Reproducibly Analysing Visual-World Eyetracking Data

Dale Barr

10/26/2022

Table of contents

Background **4**
Raw data . . . 4

I Pre-processing **5**

1 Import, epoching, and time-alignment **6**
1.1 Data import . . . 6
1.1.1 Activity: One Subject . . . 7
1.1.2 Activity: All Subjects . . . 9
1.2 Epoching and time-alignment . . . 12
1.2.1 Activity: Disambiguation Point . . . 13
1.2.2 Activity: Onset of StimSlide . . . 14
1.2.3 Activity: Combine origins . . . 14
1.2.4 Activity: Time-align . . . 15
1.3 Save the data . . . 16

2 Mapping gaze to areas of interest **17**
2.1 Image locations for each trial . . . 19
2.1.1 Activity: Get coordinates . . . 19
2.2 Identifying frames where the gaze cursor is within an AOI . . . 20
2.2.1 Activity: Create `frames_in` . . . 20
2.2.2 Activity: Create `frames_out` . . . 21
2.2.3 Activity: Combine into `pog` . . . 23
2.3 Dealing with trial dropouts . . . 24
2.3.1 Activity: Selected object . . . 25
2.3.2 Activity: Pad frames . . . 26

II Visualization and Analysis **29**

3 Plot probabilities **30**
3.1 Merge eye data with information about group and condition . . . 31
3.1.1 Activity: Get trial condition . . . 31
3.2 Plot probabilities for existing competitors . . . 32
3.2.1 Activity: Probs for `exist` condition . . . 32

Appendices **35**

References **36**

A Structure of Weighall et al. (2017) raw data **37**

Background

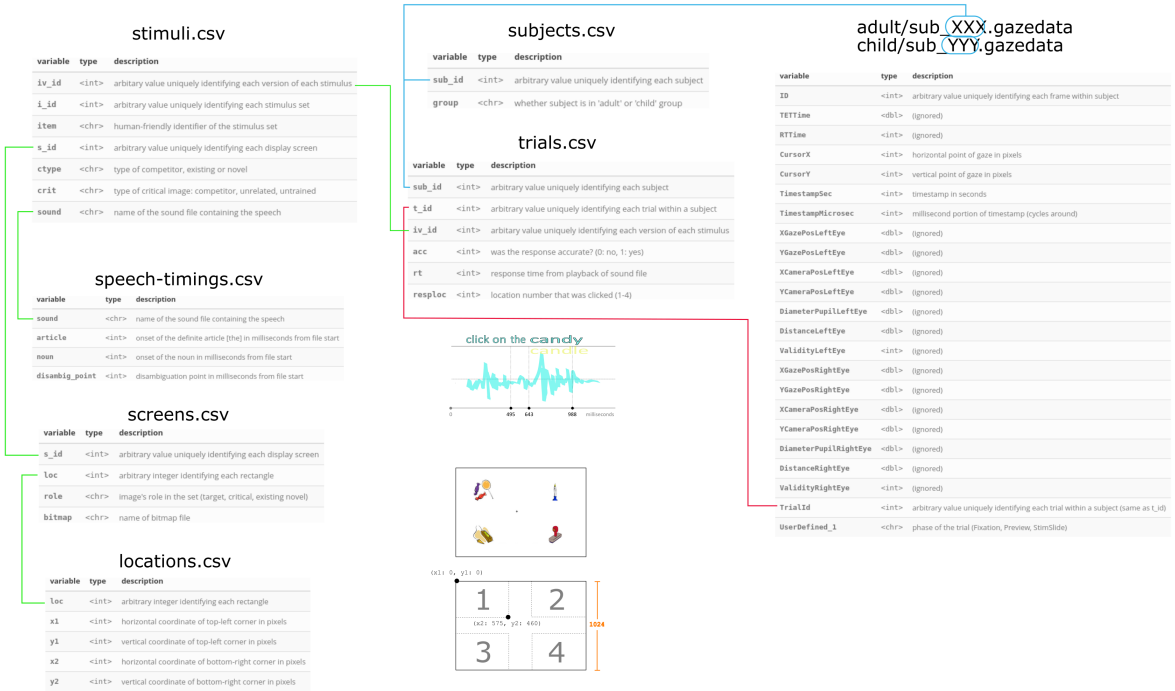
For this workshop, we will be reproducing data from a study by Weighall et al. (2017) on the role of sleep in learning novel words (lexical consolidation).

Weighall, A. R., Henderson, L. M., Barr, D. J., Cairney, S. A., & Gaskell, M. G. (2017). Eye-tracking the time-course of novel word learning and lexical competition in adults and children. *Brain and Language*, *167*, 13-27.

Raw data

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

The data has the structure below.

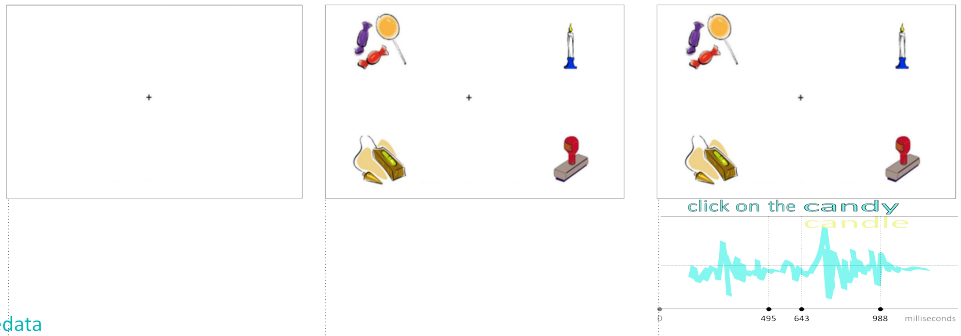


Part I

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		arbitrary value uniquely identifying each frame within subject
TETTime		(ignored)
RTTime		(ignored)
CursorX		horizontal point of gaze in pixels
CursorY		vertical point of gaze in pixels
TimestampSec		timestamp in seconds
TimestampMicrosec		millisecond portion of timestamp (cycles around)
XGazePosLeftEye		(ignored)
YGazePosLeftEye		(ignored)
XCameraPosLeftEye		(ignored)
YCameraPosLeftEye		(ignored)
DiameterPupilLeftEye		(ignored)
DistanceLeftEye		(ignored)
ValidityLeftEye		(ignored)
XGazePosRightEye		(ignored)
YGazePosRightEye		(ignored)
XCameraPosRightEye		(ignored)
YCameraPosRightEye		(ignored)
DiameterPupilRightEye		(ignored)
DistanceRightEye		(ignored)
ValidityRightEye		(ignored)
TrialId		arbitrary value uniquely identifying each trial within a subject (same as t_id)
UserDefined_1		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 **?@fig-epoching** 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.

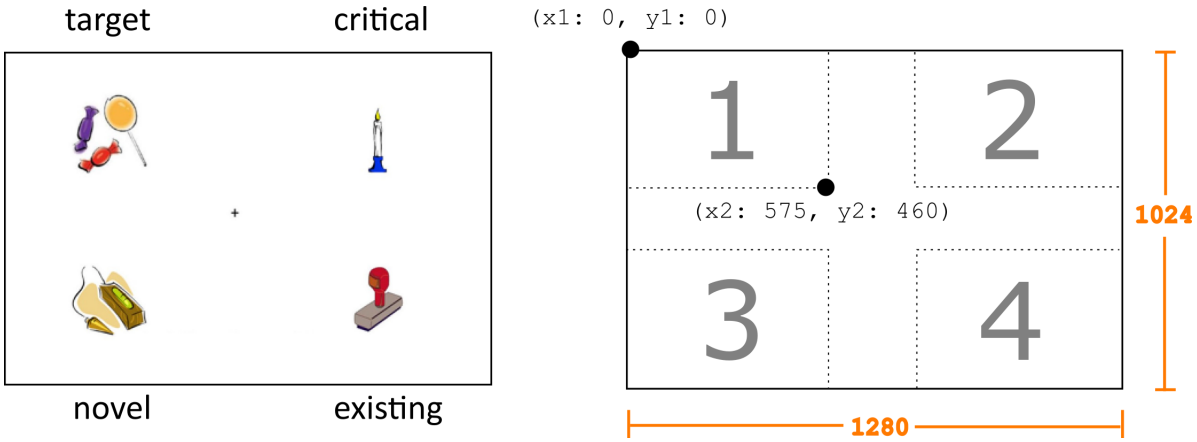
```
## if we haven't made a "data-derived" directory, do so now
if (!dir.exists("data-derived")) dir.create("data-derived")

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:

sub_id	t_id	f_c	x	y
1	1	-90	1010	209
1	1	-89	1010	214
1	1	-88	1020	216
1	1	-87	1014	217
1	1	-86	1027	221
1	1	-85	1027	220

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		arbitrary value uniquely identifying each display screen
loc		arbitrary integer identifying each rectangle
role		image's role in the set (target, critical, existing novel)
bitmap		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		arbitrary integer identifying each rectangle
x1		horizontal coordinate of top-left corner in pixels
y1		vertical coordinate of top-left corner in pixels
x2		horizontal coordinate of bottom-right corner in pixels
y2		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`).

```r
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.

```r
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
```

```r
## 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

```r
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`.

```r
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.

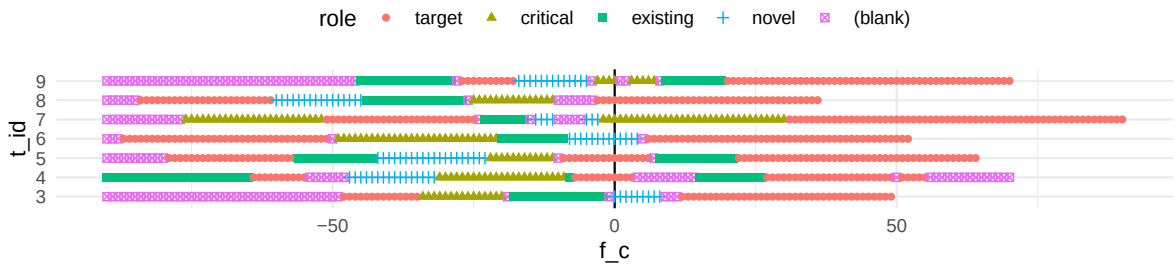
```
pog %>%
  pull(role) %>%
  levels()
```

```
[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.

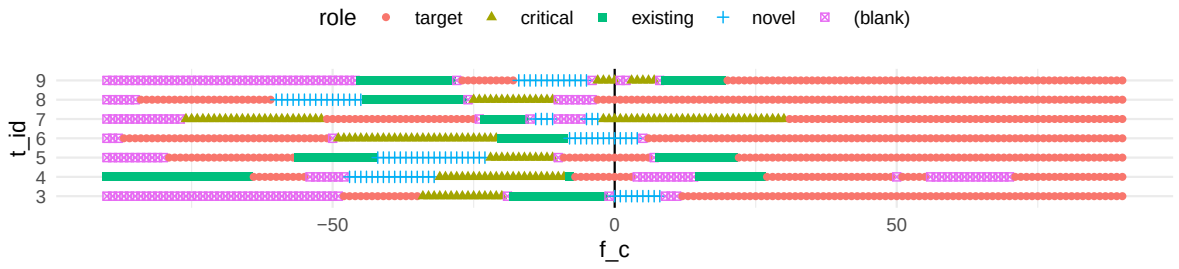
```
# A tibble: 1,021,288 x 5
   sub_id  t_id   f_c role   pad
    <int> <int> <int> <chr>  <lgl>
 1      1     1   -90 target FALSE
 2      1     1   -89 target FALSE
 3      1     1   -88 target FALSE
 4      1     1   -87 target FALSE
 5      1     1   -86 target FALSE
 6      1     1   -85 target 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")
```

```
-- Attaching packages --------------------------------------- tidyverse 1.3.2 --
v ggplot2 3.3.6      v purrr   0.3.5
v tibble  3.1.8      v dplyr   1.0.10
v tidyr   1.2.1      v stringr 1.4.1
v readr   2.1.3      v forcats 0.5.2
-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

```
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:

```r
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
```

```r
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

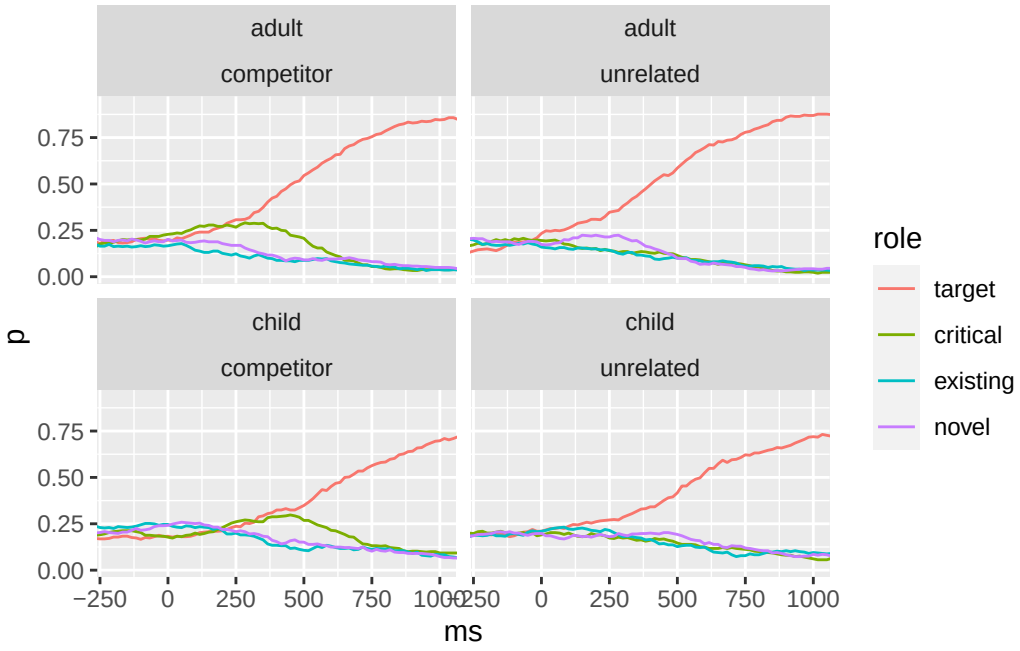
```
count_exist <- trial_cond %>%
  filter(ctype == "exist") %>%
  inner_join(pog_cts, c("sub_id", "t_id")) %>%
  count(group, crit, f_c, role, name = "Y", .drop = FALSE) %>%
  mutate(ms = 1000 * f_c / 60)
```

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`.

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

Now we are ready to plot.

```
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))
```


adult
competitor
adult
unrelated
child
competitor
child
unrelated
p
0.75
0.50
0.25
0.00
−250
0
250
500
750
1000
ms
role
target
critical
existing
novel

References

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