Spanish Vocabulary Test Developement: Portfolio

Glen Wallace

6/19/2020

Background

This workbook covers the Rasch analysis and final form construction phases a project that I am working on. The instrument is a Spanish vocabulary test of multiple choice items intended to serve as a placement test (or component of a placement test) for Spanish departments and as a level test for classifying the ability in Spanish of students who participate in second language acquisition (SLA) studies.

Vocabulary tests have a long history in SLA going back at least to Nation (1990) and have been shown to correlate very well with other more complex instruments. However, the vast majority of existing research has focused on English, creating a research gap for other languages, such as Spanish. The study uses a non-equivalent groups anchor test (NEAT) equating design, were data was collected via multiple forms that shared some items (the anchors) but also had form specific items. The constraints for designing these forms were as follows: the test had to be shorter than 150 items to reduce the risk of fatigue, and the words tested needed to be drawn uniformly from the 3000 most common words in Spanish. Previous research shows that the 3000 most common words in a language account for 95% of all words used in native speech, and is consequently the primary coverage goal for second language instruction. Furthermore the frequency of the word being tested has been shown to impact the difficulty of the item, meaning that words from different frequency levels (or bands) are required to assess students at different levels. Following convention, we defined each frequency band as having 250 words, so we have items from 12 frequency bands (12*250 = 3000).

The end result was three sets of questions: anchors, form 1 specific, and form 2 specific. Each set of questions has the following characteristics: 72 total items, with 6 items from each of the 12 frequency bands. Students took the exam online on a platform that randomized both item order and response option order per student. Following established best practices, each item included the response option "I don't know" which aims to reducing false positives from guessing. Upon completion of the test, participants received extra credit for the Spanish class they were enrolled in.

The effect of frequency is evaluated in a separate file that uses an explanatory model fit in lme4. The data cleaning and unidimensionality parts of this project are also addressed in separate files which are available upon request.

The packages used in this project are: tidyverse, mirt, and WrightMap.

Data Prep

The .csv produced during data cleaning is in long form (for the explanatory model), so we will need to pivot it in order to fit models in mirt. Furthermore, mirt can handle NAs in the data, but not when calculating INFIT and OUTFIT. Because we want to trim items based on these fit statistics, we need to make wide form data frames for each form. Note that there is a column, "freq" (i.e., the frequency of the target word), that is not used in this part of the analysis. As mentioned above, I included it for a separate explanatory model that uses the same data. (As expected, the effect of log word frequency on item difficulty is positive and statistically significant.) The "heritage" column is also not used here but is used below for visualizations.

```
df_long <- read_csv( "../../data/processed_data/jun_resp_freq_long.csv" )
```

```
## Parsed with column specification:
## cols(
##   ID = col_double(),
##   heritage = col_double(),
##   item = col_character(),
##   freq = col_double(),
##   resp = col_double()
## )
```

```
str(df_long)
```

```
## tibble [124,272 x 5] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
##  $ ID      : num [1:124272] 9.15e+08 9.15e+08 9.15e+08 9.15e+08 9.15e+08 ...
##  $ heritage: num [1:124272] 0 0 0 0 0 0 0 0 0 0 ...
##  $ item    : chr [1:124272] "W3001" "W3002" "W3003" "W3004" ...
##  $ freq    : num [1:124272] 118 225 202 152 232 157 80 101 181 175 ...
##  $ resp    : num [1:124272] 1 1 1 1 1 1 1 0 1 1 ...
##  - attr(*, "spec")=
##   .. cols(
##   ..   ID = col_double(),
##   ..   heritage = col_double(),
##   ..   item = col_character(),
##   ..   freq = col_double(),
##   ..   resp = col_double()
##   .. )
```

We need to make three wide form data frames, so it's worth writing a utility function. Note that it is hard coded to this data set for simplicity. The form_code key is as follows: "W1" = form 1, "W2" = form 2, "W3" = anchors. These codes came from the raw data.

```
mk_wide <- function(df, form_code) {
  out <- df %>%
    filter(substr(item, 1, 2) == form_code) %>%
    select(ID, item, resp) %>%
    pivot_wider(names_from=item, values_from=resp) %>%
    select(-ID)

  out
}
```

Just to be safe, it's worth checking the output before proceeding. Usually I would print the whole data frame and explore it, but it doesn't print well with this many columns so here I use `dim()` instead.

```
df_f1_wide <-mk_wide(df_long, "W1")
df_f2_wide <-mk_wide(df_long, "W2")
df_anchors_wide <-mk_wide(df_long, "W3")

dim(df_f1_wide)
```

```
## [1] 419  72
```

```
dim(df_f2_wide)
```

```
## [1] 444  72
```

```r
dim(df_anchors_wide)
```

```
## [1] 863  72
```

Determine fit statistics

Now that we have the data, we need to fit separate models to check for item fit. According to standard procedure, any items that have INFIT or OUTFIT values < 0.5 or > 1.5 will be removed. However, every time an item is removed, the fit of every other item changes (at least a little) so this needs to be done iteratively. We'll automate this with another utility function.

```r
remove_worst <- function(df, tolerance = 0.5, ...) {
  ## This function takes in a wide form response matrix and removes the worst item by
  ## INFIT or OUTFIT.
  mirt_model_specification <- paste("F = 1 -", ncol(df) )
  fit <-  mirt(data=df, model = mirt_model_specification, itemtype = "Rasch", SE = TRUE)
  in_out <- itemfit(fit, fit_stats = c( "infit" ) )

  ## For each model, we need to find the most extreme INFIT/OUTFIT value
  high_in <- max(in_out$infit)
  high_out <- max(in_out$outfit)
  low_in <- min(in_out$infit)
  low_out <- min(in_out$outfit)
  magnitudes_in_out <- c( abs(high_in -1),
                          abs(high_out - 1),
                          abs(low_in - 1),
                          abs(low_out - 1) )

  ## We need to return this model if the tolerance is met. The finished flag is used
  ## in another function.
  if (max(magnitudes_in_out) < tolerance) {
    return(list( trimmed_df = df, finished="yes", message = paste(
      "The accompanying dataframe comtains only items that are within the infit/outfit
      tolerance specified in the function call:\t", tolerance)))
  }

  ## If the tolerance isn't met, we should gather some metadata. This will be processed
  ## in another function.
  drop_lst <- list(item=NULL, value=NULL, type=NULL)

  if(abs(high_in - 1) == max(magnitudes_in_out)) {
    drop_lst$value <- high_in
    drop_lst$type <- "infit"
  } else if (abs(high_out - 1) == max(magnitudes_in_out)) {
    drop_lst$value <- high_out
    drop_lst$type <- "outfit"
  } else if (abs(low_in - 1) == max(magnitudes_in_out)) {
    drop_lst$value <- low_in
    drop_lst$type <- "infit"
  } else if (abs(low_out - 1) == max(magnitudes_in_out)) {
    drop_lst$value <- low_out
    drop_lst$type <- "outfit"
  }
```

```
  if (drop_lst$type == "infit") {
    tmp_item_df <- in_out %>% filter(infit == drop_lst$value)
    drop_lst$item <- as.character(tmp_item_df[1,1])
  } else if (drop_lst$type == "outfit") {
    tmp_item_df <- in_out %>% filter(outfit == drop_lst$value)
    drop_lst$item <- as.character(tmp_item_df[1,1])
  }

  ## In order to call this iteratively, we will need to trim the input
  out <- df %>% select(- drop_lst$item)
  list(trimmed_df = out, finished = "no", item_stats = drop_lst)
}
```

Now that we can remove the worst item, we need another function to recursively call `remove_worst()` and collect the metadata.

```
trim_data <- function( df, tolerance=0.5 ) {
  ## Call remove_worst()
  current_iteration_lst <- remove_worst(df=df, tolerance = tolerance)

  ## Break if nothing needs to be trimmed
  if(current_iteration_lst$finished == "yes") {
    stop("This was a clean dataset: no items needed to be dropped!\n\n
         Best error ever!!!!")
  }

  ## Set up list for building output
  out_lst <- list(data = current_iteration_lst$trimmed_df,
                  removed_items_df = as_tibble(current_iteration_lst$item_stats))

  ## Iterate until tolerance is met while updating output metadata
  c <- 2  # The first model was fit above
  while (current_iteration_lst$finished == "no") {
    print( paste("Fitting model number:", c) )
    c <-  c+1
    current_iteration_lst <- remove_worst(df=current_iteration_lst$trimmed_df,
                                          tolerance = tolerance)
    if (current_iteration_lst$finished == "yes") {
      return(out_lst)
    }
    out_lst$data <- current_iteration_lst$trimmed_df
    out_lst$removed_items_df <- bind_rows(out_lst$removed_items_df,
                                          as_tibble(current_iteration_lst$item_stats))
  }
}
```

The output of this call is long and not very informative, so I've suppressed it. The object it returns is a list with the trimmed data and a data frame showing each item that was removed, the type of fit statistic (INFIT or OUTFIT) that was problematic, and the value of that statistic. An example of the head of the of these data frames is given below. Because these calls work iteratively though many Rasch models, they take a while to run, so when putting the workbook together I saved them as .rds objects to eliminate redundant calculation. I use the `if (FALSE)` conditional throughout this project for blocks that generate objects I save to disc. This format is better than commenting out the section because it works better for debugging and metaprogramming.

```r
if (FALSE) {
  trimmed_f1_lst <- trim_data(df_f1_wide, tolerance = 0.5)
  write_rds(trimmed_f1_lst, "../mods/trimmed_f1_lst.rds")
}
trimmed_f1_lst <- read_rds("../mods/trimmed_f1_lst.rds")

if (FALSE) {
  trimmed_f2_lst <- trim_data(df_f2_wide, tolerance = 0.5)
  write_rds(trimmed_f2_lst, "../mods/trimmed_f2_lst.rds")
}
trimmed_f2_lst <- read_rds("../mods/trimmed_f2_lst.rds")

if (FALSE) {
  trimmed_anchors_lst <- trim_data(df_f2_wide, tolerance = 0.5)
  write_rds(trimmed_anchors_lst, "../mods/trimmed_anchors_lst.rds")
}
trimmed_anchors_lst <- read_rds("../mods/trimmed_anchors_lst.rds")
```

Here's the example mentioned above.

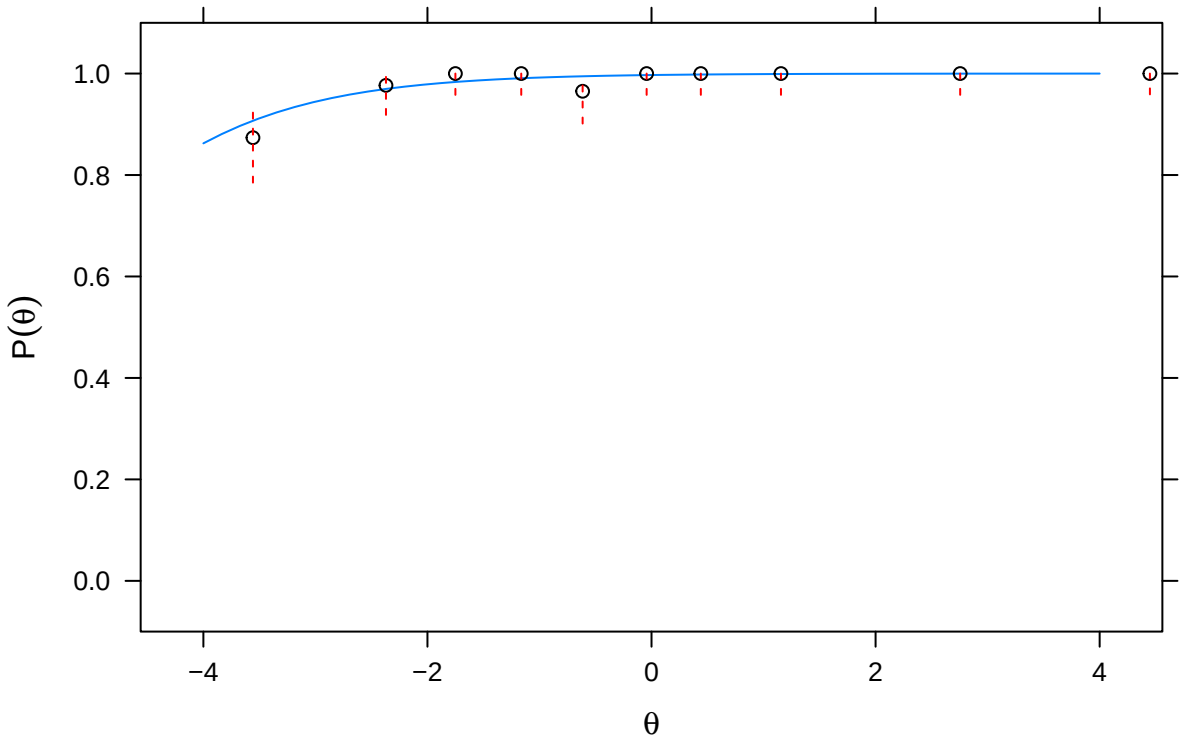
```r
head(trimmed_anchors_lst$removed_items_df)
```

```
## # A tibble: 6 x 3
##   item  value type
##   <chr> <dbl> <chr>
## 1 W3122  2.63 outfit
## 2 W3125  2.39 outfit
## 3 W3015  2.32 outfit
## 4 W3049  1.90 outfit
## 5 W3137  1.97 outfit
## 6 W3018  1.80 outfit
```

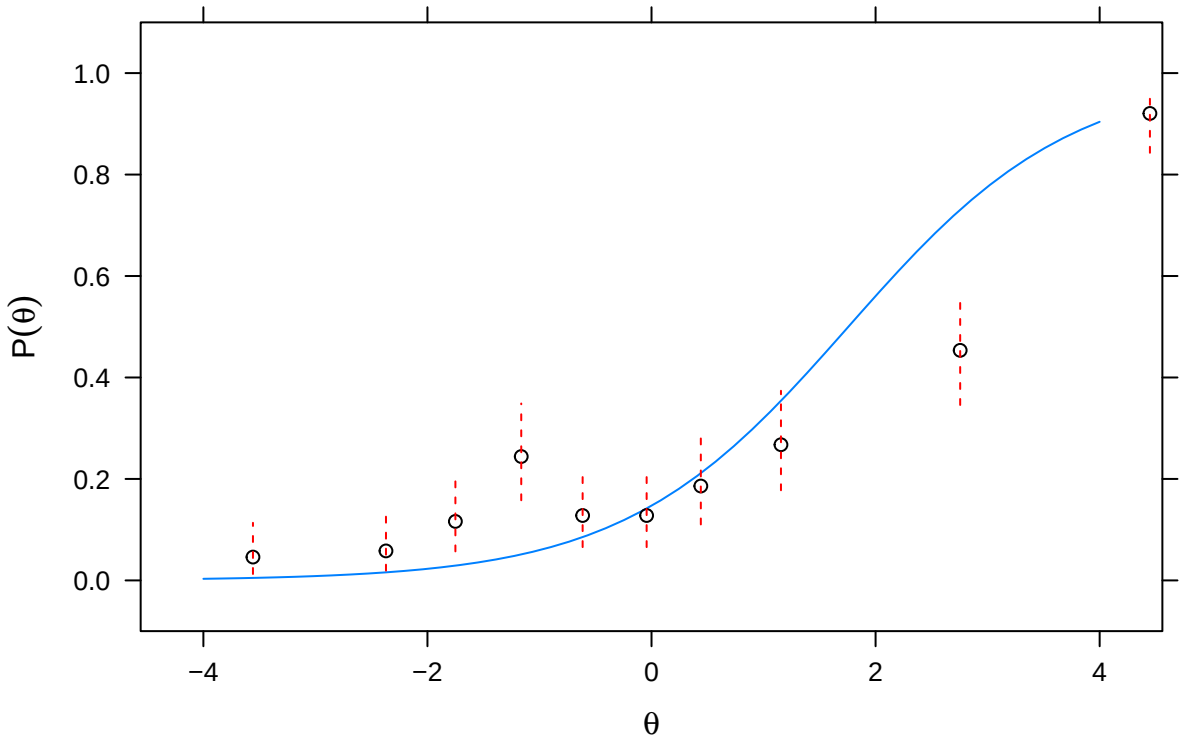
As you can see, there are some VERY bad fit values. To keep things clean I don't do a full investigation here, but basically there are two reasons for this: 1) there are ceiling and floor effects, and 2) some items just have very strange behavior. Here's an example of each. The first is item W3041, the second is W3122.

```r
bad_fit_anchors <-  mirt(data=df_anchors_wide, model = "F = 1 - 72",
                         itemtype = "Rasch", SE = TRUE)
```

```r
itemfit(bad_fit_anchors, empirical.plot = 23)
```




```
itemfit(bad_fit_anchors, empirical.plot = 62)
```



Construct trimmed dataset

Because we have three different trimmed data frames, there are a couple of ways to get the dataset that we want to use. Here, I make a character vector of all the items that need to be dropped, and then use that to subset the original long form data frame. Because mirt needs wide form data, this seems a bit circuitous, but it is helpful because I also need to fit an explanatory model which requires long form data, so I just subset the long form data, export that data frame, and then pivot to a wide dataframe.

```
items_to_drop_vec <- c(trimmed_f1_lst$removed_items_df$item,
                       trimmed_f2_lst$removed_items_df$item,
                       trimmed_anchors_lst$removed_items_df$item
                       )

df_goodfit_long <- df_long %>%
  filter(!(item %in% items_to_drop_vec))

df_goodfit_long <- read_csv("../../data/processed_data/jun_goodfit_long.csv")
```

```
## Parsed with column specification:
## cols(
##   ID = col_double(),
##   heritage = col_double(),
##   item = col_character(),
##   freq = col_double(),
##   resp = col_double()
## )
```

```
df_goodfit_wide <- df_goodfit_long %>%
  select(ID, item, resp) %>%
  pivot_wider(names_from=item, values_from=resp) %>%
  select(-ID)
```

Fit the final model

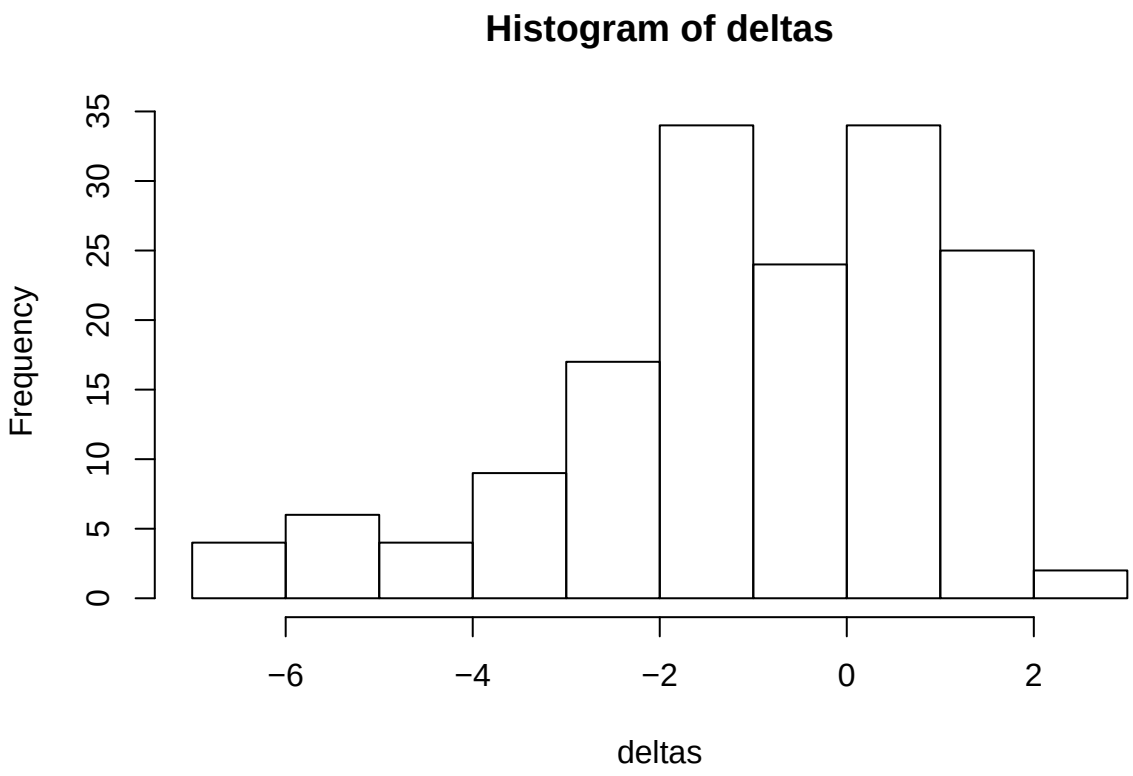
Now that we have only the items with good fit, we'll fit the final model. Because this is the model that will be used in a peer reviewed article, I save the object and wrap the calls in `if (FALSE)` statements to make sure that I don't overwrite the object. The estimates in mirt are stable, but it's better safe than sorry.

```
if (FALSE) {
  m_goodfit_mirt <- mirt(data=df_goodfit_wide, model = "F = 1 - 159",
                         itemtype = "Rasch", SE = TRUE)
  write_rds(m_goodfit_mirt, "../mods/mirt_rasch_goodfit.rds")
}
m_goodfit_mirt <- read_rds("../mods/mirt_rasch_goodfit.rds")
```

Deltas

With the model fit, we can proceed to extract the deltas. We'll also take a peek at the point estimates for the deltas with a histogram.

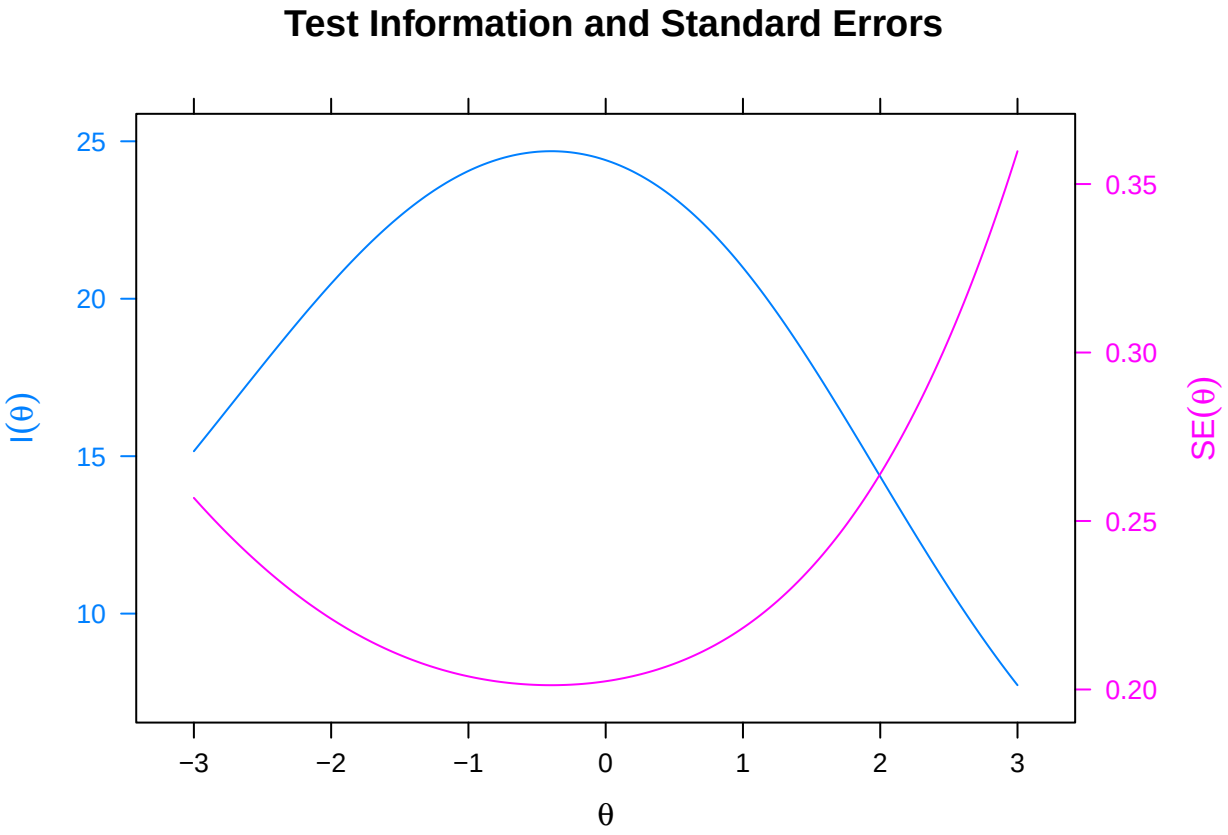
```
df_pars_goodfit_mirt <- coef(m_goodfit_mirt, IRTpars = TRUE, simplify = TRUE)
deltas <- df_pars_goodfit_mirt$items[,2]  # we only need the point estimates for now
hist(deltas)
```



TIC

Using the deltas we can now calculate the TIC and SEM for the test. This will not be definitive because we have not constructed the final form of the test, but it will give a good approximation. To interpret the graph, it is helpful to know that the `mirt()` function sets 0 as the mean of the prior for the MAP estimate of the the person ability distribution.

```
plot(m_goodfit_mirt, type = "infoSE", theta_lim = c(-3, 3))
```

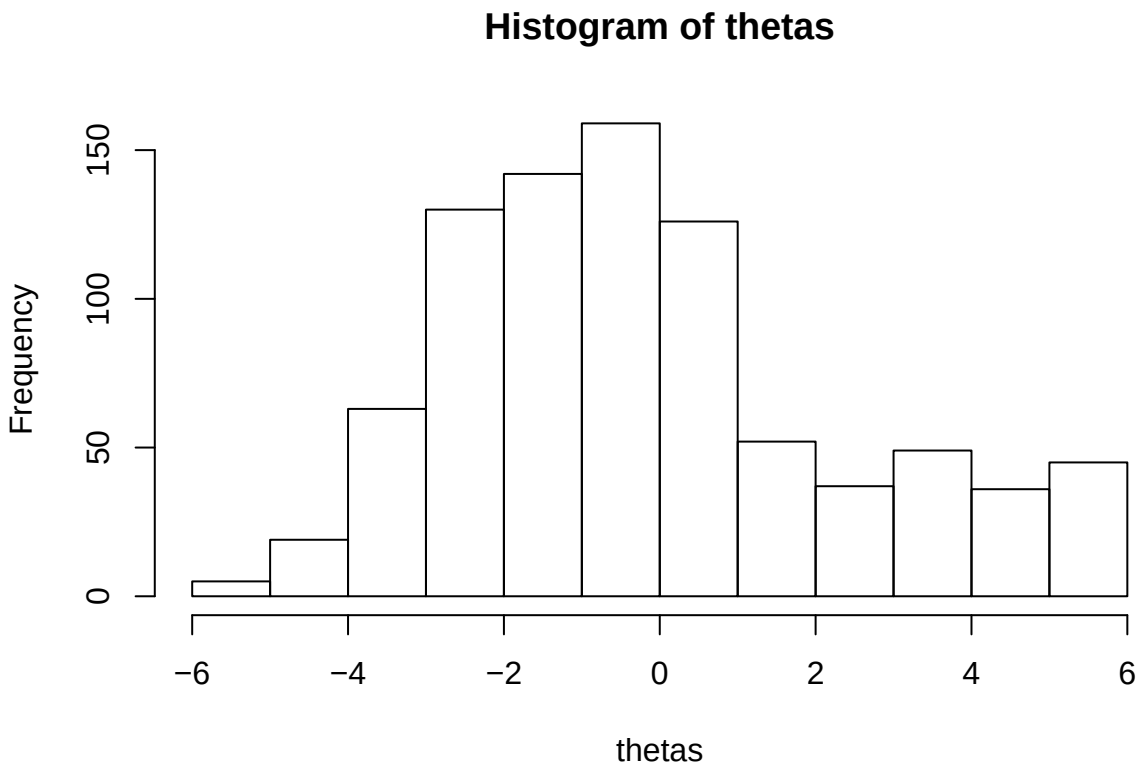


So we're doing pretty well for most students, with the most information available for students slightly below the mean.

thetas

`mirt()` calculates the model using a factor analysis estimation method (see the documention with `?mirt`), so a separate call to `fscores` is needed to estimate the thetas.

```
# returns a matrix with just thetas and SEs
thetas_goodfit_mirt <- fscores(m_goodfit_mirt, method = "MAP",
                               full.scores = TRUE, full.scores.SE = TRUE)
thetas <- thetas_goodfit_mirt[,1]
hist(thetas)
```




```
mean(thetas)
```

```
## [1] -0.1534142
```

Attach demos to thetas

Fortunately `fscroes` can return the thetas as an appended column to the input data frame. Unfortunately, the rows of this data frame get reoredered in the process so we need to do some data gymnastics to realign the rows. There are also some complications about converting between matrices and data frames.

The first step is getting the thetas (f scores) attached to the data matrix. We see here that the Rasch reliablity coeficient is very high.

```
# Returns a matrix
tmp_mtx <- fscores(m_goodfit_mirt, method = "MAP",
                   full.scores = FALSE, full.scores.SE = FALSE)
```

```
##
## Method:  MAP
##
## Empirical Reliability:
##
##      F
## 0.9743
```

```
cat("\n\n") # makes output clearer
```

```
pattern_ordered_mtx <- tmp_mtx[, order(colnames(tmp_mtx))]
colnames(pattern_ordered_mtx)[1:10]
```

```
##  [1] "F"     "SE_F"  "W1007" "W1009" "W1010" "W1011" "W1012" "W1019" "W1020"
## [10] "W1021"
```

Now we'll create a wide form data matrix that contains student IDs. The item columns are in the same order (as shown below) but the leading columns create a mismatch between the column names.

```
df_goodfit_wide_with_ID <- df_goodfit_long %>%
  select(ID, item, resp) %>%
  pivot_wider(names_from=item, values_from=resp)

mtx_og_ordered <- as.matrix(df_goodfit_wide_with_ID[, order(colnames(df_goodfit_wide_with_ID))])
colnames(mtx_og_ordered)[1:10]
```

```
##  [1] "ID"    "W1007" "W1009" "W1010" "W1011" "W1012" "W1019" "W1020" "W1021"
## [10] "W1022"
```

```
cat("\n\n Are all of the named columns in the same order?\n")
```

```
##
##
##  Are all of the named columns in the same order?
```

```
toString(colnames(mtx_og_ordered[2:160])) == toString(colnames(pattern_ordered_mtx[3:161]))
```

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

However, as mentioned above, even thought the columns are lined up, the rows are not, as demonstrated by looking at a slice of the head of each matrix.

```
pattern_ordered_mtx[1:6,1:10]
```

```
##              F      SE_F W1007 W1009 W1010 W1011 W1012 W1019 W1020 W1021
## [1,] -5.630132 0.5361791    NA    NA    NA    NA    NA    NA    NA    NA
## [2,] -5.104862 0.4903895    NA    NA    NA    NA    NA    NA    NA    NA
## [3,] -3.949290 0.3928107    NA    NA    NA    NA    NA    NA    NA    NA
## [4,] -3.401476 0.3493714    NA    NA    NA    NA    NA    NA    NA    NA
## [5,] -3.169153 0.3326739    NA    NA    NA    NA    NA    NA    NA    NA
## [6,] -5.355820 0.5118177    NA    NA    NA    NA    NA    NA    NA    NA
```

```
mtx_og_ordered[1:6,1:10]
```

```
##             ID W1007 W1009 W1010 W1011 W1012 W1019 W1020 W1021 W1022
## [1,] 914720104     1     1     1     1     1     1     0     1     1
## [2,] 914718255     1     1     1     1     1     1     0     1     1
## [3,] 912165732     1     1     0     1     0     0     0     1     1
## [4,] 913969102     1     1     1     0     1     1     0     1     1
## [5,] 915515826     1     1     1     1     1     1     1     1     1
## [6,] 915096675     1     1     1     1     1     0     0     0     1
```

This creates a headache. Fortunately, we can brute force our way through the problem by creating a pattern matching algorithm that takes advantage of the fact that in a Rasch model, each response pattern has a unique theta estimate. Another trick that gets used is replacing NAs with an arbitrary integer, and then converting each row into a single string. The commented call to `print()` is helpful for debugging.

NOTE: This is not especially fast solution and might need to be revisited if the data were a few orders of magnitude larger. But it's easy to code, easy to understand, and runs in about two minutes with this data, so it seemed like the right decision.

```
if (FALSE) {
  theta_for_ID <- matrix(nrow = nrow(mtx_og_ordered), ncol=2)

  for (j in 1:nrow(pattern_ordered_mtx)) {
    for (i in 1:nrow(mtx_og_ordered)) {
```

```
      row_og <- as.character(mtx_og_ordered[i, 2:160])
      row_og <- replace_na(row_og, "5")
      row_og <- toString(row_og)
      row_pattern <- as.character(pattern_ordered_mtx[j,3:161])
      row_pattern <- replace_na(row_pattern, "5")
      row_pattern <- toString(row_pattern)

      if (row_og == row_pattern) {
        theta_for_ID[i,1] <- mtx_og_ordered[i,1]
        theta_for_ID[i,2] <- pattern_ordered_mtx[j,1]
        #print( paste("Row is: ", i))
      }
    }
  }
  write.csv(theta_for_ID, "../csvs/thetas_and_ids_mirt_trimmed_ds.csv",
            row.names = FALSE)
}

theta_for_ID_df <- read_csv("../csvs/thetas_and_ids_mirt_trimmed_ds.csv")
```

```
## Parsed with column specification:
## cols(
##   V1 = col_double(),
##   V2 = col_double()
## )
```

```
colnames(theta_for_ID_df) <- c("ID", "theta")
head(theta_for_ID_df)
```

```
## # A tibble: 6 x 2
##          ID  theta
##       <dbl>  <dbl>
## 1 914720104 -2.28
## 2 914718255 -0.814
## 3 912165732 -2.95
## 4 913969102 -1.40
## 5 915515826 -1.76
## 6 915096675 -2.28
```

Add heritage speaker variable

With the IDs and theta's paired, it's easy to add the heritage variable back into the data. This will allow for comparisons between native speakers of English and heritage speakers of Spanish. We'll also want to know which school each student attended and which course they were enrolled in, but we'll do that below because it requires the raw data.

```
theta_heritage_df <- df_goodfit_long %>%
  select(ID, heritage) %>%
  unique() %>%
  left_join(theta_for_ID_df, by="ID")
head(theta_heritage_df)
```

```
## # A tibble: 6 x 3
##          ID heritage  theta
##       <dbl>    <dbl>  <dbl>
```

```
## 1 914720104        0 -2.28
## 2 914718255        0 -0.814
## 3 912165732        0 -2.95
## 4 913969102        0 -1.40
## 5 915515826        0 -1.76
## 6 915096675        0 -2.28
```

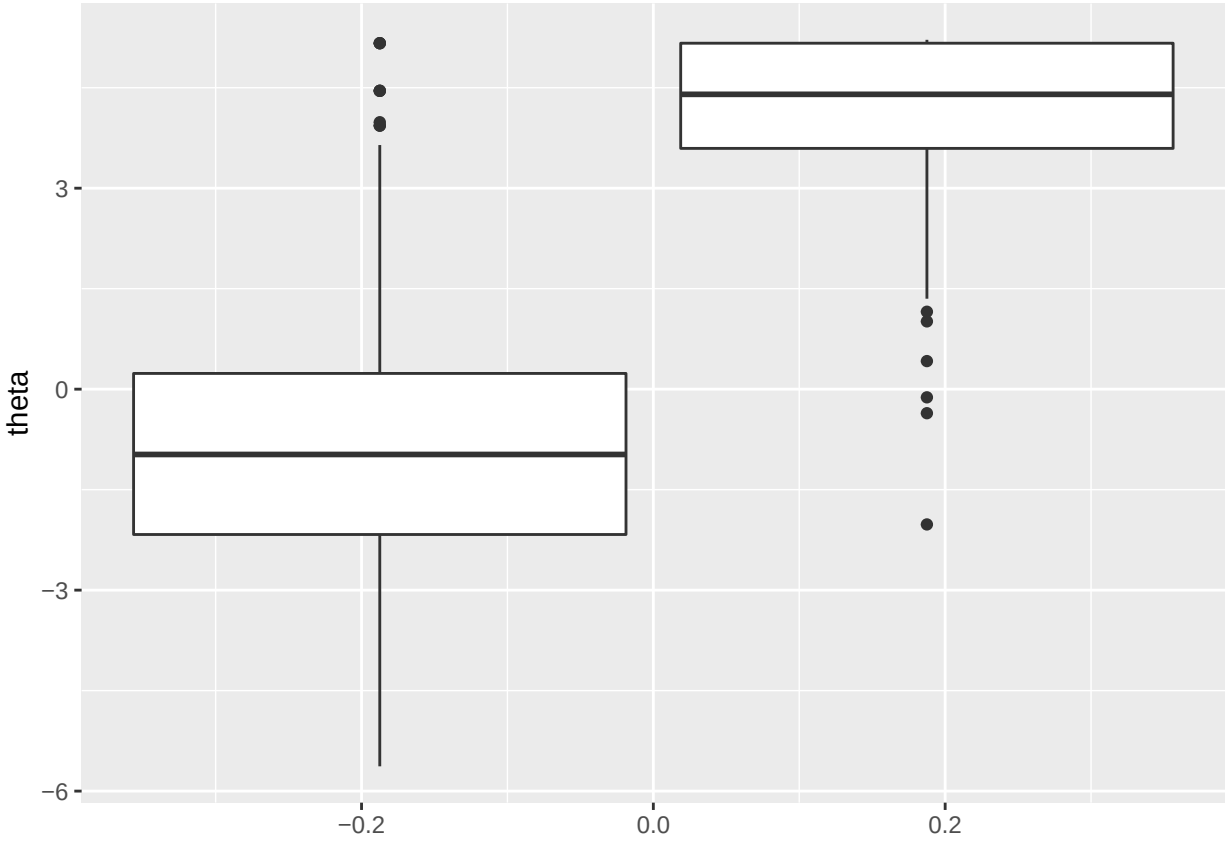
This makes it easy to do things like see what percentage of students are heritage speakers.

```
sum(theta_heritage_df$heritage) / nrow(theta_heritage_df)
```

```
## [1] 0.1564311
```

Or use a boxplot to get a sense of how the abilities of the two groups compare.

```
ggplot(theta_heritage_df, aes(y=theta, group=heritage)) +
  geom_boxplot()
```



Add course and school

Because the course and school columns were trimmed in the original data manipulation, the raw files need to be loaded. These files are pretty messy, so there's some cleaning that need to be done. For starters, the raw files start with two identical rows that name the columns. `read_csv()` can handle the first, but we need to trim the second manually.

One of the columns does code the university, but the labeling is obtuse so it's easier to just make a new column since the names of the files indicate which university they pertain to.

```
ucd_1_sections <- df_ucd_1_wide %>%
  select(ID, SECTION)
ucd_1_sections$school <- "UCD"

ucd_2_sections <- df_ucd_2_wide %>%
  select(ID, SECTION)
ucd_2_sections$school <- "UCD"

ucd_sections <- bind_rows( ucd_1_sections, ucd_2_sections) %>%
  rename(section = SECTION)
```

And then we do exactly the same thing for the other school. Note that the column indicating the course the student took has an arbitrary. Unfortunately this kind of thing is just par for the course.

```
uwl_1_sections <- df_uwl_1_wide %>%
  select(ID, Q1175)
uwl_1_sections$school <- "UWL"

uwl_2_sections <- df_uwl_2_wide %>%
  select(ID, Q1175)
uwl_2_sections$school <- "UWL"

uwl_sections <- bind_rows( uwl_1_sections, uwl_2_sections) %>%
  rename(section = Q1175)
```

And then we combine these data frames to have all data in one object. Also, ID is currently a factor so we'll convert it to numeric.

```
all_sections <- bind_rows(ucd_sections, uwl_sections)
all_sections$ID <- as.numeric(all_sections$ID)
```

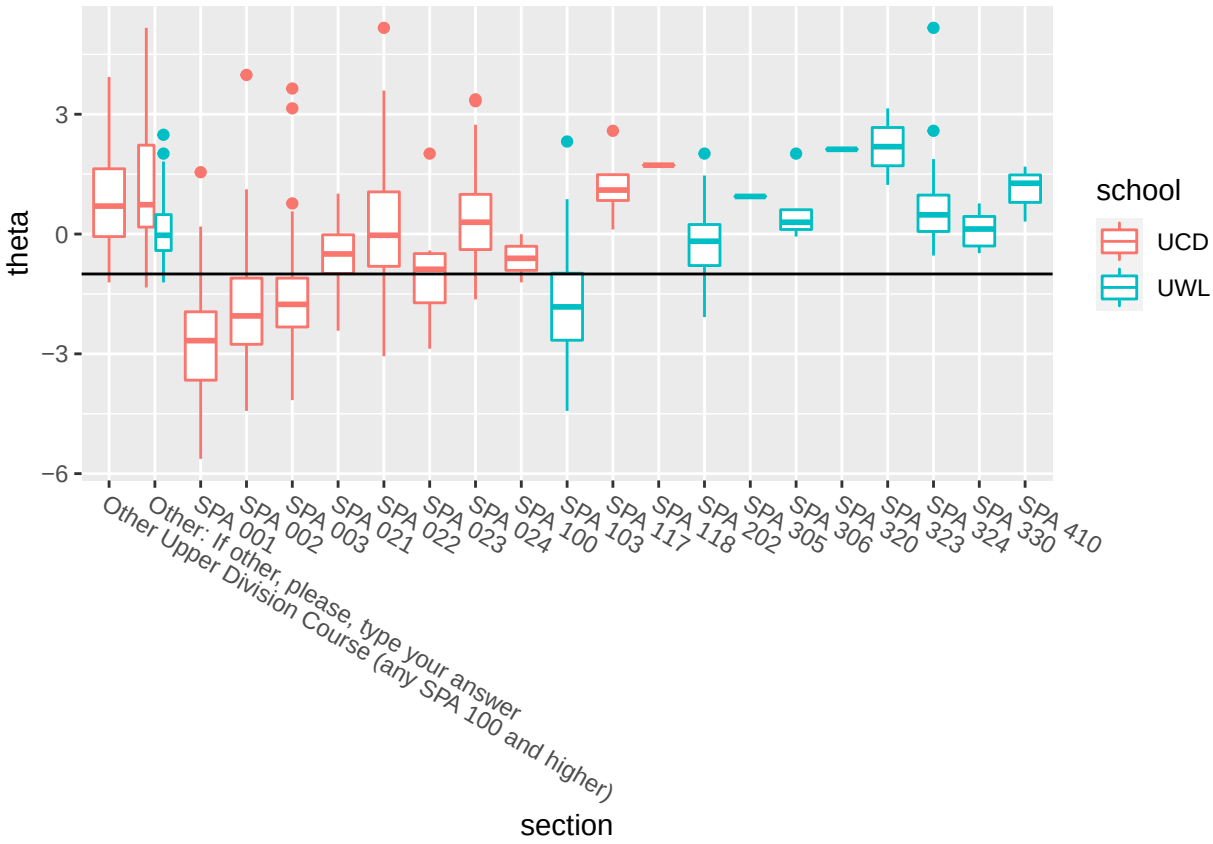
And we perform a join with the data frame that has the thetas.

```
section_theta_heritage_df <- theta_heritage_df %>%
  left_join(all_sections, by="ID")
```

Exploratory analysis

Now we can do some work. The code block below generates a boxplot that allows from some exploratory analysis.

```
section_theta_heritage_df %>%
  filter(heritage == 0) %>%
  ggplot(aes( y=theta, x= section, color=school)) +
  geom_boxplot() +
  geom_hline(yintercept = -1) +
  theme(axis.text.x = element_text( angle = -30, vjust=1, hjust=0))
```



Remember that we want to establish a decision criteria that makes a meaningful classification based on test score. The horizontal line was eyeballed as a possible breakpoint, but it turns out that looking at the data grouped by course isn't the most informative approach. This is in part because the n of some cells is very low, making the summaries provided above less informative than they appear.

```
table(section_theta_heritage_df$section)
```

```
##
## Other Upper Division Course (any SPA 100 and higher)
##                                                   45
##            Other: If other, please, type your answer
##                                                  215
##                                              SPA 001
##                                                  114
##                                              SPA 002
##                                                   86
##                                              SPA 003
##                                                  156
##                                              SPA 021
##                                                   24
##                                              SPA 022
##                                                   42
##                                              SPA 023
##                                                    7
##                                              SPA 024
##                                                   52
##                                              SPA 100
##                                                    4
```

```
##                                                 SPA 103
##                                                      56
##                                                 SPA 117
##                                                      11
##                                                 SPA 118
##                                                       1
##                                                 SPA 149
##                                                       1
##                                                 SPA 202
##                                                      84
##                                                 SPA 305
##                                                       1
##                                                 SPA 306
##                                                       5
##                                                 SPA 320
##                                                       1
##                                                 SPA 323
##                                                       6
##                                                 SPA 324
##                                                      30
##                                                 SPA 330
##                                                       7
##                                                 SPA 410
##                                                       4
```

It turns out to be much better to look at the students grouped by year of instruction rather than course. Fortunately, this classification is straightforward.

```
first_yr <- c( "SPA 001", "SPA 002", "SPA 003", "SPA 103")
second_yr <- c( "SPA 021", "SPA 022", "SPA 023", "SPA 024", "SPA 202" )
# leave third as else

# Also pre-declare container objects before a loop
yr <- vector(mode="character", length=nrow(section_theta_heritage_df))

for (i in seq_along(section_theta_heritage_df$section)) {
  if (section_theta_heritage_df$section[i] %in% first_yr) {
    yr[i] <- "First"
  } else if (section_theta_heritage_df$section[i] %in% second_yr) {
    yr[i] <- "Second"
  } else {
    yr[i] <- "Third"
  }
}

section_theta_heritage_df$year <- yr
```

And the distribution of students across years is not wildly unbalanced.

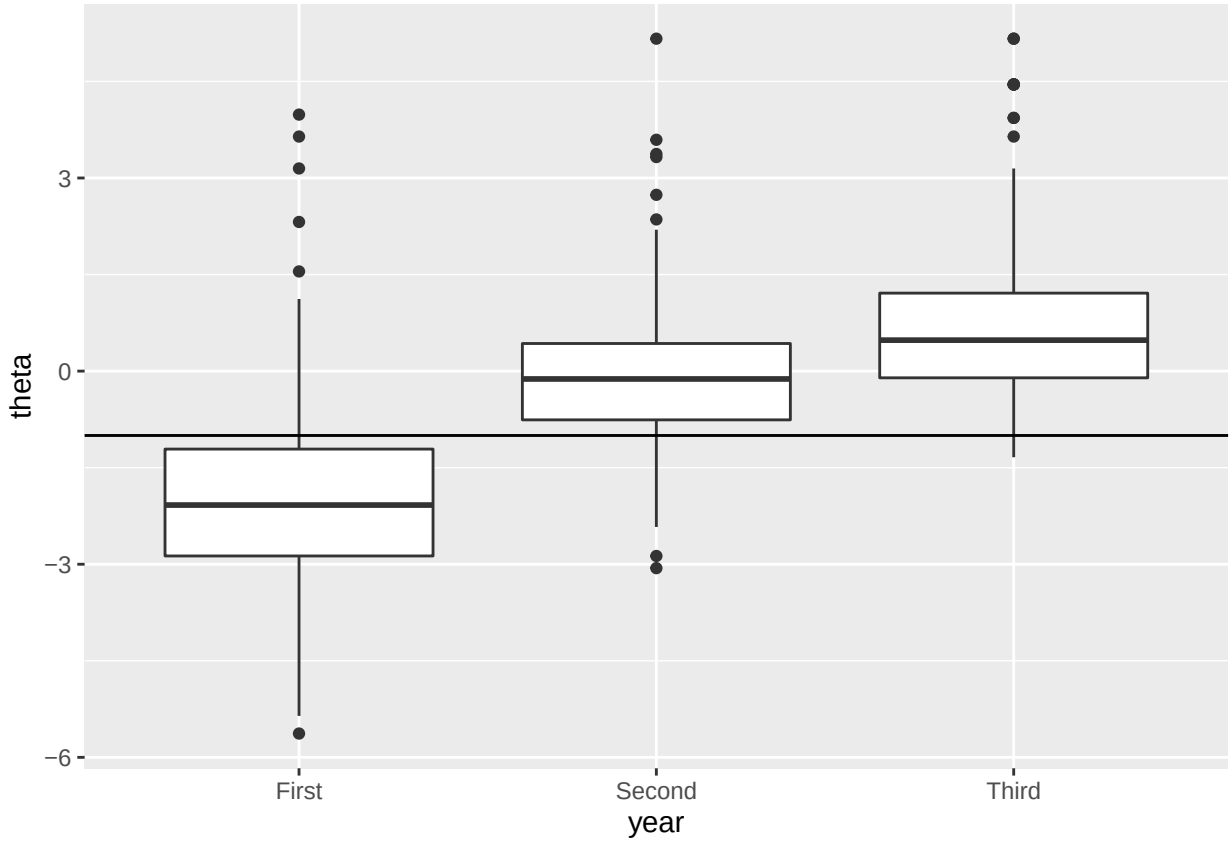
```
table(section_theta_heritage_df$year)
```

```
##
##  First Second  Third
##    412    209    331
```

Using the same approach as above, the boxplot is now much more informative and suggests that classification

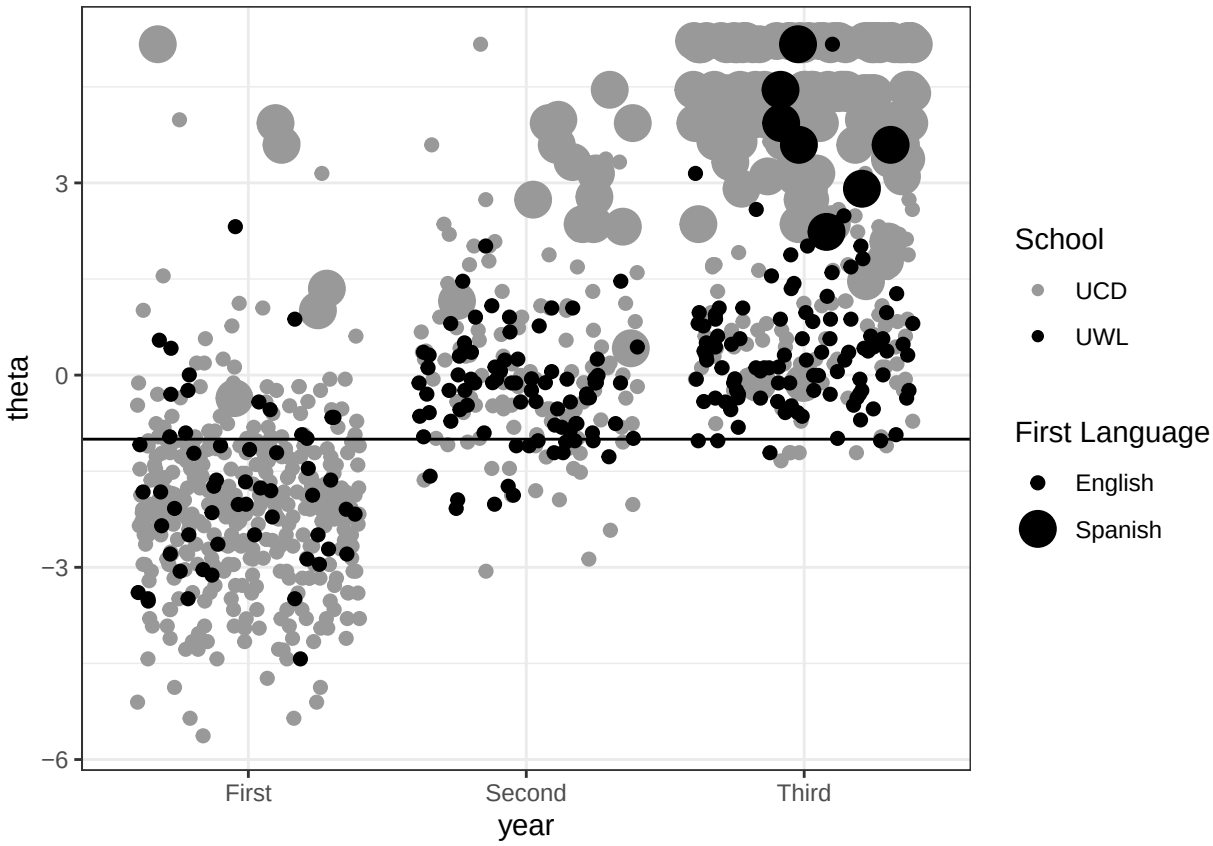
between first-year and not-first-year students should be possible.

```
section_theta_heritage_df %>%
  filter(heritage == 0) %>%
  ggplot(aes( y=theta, x= year)) +
  geom_boxplot() +
  geom_hline(yintercept = -1)
```



With a bit more work, we can see that, by score, almost all of the heritage speakers should be in third year courses. However, they are not, so we'll want to see how their presence affects classification later on.

```
section_theta_heritage_df %>%
  ggplot(aes(y=theta, x= year, color=factor(school), size=(factor(heritage)))) +
  geom_jitter() +
  geom_hline(yintercept = -1) +
  theme_bw() +
  scale_color_manual(name  ="School",
                     breaks=c("UCD", "UWL"),
                     labels=c("UCD", "UWL"),
                     values = c("#999999", "#000000")) +
  scale_size_discrete(name  = "First Language",
                      breaks=c(0, 1),
                      labels=c("English", "Spanish"))
```



Finding the best cutpoint?

Finding the best cutpoint will require another function, which I've called `classify_students()`. It takes a cutoff value and a dataframe with columns ability (theta) and year of instruction, with rows indicating different students. The output is a list with the percent of correct classifications and a numeric vector of 1s and 0s indicating how each row was classified.

```
classify_students <- function(cutoff, theta_yr_df) {
  n_rows <- nrow(theta_yr_df)
  theta <- theta_yr_df$theta
  yr <- theta_yr_df$year
  classifications <- vector(mode="integer", length=n_rows)

  for (i in 1:n_rows) {
    if (theta[i] <= cutoff && yr[i] == "First") {
      classifications[i] <- 1
    } else if (theta[i] >= cutoff && yr[i] != "First"){
      classifications[i] <- 1
    }
  }
  list( perc_correct = sum(classifications) / length(classifications),
        classifications = classifications)
}
```

Here's a simple unit test showing that `classify_students()` works properly. If anyone else were going to use it, I'd sanitize imputs and do more extensive testing, but this works for the current usage.

```
tst_df <- tibble(
  theta = c( -1, -1, -1, 1, 1, 1),
  year = c( "First", "Second", "Third", "First", "Second", "Third" )
)

classify_students(cutoff=0, theta_yr_df = tst_df)
```

```
## $perc_correct
## [1] 0.5
##
## $classifications
## [1] 1 0 0 0 1 1
```

Grid search

There are several ways to establish the cutoff, but the simplest is a grid search over reasonable values based on visual analysis. If this needed to be fully generalizable, I'd implement a binary search, but in this application it isn't necessary. Here's a utility function that will do the job.

```
grid_search_cutoff = function(df) {
  grid_search_df <-  tibble(threshold = seq( from = -1.5, to = 0, length.out = 100))
  perc_classified <- vector(mode="numeric", length=nrow(grid_search_df))

  for (i in 1:nrow(grid_search_df)) {
    perc_classified[i] <- classify_students(grid_search_df$threshold[i],
                                            theta_yr_df = df)$perc_correct
  }

  grid_search_df$perc_classified <- perc_classified
  grid_search_df %>%
    filter( perc_classified == max(perc_classified) ) %>%
    # The line above can give multiple rows with some threshold,
    # so we grab just the top one
    slice(n = 1)
}
```

For the full data, the best threshold is -1.07, which is 87.7% accurate.

```
grid_search_cutoff(section_theta_heritage_df)
```

```
## # A tibble: 1 x 2
##   threshold perc_classified
##       <dbl>           <dbl>
## 1     -1.08           0.877
```

However, the algorithm arguably does a bit better since students at UCD do not take a placement test, meaning that their enrollment is not necessarily a good indicator of external validity. So let's see how it performs on just UWL students.

```
grid_uwl_only_df <- section_theta_heritage_df %>%
  filter(school == "UWL")

grid_search_cutoff(grid_uwl_only_df)
```

```
## # A tibble: 1 x 2
##   threshold perc_classified
##       <dbl>           <dbl>
```

```
## 1     -1.45           0.895
```

That's not much better, but it is very close to 90% accurate. The visual analysis also suggests that more students in first-year courses are misclassified, so let's see what it looks like for only second- and third-year students from both schools.

```
not_first_year_df <- section_theta_heritage_df %>%
  filter(year != "First")

table(classify_students(-1.05, theta_yr_df = not_first_year_df)$classifications)
```

```
##
##   0   1
##  35 505
```

That's even better, showing that classification is globally pretty good (high 80s) and quite good for some groups.

```
505 / (505 + 35)
```

```
## [1] 0.9351852
```

Select items for final form

Now that we've demonstrated the test works for its intended purpose, we need to create the "best" form from the available items. Since we've established a cutoff and are not trying to create parallel forms (we don't have enough items, yet) "best" means using the items with deltas closest to the cutoff as those items will be most informative for the classification. However, my colleagues felt strongly about trying to maintain 12 items per frequency band, so I tried to accommodate that. As will be shown below, it wasn't quite possible, but I got pretty close. The first step is building a new data frame with columns for item names, deltas, the frequency band of the item (which is calculated using a substring of the item name), and the distance of the item's difficulty from the threshold. I was able to argue for removing the first frequency band since (unsurprsingly) all of the items in that band extremely easy and provided very litle information for the population of study. Note that the frequency bands are zero indexed.

```
item_deltas_df <- tibble(item = names(deltas),
                         delta = deltas
                         )
item_deltas_df$item_num  <- as.numeric(substr(item_deltas_df$item, 3, 5))
item_deltas_df$band <- floor((item_deltas_df$item_num -1) / 12)
item_deltas_df$dist_to_threshold <- abs(item_deltas_df$delta + 1)  # threshold of -1
head(item_deltas_df)
```

```
## # A tibble: 6 x 5
##   item  delta item_num  band dist_to_threshold
##   <chr> <dbl>    <dbl> <dbl>             <dbl>
## 1 W3002 -4.41        2     0              3.41
## 2 W3003 -5.67        3     0              4.67
## 3 W3004 -6.18        4     0              5.18
## 4 W3006 -2.24        6     0              1.24
## 5 W1007 -5.49        7     0              4.49
## 6 W1009 -5.05        9     0              4.05
```

With that information, we can write another utility function to build up a new data frame that has only the twelve best items in a given frequency band.

```
grab_top_12_in_band <- function(df, n_band) {
  tmp_df <- item_deltas_df %>%
```

```
  filter(band == n_band) %>%
  arrange(dist_to_threshold)

  if (nrow(tmp_df) > 11) {
    out <- tmp_df$item[1:12]
  } else {
    warning("\nThere were not at leat 12 items remaining in the following band: \n")
    cat( "Band:\t", n_band, "\nRemaining items:\t", nrow(tmp_df), "\n\n" )
    out <- tmp_df$item[1:nrow(tmp_df)]
  }
  out
}
```

And other function to loop over all the bands.

```
grab_all_bands <- function (df) {
  out <- grab_top_12_in_band(df=df, n_band=1)
  for (i in 2:11) {
    out <- c(out, grab_top_12_in_band( df=df, n_band=i))
  }
  out
}
```

Running this we see that there are three bands that don't have the minimum of 12 items after removing items based INFIT and OUTFIT. At the request of my collaborators I filled these gaps with items from neighboring frequency bands rather than with items that would provide the most information for students near the cut score. If there were many more items, it would be worth automating looking at the surrounding bands for each band that is missing, but in this case it's easier to just do it by hand.

```
final_items <- grab_all_bands(item_deltas_df)
```

```
## Warning in grab_top_12_in_band(df = df, n_band = i):
## There were not at leat 12 items remaining in the following band:

## Band:     2
## Remaining items:  11

## Warning in grab_top_12_in_band(df = df, n_band = i):
## There were not at leat 12 items remaining in the following band:

## Band:     10
## Remaining items:  8

## Warning in grab_top_12_in_band(df = df, n_band = i):
## There were not at leat 12 items remaining in the following band:

## Band:     11
## Remaining items:  9
```

```
final_items
```

```
##   [1] "W1020" "W2024" "W2019" "W3013" "W1024" "W1019" "W3016" "W1021" "W3014"
##  [10] "W3017" "W1023" "W2020" "W3030" "W3028" "W2036" "W2033" "W3025" "W3027"
##  [19] "W3029" "W1031" "W1036" "W2035" "W1034" "W3040" "W2044" "W1045" "W1047"
##  [28] "W3037" "W1044" "W2048" "W1046" "W3038" "W3039" "W3041" "W3042" "W3050"
##  [37] "W2059" "W2056" "W1056" "W2055" "W1060" "W2058" "W3052" "W3051" "W3053"
##  [46] "W1055" "W3054" "W1069" "W3064" "W1068" "W3065" "W3063" "W2068" "W1071"
##  [55] "W1072" "W1067" "W3062" "W3061" "W2069" "W2083" "W2081" "W3076" "W3078"
##  [64] "W2079" "W3073" "W1082" "W2080" "W1081" "W2084" "W3077" "W1079" "W1095"
```

```
##  [73] "W2093" "W3090" "W2095" "W3089" "W1096" "W2092" "W3087" "W2091" "W1094"
##  [82] "W1093" "W1092" "W1104" "W1106" "W3102" "W1108" "W2107" "W3101" "W1105"
##  [91] "W3098" "W2104" "W2108" "W3099" "W2103" "W1116" "W1115" "W3109" "W1117"
## [100] "W3111" "W3110" "W2116" "W2117" "W2120" "W2115" "W2119" "W1118" "W3121"
## [109] "W3126" "W2131" "W1130" "W2132" "W1131" "W1129" "W3123" "W2142" "W3135"
## [118] "W3134" "W3138" "W1140" "W3133" "W3136" "W1142" "W1141"
```

To keep this workbook clean I've removed that code because it isn't very informative or interesting. The block below gives an example of how it was implimented using dplyr.

```
item_deltas_df %>%
  filter(band == 8) %>%
  arrange(dist_to_threshold)
```

The results, however, are relevant:

Need 1 for band 2: W2022 from band 1

Need 4 for band 10 and need 3 for band 11, so in total7 from band 9 and below, which gives: 3114 1120 1119 from band 9 3100 1103 1107 2106 from band 8

So we need to add this items to the vector of items names we already have.

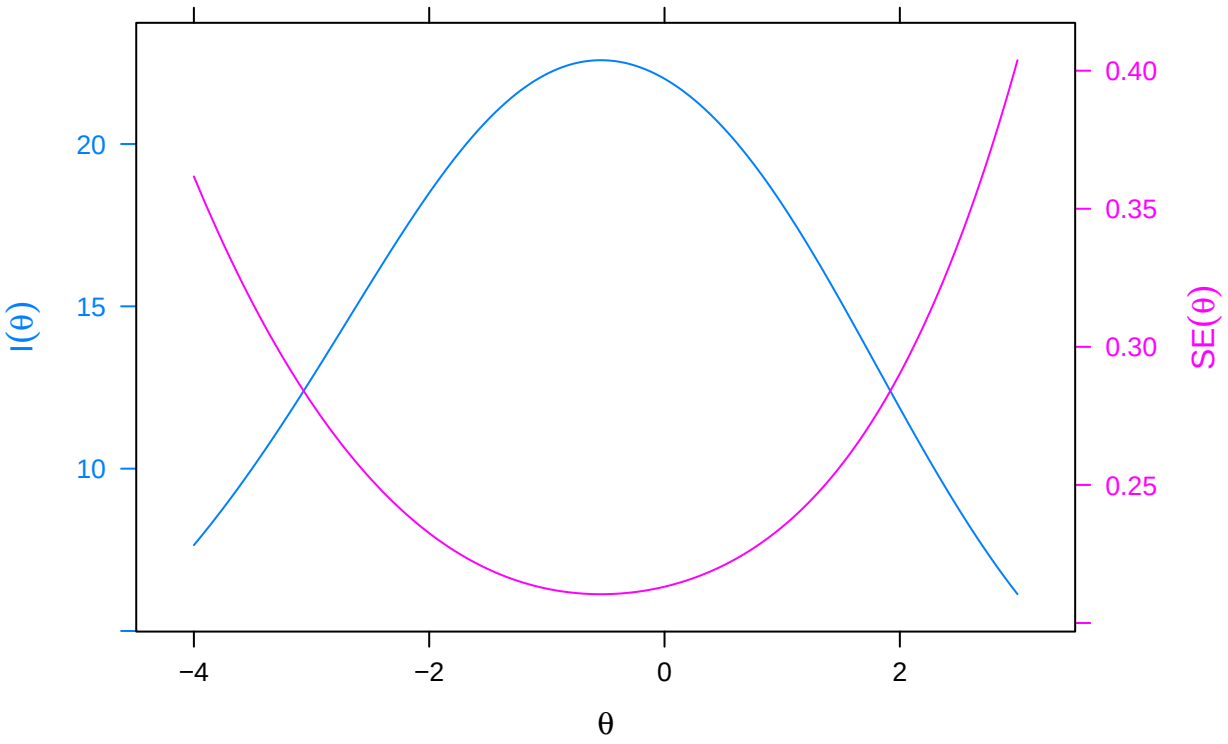
```
final_items <- c(final_items,
                 c("W2022", "W3114", "W1120", "W1119", "W3100", "W1103", "W1107", "W2106"))
```

Visualizations

TIC for final form

With the names of the items that we want to include in hand, we can estimate the performance of the final form by using `which.items` argument of the `plot()` function for mirt objects.

```
item_col_indices <- match(final_items, names(df_goodfit_wide))
plot(m_goodfit_mirt, type = "infoSE", theta_lim = c(-4, 3),
     which.items=item_col_indices)
```



And we should recalculate the reliability coefficient, since it must be lower because we removed items. As you can see, it's still very good.

```
# We just want the side effects of this call, not the object it returns,
# so we'll assign it to a dummy variable
dummy <- df_goodfit_wide %>%
  select(final_items) %>%
  mirt(model = "F = 1 - 132", itemtype = "Rasch", SE = TRUE) %>%
  fscores(method = "MAP", full.scores = FALSE, full.scores.SE = FALSE)
```

```
## Note: Using an external vector in selections is ambiguous.
## i Use `all_of(final_items)` instead of `final_items` to silence this message.
## i See <https://tidyselect.r-lib.org/reference/faq-external-vector.html>.
## This message is displayed once per session.
```

```
## Iteration: 1, Log-Lik: -35410.420, Max-Change: 2.69883Iteration: 2, Log-Lik: -30457.117, Max-Change:
##
## Calculating information matrix...
##
## Method:  MAP
##
## Empirical Reliability:
##
##      F
## 0.9689
```

We can even estimate the raw score that corresponds to our cutoff value, or other values that might be of interest.

```r
theta <- matrix(seq(-3, 3))
cat("The raw score that corresponds to the cutoff is:\n")
```

```
## The raw score that corresponds to the cutoff is:
```

```r
expected.test( m_goodfit_mirt, Theta=matrix(-1), which.items = item_col_indices )
```

```
## [1] 58.23349
```

```r
cat("\nAnd the raw scors for integer theta names from -3 to 3 are:\n")
```

```
##
## And the raw scors for integer theta names from -3 to 3 are:
```

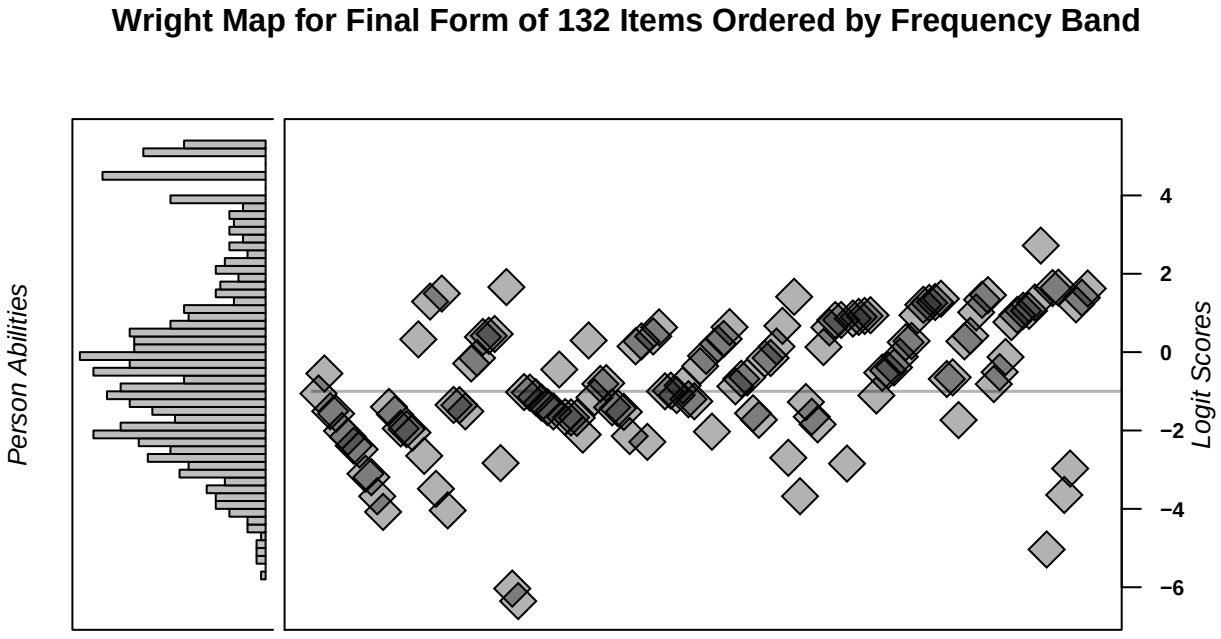
```r
expected.test( m_goodfit_mirt, theta, which.items = item_col_indices )
```

```
## [1]  21.94222  37.62498  58.23349  80.64734 101.00698 116.09830 124.93640
```

Wright map of final form

We can also make a Wright map using the WrightMap package. If more custonization is required, I could also build this is ggplot, but `wrightMap()` gives a good start.

```r
wrightMap( thetas=thetas_goodfit_mirt[,1], thresholds = deltas[final_items],
           # Titles and axes
           main.title = 'Wright Map for Final Form of 132 Items Ordered by Frequency Band',
           axis.persons = 'Person Abilities',
           axis.items = "",
           axis.logits = 'Logit Scores',
           axis.persons.cex = 3,

           # Histogram
           breaks = 40,
           dim.names = '',   # Since there is only one dim, this reduces clutter
           dim.color = 8,

           # Symbols for items
           thr.sym.cex = 2,
           thr.sym.col.fg = 1,
           show.thr.lab = FALSE,

           # Suppress item labels because there are way to many to print right
           label.items = "",
           label.items.ticks = FALSE,

           # Add line at cutpoint
           cutpoints = -1,

           # Misc.
           return.thresholds = FALSE  # Reduces clutter in terminal
)
```



And that's that. Hope you enjoyed the worked example!