intended.

```
pog_subj %>%
  shuffle_ml() %>%
  count(sub_id, group)
```

```
# A tibble: 83 x 3
   sub_id group     n
   <fct>  <fct> <int>
 1 1      adult    62
 2 2      child    62
 3 3      adult    62
 4 4      adult    62
 5 5      child    62
 6 6      adult    62
 7 7      adult    62
 8 8      adult    62
 9 9      adult    62
10 10     adult    62
# ... with 73 more rows
```

Looks good! Now we are ready to use `shuffle_ml()` to build our NHD (null-hypothesis

distribution). Note that because we are only shuffling `group`, we can only use the NHD for tests of `group` and `group-by-crit`. If we also want to run a test of the main effect of `crit`, we would have to shuffle `crit` (and because it's a mixed design, you'd have to "synchronize" the shuffling over the levels of group, for which `shuffle_each_sync()` has been provided).

We'll use `cluster_nhds()` to get our null hypothesis distribution from 1000 monte carlo runs, and then the `pvalues()` function to derive p-values for our original clusters.

```
## make sure we're not using an old version
stopifnot(packageVersion("clusterperm") > "0.1.0")

## warning: can take many minutes!!
set.seed(62) # for reproducibility
nhds <- cluster_nhds(1000, pog_subj, ms,
                     p ~ group * crit + Error(sub_id / crit),
                     shuffle_ml)

cp_result <- pvalues(clusters %>% filter(effect != "crit"),
                     nhds %>% filter(effect != "crit"))

saveRDS(cp_result,
        file = "data-derived/cluster-permutation-result.rds")
```

Now let's print out the results.

```
cp_result
```

```
# A tibble: 2 x 6
  effect        b0    b1  sign   cms        p
  <chr>      <int> <int> <dbl> <dbl>    <dbl>
1 group        150   300    -1 169.  0.000999
2 group:crit    33    66     1  21.5 0.127
```

We have a clear main effect of `group` extending from 150-300 ms, which is where on the figure we can see the adults have a higher probability of looking at either picture than the children ($p < .001$). There was a group-by-competition interaction detected on the original data from 33-66 ms, but it was not statistically significant ($p = 0.127$).

References

Barr, Dale J. 2008. "Analyzing 'Visual World' Eyetracking Data Using Multilevel Logistic Regression." *Journal of Memory and Language* 59: 457–74.

Bullmore, Edward T, John Suckling, Stephan Overmeyer, Sophia Rabe-Hesketh, Eric Taylor, and Michael J Brammer. 1999. "Global, Voxel, and Cluster Tests, by Theory and Permutation, for a Difference Between Two Groups of Structural MR Images of the Brain." *IEEE Transactions on Medical Imaging* 18: 32–42.

Maris, Eric, and Robert Oostenveld. 2007. "Nonparametric Statistical Testing of EEG-and MEG-Data." *Journal of Neuroscience Methods* 164: 177–90.

Mirman, Daniel, James A Dixon, and James S Magnuson. 2008. "Statistical and computational models of the visual world paradigm: Growth curves and individual differences." *Journal of Memory and Language* 59 (4): 475–94.

Sassenhagen, Jona, and Dejan Draschkow. 2019. "Cluster-Based Permutation Tests of MEG/EEG Data Do Not Establish Significance of Effect Latency or Location." *Psychophysiology* 56: e13335.

Weighall, AR, Lisa-Marie Henderson, DJ Barr, Scott Ashley Cairney, and Mark Gareth Gaskell. 2017. "Eye-Tracking the Time-Course of Novel Word Learning and Lexical Competition in Adults and Children." *Brain and Language* 167: 13–27.

A Structure of Weighall et al. (2017) raw data

If you have trouble viewing the image below, you can download a PDF version of the figure.