Assignment 10

Instructions: Make an R Notebook and in it answer the question or questions below. When you are done, hand in on Quercus the *output* from Previewing (or Knitting) your Notebook, probably an `html` or `pdf` file. An `html` file is easier for the grader to deal with. Do *not* hand in the Notebook itself. You want to show that you can (i) write code that will answer the questions, (ii) run that code and get some sensible output, (iii) write some words that show you know what is going on and that reflect your conclusions about the data. Your goal is to convince the grader that you *understand* what you are doing: not only doing the right thing, but making it clear that you know *why* it's the right thing.

Do *not* expect to get help on this assignment. The purpose of the assignments is for you to see how much *you* have understood. You will find that you also learn something from grappling with the assignments. The time to get help is after you watch the lectures and work through the problems from PASIAS, via tutorial and the discussion board, that is *before* you start work on the assignment. The only reasons to contact the instructor while working on an assignment are to report (i) something missing like a data file that cannot possibly be read, (ii) something *beyond your control* that makes it impossible to finish the assignment in time after you have started it.

There is a time limit on this assignment (you will see Quercus counting down the time remaining).

1. The file in http://ritsokiguess.site/STAD29/bfi.csv contains responses by 2436 individuals to a personality questionnaire. There are 25 items on the questionnaire, labelled by a letter and a number (thus in an item like `O3` the O is a letter). Each response is on a scale from 1 to 6, 1 being Very Inaccurate (as a description of the respondent), and 6 being Very Accurate. Descriptions of the items are in http://ritsokiguess.site/STAD29/bfi.txt. (You can ignore the codes like `q_146` in that file.) We will be carrying out a factor analysis, and hoping to see that the items featuring in each factor have something in common that we can talk about.

   (a) Read in the data and display some of it.

Solution:

The usual:

```
my_url <- "bfi.csv"
bfi <- read_csv(my_url)
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   .default = col_double()
## )
## i Use `spec()` for the full column specifications.
```

```
bfi
```

```
## # A tibble: 2,436 x 26
##       id    A1    A2    A3    A4    A5    C1    C2    C3    C4    C5    E1    E2
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
##  1 61617     2     4     3     4     4     2     3     3     4     4     3     3
```

```
##  2 61618     2     4     5     2     5     5     4     4     3     4     1     1
##  3 61620     5     4     5     4     4     4     5     4     2     5     2     4
##  4 61621     4     4     6     5     5     4     4     3     5     5     5     3
##  5 61622     2     3     3     4     5     4     4     5     3     2     2     2
##  6 61623     6     6     5     6     5     6     6     6     1     3     2     1
##  7 61624     2     5     5     3     5     5     4     4     2     3     4     3
##  8 61629     4     3     1     5     1     3     2     4     2     4     3     6
##  9 61633     2     5     6     6     5     6     5     6     2     1     2     2
## 10 61634     4     4     5     6     5     4     3     5     3     2     1     3
## # ... with 2,426 more rows, and 13 more variables: E3 <dbl>, E4 <dbl>,
## #   E5 <dbl>, N1 <dbl>, N2 <dbl>, N3 <dbl>, N4 <dbl>, N5 <dbl>, O1 <dbl>,
## #   O2 <dbl>, O3 <dbl>, O4 <dbl>, O5 <dbl>
```

There are (correctly) 2436 rows, one per person, and there are (correctly) 26 columns, one for each item on the questionnaire and one for the IDs. The ID is a number, but it has no meaning as a number.

Extra: the data actually come from the `psych` package, the data set called `bfi`. The data set there has three extra columns, gender, education and age, which I removed, since they would have no impact on our analysis. If you have taken psychology, you might be suspecting that the initials BFI and the initial letters of the column names have something to do with the Big Five personality traits, and you would be right. This is not actually an official "Big Five Inventory"; the items came from another place called the International Personality Item Pool, but were chosen to match up with the Big Five peronality traits. We'll see how well they did.

I also got rid of some missing values for you.

(b) Make a scree plot. This will mean running a suitable principal components analysis first.

Solution:

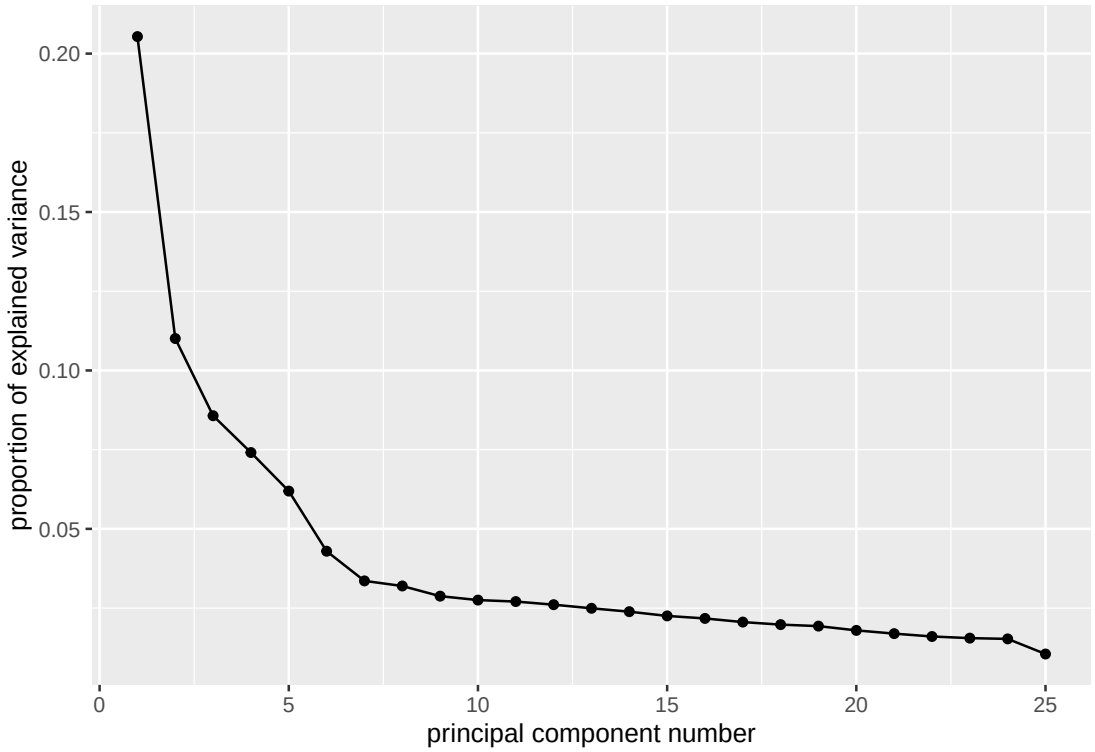
The first thing is to get rid of the column of IDs:

```
bfi %>% select(-id) -> bfi_numbers
bfi.1 <- princomp(bfi_numbers, cor = TRUE)
```

Note that selecting only the numeric columns via something like `select(where(is.numeric))` *will not work* here, because the IDs are numbers as well (as far as R is concerned, even though in your head they're only labels).

The easiest way to get a scree plot is to use `ggscreeplot`.[1] Feed this the output from `princomp`:

```
ggscreeplot(bfi.1)
```



If you don't like that, you'll need to grab the eigenvalues from the `princomp` output:

```
names(bfi.1)
```

```
## [1] "sdev"     "loadings" "center"   "scale"    "n.obs"    "scores"   "call"
```

So it's `sdev` (for "standard deviations") that we need:

```
bfi.1$sdev
```

```
##    Comp.1    Comp.2    Comp.3    Comp.4    Comp.5    Comp.6    Comp.7    Comp.8
## 2.2659018 1.6588811 1.4637971 1.3610024 1.2442519 1.0361382 0.9162636 0.8939833
##    Comp.9   Comp.10   Comp.11   Comp.12   Comp.13   Comp.14   Comp.15   Comp.16
## 0.8479323 0.8295112 0.8224192 0.8073412 0.7894637 0.7723748 0.7503938 0.7370925
##   Comp.17   Comp.18   Comp.19   Comp.20   Comp.21   Comp.22   Comp.23   Comp.24
## 0.7172988 0.7032092 0.6947226 0.6700157 0.6506659 0.6329861 0.6227395 0.6179456
##   Comp.25
## 0.5123856
```

A little trickery will get this into something plottable (that you have mainly seen before). The first thing is that this is a named vector, so `enframe` will make it into a two-column dataframe:

```
bfi.1$sdev %>%
  enframe()
```

```
## # A tibble: 25 x 2
##    name    value
##    <chr>   <dbl>
##  1 Comp.1  2.27
##  2 Comp.2  1.66
```

```
##  3 Comp.3  1.46
##  4 Comp.4  1.36
##  5 Comp.5  1.24
##  6 Comp.6  1.04
##  7 Comp.7  0.916
##  8 Comp.8  0.894
##  9 Comp.9  0.848
## 10 Comp.10 0.830
## # ... with 15 more rows
```

then we want to pull those numbers out of the column called `name`, to put on the x-axis of the plot, a technique we have also seen most of before:

```
bfi.1$sdev %>%
  enframe() %>%
  separate(name, into = c("junk", "component"), sep = "\\.", convert = TRUE)
```

```
## # A tibble: 25 x 3
##    junk  component value
##    <chr>     <int> <dbl>
##  1 Comp          1 2.27
##  2 Comp          2 1.66
##  3 Comp          3 1.46
##  4 Comp          4 1.36
##  5 Comp          5 1.24
##  6 Comp          6 1.04
##  7 Comp          7 0.916
##  8 Comp          8 0.894
##  9 Comp          9 0.848
## 10 Comp         10 0.830
## # ... with 15 more rows
```

You might have been thinking about `parse_number`, but it treats those dots as decimal points. The problem with `separate` is that it needs a "regular expression" of things to split at, and "dot" has a special meaning in regular expressions ("one of any character"), while we want a literal dot, so we have to "escape" it. The `convert` thing is to make sure that the component actually *is* a number as far as R is concerned. I forgot this the first time, and wondered why my graph looked strange.

Another way to do it is this way: replace the dots in `name` with something else, *then* use `parse_number`:

```
bfi.1$sdev %>%
  enframe() %>%
  mutate(name = str_replace(name, "\\.", " ")) %>%
  mutate(component = parse_number(name))
```

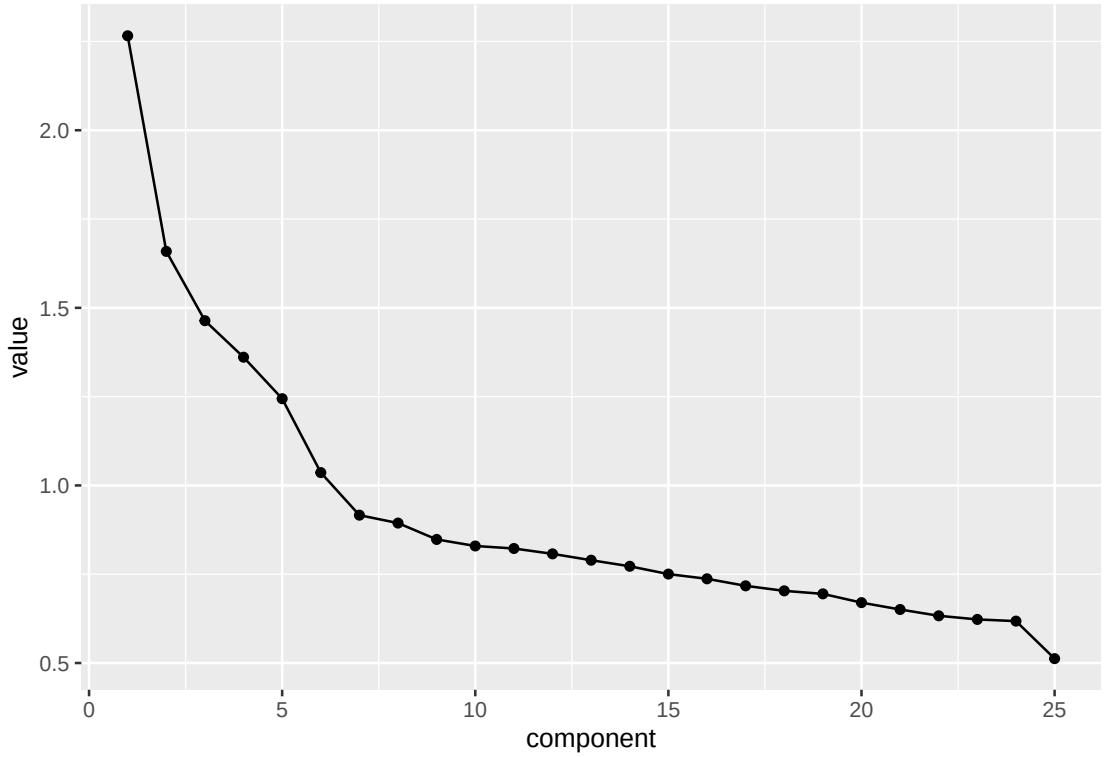
```
## # A tibble: 25 x 3
##    name    value component
##    <chr>   <dbl>     <dbl>
##  1 Comp 1  2.27          1
##  2 Comp 2  1.66          2
##  3 Comp 3  1.46          3
##  4 Comp 4  1.36          4
```

```
##  5 Comp 5  1.24          5
##  6 Comp 6  1.04          6
##  7 Comp 7  0.916         7
##  8 Comp 8  0.894         8
##  9 Comp 9  0.848         9
## 10 Comp 10 0.830        10
## # ... with 15 more rows
```

This suffers from the same problem with the dots as above, so I don't really know how much of a gain it is.[2]

After all that, making the scree plot is straightforward:

```
bfi.1$sdev %>%
  enframe() %>%
  separate(name, into = c("junk", "component"), sep = "\\.", convert = TRUE) %>%
  ggplot(aes(x = component, y = value)) + geom_point() + geom_line()
```



It might have been necessary to square the standard deviations (the plot may use variances), but I'm not going to quibble if you got this far. (In any case, using SDs instead of variances is not going to change where the elbows occur, or, very much, what they look like.)

(c) Briefly justify the use of six factors.

Solution:

There is a clear elbow at 7 components/factors, so that we should go back one and take six factors. (Note: seven is the first one on the scree; for this, we want the last one on the mountain.)

You might consider that the sixth component is itself almost on the scree. It turns out that the sixth factor doesn't have all that much to say.

Extra (a long one, sorry): yesterday, I learned of a thing called "parallel analysis" that is a more automated version of looking for elbows. It is a bit computationally intensive (coding *and* brain-power), but I will take you through it. The idea is to work out something like a "null hypothesis" scree plot, and compare it with the scree plot that you actually had. If anything on your scree plot is above the "null", that indicates "real" factors, more than chance.

What do I mean by "null hypothesis" here? I mean, where each variable is independent of the others, so that you would need as many factors as you have variables (there is no correlation to take advantage of). Then you work out what such a scree plot would look like, and plot it on top of your actual scree plot. (Actually, you don't quite do that; you work out many such fake scree plots and take the median of them, and then plot *that*.)

The clearest description I found of how to do it is actually on a SAS web page. It even comes with SAS code to do it, which is truly horrific (because SAS). The piece to look at is the one called Methodology. We have already done the first step, which is to make the scree plot for our data. The business end of the process is in their step 2.

The first thing we have to do is to generate some random normal data (`rnorm`) of the same dimensions as the original data (in our case, 2436 rows by 25 columns, not counting the IDs). I don't want to do one *quite* as big as this right off, because I want to get my method working first, so let's do 10 rows and 2 columns. Putting these in a dataframe is annoying (because you have to name all the columns), but I think I can do it using the `matrix` data type (because all the random normal values are (decimal) numbers).

So, I need $10 \times 2 = 20$ random (standard) normals to begin with, and before I do *that* I set the random number seed so that the values won't change every time I knit this:

```
set.seed(457299)
z <- rnorm(20)
z
```

```
##  [1]  1.621867352 -0.746347365 -0.268930797 -0.699535090  0.213237930
##  [6]  0.708968535 -1.078329045  0.791310415  0.004046959  1.095879569
## [11] -1.655475142 -1.206874304  1.268749118  0.838393233 -0.746106341
## [16]  0.052753612  1.514875388 -0.112308710  0.266535207 -1.720378300
```

Next, I arrange these into a `matrix` of the right shape:

```
m <- matrix(z, nrow = 10)
m
```

```
##                [,1]        [,2]
##  [1,]  1.621867352 -1.65547514
##  [2,] -0.746347365 -1.20687430
##  [3,] -0.268930797  1.26874912
##  [4,] -0.699535090  0.83839323
##  [5,]  0.213237930 -0.74610634
##  [6,]  0.708968535  0.05275361
##  [7,] -1.078329045  1.51487539
##  [8,]  0.791310415 -0.11230871
##  [9,]  0.004046959  0.26653521
## [10,]  1.095879569 -1.72037830
```

I only had to specify that there were 10 rows (or that there were 2 columns); it can work out the other one from the fact that I had 20 values in `z`.[3] This neither has, nor needs, column names, an advantage of using dataframes here.

Next thing is to calculate the correlation matrix, thinking of this as two variables with ten observations each (like a dataframe of the same shape):

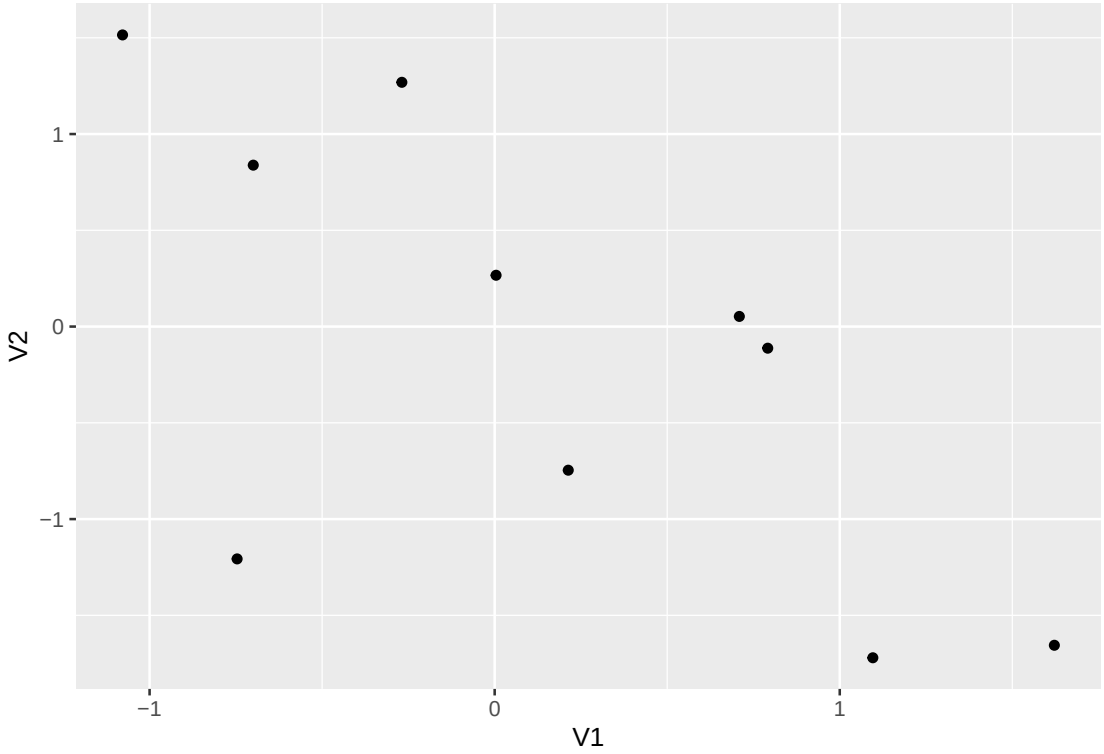
```
v <- cor(m)
v
```

```
##             [,1]       [,2]
## [1,]  1.0000000 -0.6698973
## [2,] -0.6698973  1.0000000
```

There is actually a surprisingly substantial negative correlation here, given that the values were randomly and independently generated:

```
m %>% as_tibble() %>%
  ggplot(aes(x=V1, y=V2)) + geom_point()
```

```
## Warning: The `x` argument of `as_tibble.matrix()` must have unique column names if `.name_r
## Using compatibility `.name_repair`.
```



but that is, in fact, completely chance. The last stage is to find the "eigenvalues" of this correlation matrix. Eigenvalues are a concept from linear algebra (if you have taken MATA23, you'll have seen them there):

```
eigen(v)$values
```

```
## [1] 1.6698973 0.3301027
```

There are two eigenvalues here, because there are two columns. Here, the first one is quite a

bit bigger than the second one, because the data we generated happened to be quite strongly correlated. Let's try the process again:

```
z <- rnorm(20)
m <- matrix(z, nrow = 10)
v <- cor(m)
v
```

```
##           [,1]      [,2]
## [1,] 1.0000000 0.3882933
## [2,] 0.3882933 1.0000000
```

This time, the correlation is smaller, so the eigenvalues should be closer together. Are they?

```
eigen(v)$values
```

```
## [1] 1.3882933 0.6117067
```

They are.

The last part of step 2 in Methodology is to rank these largest to smallest, which R will do for us automatically. Our scree plot wants these as fractions of the total, so we'll work these out as well (below).

If you look ahead to step 3, you'll see that we have to do this many times, so it would be smart to code up the above as a function before we do that. This is mostly copying and pasting our code. So that we can use our function for the real thing (for data of the size that we actually had), we'll have the number of observations and variables be input to our function. Handy coding hint: make a code chunk, in it type `fun`, wait a moment until "`fun` (snippet)" appears, then hit Enter. This gives you a function skeleton; type the name of the function, hit Tab, type the inputs, hit Tab again, and that will put you in the body of the function, where you can type/copy the function code.

```
sim_eigen <- function(n_obs, n_var) {
  z <- rnorm(n_obs*n_var)
  m <- matrix(z, nrow = n_obs)
  v <- cor(m)
  evals <- eigen(v)$values
  tibble(j = 1:n_var, e = evals/sum(evals))
}
```

To make this work, I had to generate the right number of random normals (number of rows times number of columns), and make a matrix of the right size (with as many rows as observations). I decided to make the output be a dataframe, with a column `j` that says which eigenvalue is which.

The output should be as many eigenvalues as variables (reasonably close together, since any correlation is just chance):

```
sim_eigen(12, 3)
```

```
## # A tibble: 3 x 2
##       j     e
##   <int> <dbl>
## 1     1 0.529
## 2     2 0.331
## 3     3 0.140
```

If you're a fan of The Simpsons, you are probably now thinking of this.

Step 2 is finally done. Let's hope it's all downhill from here.

Step 3 says to do this lots of times. Let's do exactly 3 times, since we're still testing, and go back to our 10 rows of 2 variables:

```
sim <- rerun(3, sim_eigen(10, 2))
sim
```

```
## [[1]]
## # A tibble: 2 x 2
##       j     e
##   <int> <dbl>
## 1     1 0.545
## 2     2 0.455
##
## [[2]]
## # A tibble: 2 x 2
##       j     e
##   <int> <dbl>
## 1     1 0.607
## 2     2 0.393
##
## [[3]]
## # A tibble: 2 x 2
##       j     e
##   <int> <dbl>
## 1     1 0.522
## 2     2 0.478
```

These are (mostly) convincingly just over 50% and just under.

Step 4 says to work out the median of the biggest eigenvalues, the second biggest, and so on. We can go back to the tidyverse to take care of this:

```
sim %>% enframe()
```

```
## # A tibble: 3 x 2
##    name value
##   <int> <list>
## 1     1 <tibble [2 x 2]>
## 2     2 <tibble [2 x 2]>
## 3     3 <tibble [2 x 2]>
```

Each row is one simulation, and the two eigenvalues are in that list-column called `value`. Let's pull them out:

```
sim %>% enframe() %>%
  unnest(value)
```

```
## # A tibble: 6 x 3
##    name     j     e
##   <int> <int> <dbl>
## 1     1     1 0.545
## 2     1     2 0.455
```

```
## 3     2     1 0.607
## 4     2     2 0.393
## 5     3     1 0.522
## 6     3     2 0.478
```

Now we see the value of having that column called `j`: if we group by `j` and summarize `e`, we'll have exactly what we wanted.[4]

Next, then:

```
sim %>% enframe() %>%
  unnest(value) %>%
  group_by(j) %>%
  summarize(med = median(e))
```

```
## # A tibble: 2 x 2
##       j   med
##   <int> <dbl>
## 1     1 0.545
## 2     2 0.455
```

and that, then, could be drawn on top of the scree plot. We'll see how shortly, but before we do that, let's put these steps into a function as well so that we don't have to repeat ourselves:

```
sim_median <- function(n_obs, n_var, n_sim) {
  sim <- rerun(n_sim, sim_eigen(n_obs, n_var))
  sim %>% enframe() %>%
  unnest(value) %>%
  group_by(j) %>%
  summarize(med = median(e))
}
sim_median(n_obs = 10, n_var = 2, n_sim = 3)
```

```
## # A tibble: 2 x 2
##       j   med
##   <int> <dbl>
## 1     1 0.535
## 2     2 0.465
```

The last line was to test the function on the one we just worked through. I think the answers are similar enough, given that we only did three simulations. (The new function has three inputs, so I named them when running the function to make sure that I got the right thing in the right place.)

All right, are you ready for the real thing? Hold your breath. There were 2436 observations of 25 variables, and we'll do 1000 simulations. (I think `eigen` is quite quick.):

```
medians <- sim_median(n_obs = 2436, n_var = 25, n_sim = 1000)
medians
```

```
## # A tibble: 25 x 2
##        j    med
##    <int>  <dbl>
##  1     1 0.0474
##  2     2 0.0463
##  3     3 0.0455
```

```
##  4     4 0.0448
##  5     5 0.0442
##  6     6 0.0436
##  7     7 0.0430
##  8     8 0.0424
##  9     9 0.0419
## 10    10 0.0414
## # ... with 15 more rows
```

I seem to need to hand-construct the screeplot to add this to it, but we have the tools to do it now. This is step 1, that I should have done first. There is some trickery here to do it as a pipeline (which does not have to be done this way):

```
bfi_numbers %>%
  cor() %>% eigen() %>%
  .[["values"]] %>%
  enframe() -> eigen0
eigen0
```

```
## # A tibble: 25 x 2
##     name value
##    <int> <dbl>
##  1     1 5.13
##  2     2 2.75
##  3     3 2.14
##  4     4 1.85
##  5     5 1.55
##  6     6 1.07
##  7     7 0.840
##  8     8 0.799
##  9     9 0.719
## 10    10 0.688
## # ... with 15 more rows
```

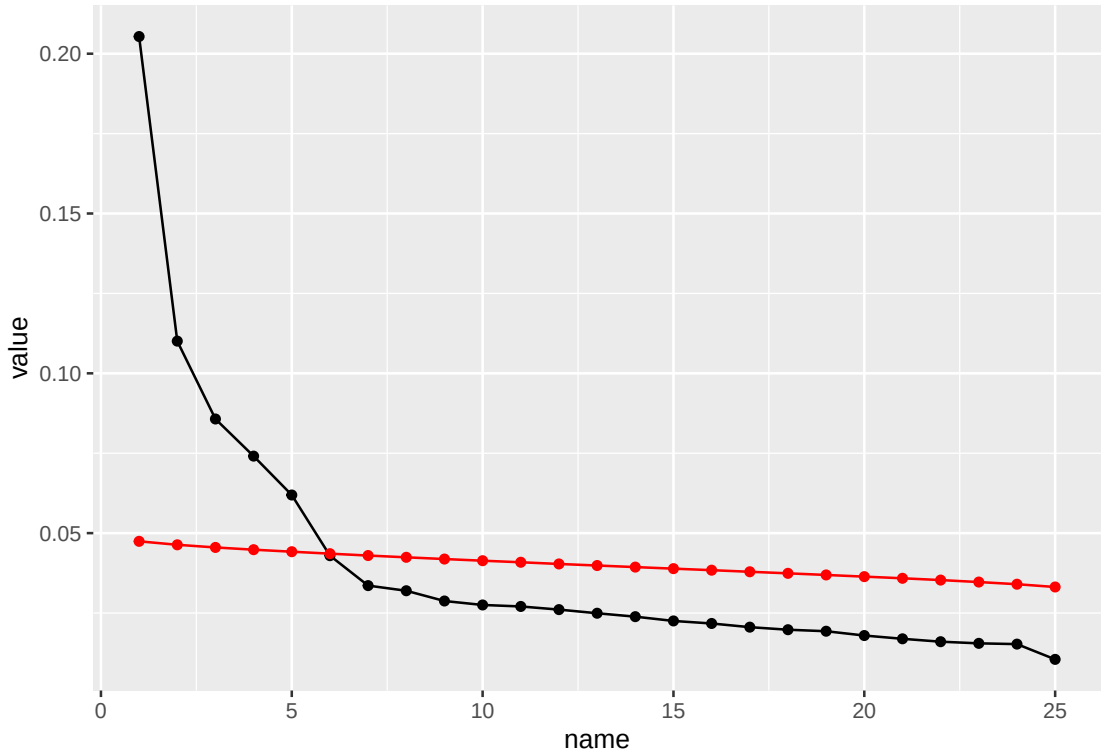
I forgot to divide these by their total:

```
eigen0 %>% mutate(value = value / sum(value)) -> eigen0
eigen0
```

```
## # A tibble: 25 x 2
##     name  value
##    <int>  <dbl>
##  1     1 0.205
##  2     2 0.110
##  3     3 0.0857
##  4     4 0.0741
##  5     5 0.0619
##  6     6 0.0429
##  7     7 0.0336
##  8     8 0.0320
##  9     9 0.0288
## 10    10 0.0275
## # ... with 15 more rows
```

All right (finally), here we go:

```
ggplot(eigen0, aes(x = name, y = value)) + geom_point() + geom_line() +
  geom_point(data = medians, aes(x = j, y = med), colour = "red") +
  geom_line(data = medians, aes(x = j, y = med), colour = "red")
```



All the black points that are above the red ones correspond to components / factors that should be included. In this problem, we should have five factors rather than six, but as you see, the decision about the sixth one is very close.

Gosh, that was long.

(d) Run a factor analysis with six factors. Display the factor loadings. (There is no need to obtain factor scores.)

Solution:

If you saved your dataframe of just the responses before, you can use it again here (best):

```
bfi.2 <- factanal(bfi_numbers, 6)
```

If not, select off the IDs first:

```
bfi %>% select(-id) %>%
  factanal(6) -> bfi.2a
```

(there is no harm in getting factor scores if you did, though the most aware thing is to see that you don't need them for this question, and remove the request for them if you had one), and then:

```
bfi.2$loadings
```

```
##
## Loadings:
##    Factor1 Factor2 Factor3 Factor4 Factor5 Factor6
## A1  0.106                  -0.530  -0.105   0.135
## A2          0.239   0.128   0.663
## A3          0.359   0.122   0.601           0.143
## A4          0.231   0.243   0.401  -0.129
## A5 -0.142   0.435   0.107   0.461           0.224
## C1                  0.557           0.189
## C2                  0.677                   0.160
## C3                  0.548   0.107
## C4  0.219          -0.638  -0.105  -0.133   0.300
## C5  0.278  -0.182  -0.546                   0.143
## E1         -0.583          -0.120           0.156
## E2  0.239  -0.674          -0.110           0.114
## E3          0.566   0.102   0.170   0.251   0.237
## E4 -0.136   0.674   0.115   0.222  -0.107   0.148
## E5          0.506   0.303           0.207
## N1  0.819                  -0.167          -0.125
## N2  0.805                  -0.128          -0.176
## N3  0.710
## N4  0.561  -0.344  -0.162                   0.181
## N5  0.512  -0.163                  -0.158   0.165
## O1          0.234   0.127           0.483   0.181
## O2  0.158                          -0.494   0.137
## O3          0.337                   0.572   0.193
## O4  0.205  -0.169           0.133   0.352   0.177
## O5                                 -0.572   0.146
##
##                Factor1 Factor2 Factor3 Factor4 Factor5 Factor6
## SS loadings      2.722   2.652   2.075   1.672   1.512   0.614
## Proportion Var   0.109   0.106   0.083   0.067   0.060   0.025
## Cumulative Var   0.109   0.215   0.298   0.365   0.425   0.450
```

(e) Which items belong to factor 1? What do they appear to have in common, in terms of personality?

Solution:

Look for the large (far from zero) loadings. These belong to items N1 through N5. These are: gets angry easily, gets irritated easily, has frequent mood swings, often feel blue, panic easily. Devotees of the Big Five will recognize this as "neurotic" (hence the N), but feel free to use something less technical like "off-centre" or "out of equilibrium" or "has unhealthy feelings" to describe this.

(f) Which items belong to factor 2? What kinds of personality do they appear to have in common? What does it mean that some of the loadings are negative?

Solution:

E1 through E5, and possibly A5 (depending on where you draw the line). E1 and E2 have negative loadings, which we'll get to shortly.

These are:

- E1: Doesn't talk a lot.
- E2: Finds it difficult to approach others.
- E3: Knows how to captivate people.
- E4: Makes friends easily.
- E5: Takes charge.
- A5: Makes people feel at ease

These are all about dealing with other people, the first two failing to do so, and the others in a more positive way. The Big Five name for this is Extraversion. The negative loadings on the first two reflect that someone who rates themselves high on those will be highly *introverted*, so they're the opposite way around from the others, "reverse-coded" in the jargon.

Item A5 seems to belong with these, though the A items as a whole are about being aware of the feelings of others. (You see that they are together in factor 4, with "I am indifferent to the feelings of others" having a negative loading.) The Big Five term is Agreeableness, being friendly to others vs. being critical of others.

Extra: factors 3 through 6 are for the most part pretty clear:

- factor 3 is C1 through C5
- factor 4 is A1 through A5
- factor 5 is O1 through O5
- by the time we get to factor 6, there is not much left. Maybe A5 (again), C4, and E3. These are "Make people feel at ease", "Do things in a half-way manner", and "Know how to captivate people", which don't seem to have all that much to do with each other. By the time we get to this point, though, we have exhausted the Big Five and there is not much left.

All of which makes me wonder why the scree plot didn't say to use *five* factors rather than six.

The other thing I didn't ask you to examine was the uniquenesses:

```
bfi.2$uniquenesses
```

```
##        A1        A2        A3        A4        A5        C1        C2        C3
## 0.6749554 0.4821888 0.4734142 0.6962639 0.5156239 0.6371819 0.4983659 0.6854219
##        C4        C5        E1        E2        E3        E4        E5        N1
## 0.4236346 0.5663439 0.6129024 0.4532778 0.5216063 0.4315218 0.5988271 0.2730493
##        N2        N3        N4        N5        O1        O2        O3        O4
## 0.3023016 0.4800265 0.5041211 0.6519006 0.6626007 0.7028841 0.5095268 0.7563584
##        O5
## 0.6398522
```

None of these is especially worrying, which is no surprise because everything appeared in one of the first five factors (the one that their Big Five personality trait said).

(g) The individual with ID 63957 has the most negative score on factor 2. Does this individual have the kinds of scores on the appropriate items that you would expect? Explain briefly.

Solution:

This should be a highly *introverted* person, with low scores on items E3 through E5 (and possibly A5) and high scores on items E1 and E2. Are they? Take a look at their data to see. I'm going to grab just the relevant columns (including A5 for me, but you don't need to):

```
bfi %>% filter(id == 63957) %>%
  select(A5, starts_with("E"))
```

```
## # A tibble: 1 x 6
##      A5    E1    E2    E3    E4    E5
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1     4     6     6     1     1     1
```

Bearing in mind that this is a six-point scale, the scores on the E items are exactly as we would have expected: the highest on the first two items and the lowest on the other three. We might have expected a lower score on A5, but it *is* possible to make people feel at ease without being an extravert, so this is reasonable.

2. An insurance company held a company picnic one summer. Unfortunately, a number of the employees got food poisoning after the picnic, and the issue arose whether it was due to one or more of the foods at the picnic. After some investigation, suspicion centred around two of the foods: crabmeat and potato salad. Each employee was given a questionnaire asking for a yes or no response to three questions: (i) whether they ate any crabmeat, (ii) whether they ate any potato salad, (iii) whether they got food poisoning. The company statistician counted up how many responses there were for each of the $2^3 = 8$ possible combinations, and saved the data in http://ritsokiguess.site/STAD29/picnic.csv. Unfortunately, the company statistician didn't know what to do next. Can you help the insurance company find out whether either or both of the two foods were responsible for the food poisoning?

   (a) Read in and display the data.

Solution:

With only eight rows, you will probably see it all:

```
my_url <- "picnic.csv"
picnic <- read_csv(my_url)
```

```
##
## -- Column specification --------------------------------------------------------
## cols(
##   crabmeat = col_character(),
##   potato_salad = col_character(),
##   illness = col_character(),
##   count = col_double()
## )
```

```
picnic
```

```
## # A tibble: 8 x 4
##   crabmeat potato_salad illness count
##   <chr>    <chr>        <chr>   <dbl>
## 1 yes      yes          yes       120
## 2 yes      yes          no         80
## 3 yes      no           yes         4
```

```
## 4 yes      no           no         31
## 5 no       yes          yes        22
## 6 no       yes          no         24
## 7 no       no           yes         0
## 8 no       no           no         23
```

A number of people (in `count`), and the eight combinations of yes and no answers to the three questions.

If you want to guess about what is happening, you might note that the two very low frequencies are for people who got sick, but who did not have the potato salad. You might infer from this that the potato salad has something to do with it. The two highest frequencies are not so helpful, though, because they are people who ate both; some of them got sick and some didn't.

(b) Build a suitable log-linear model to find the significant associations. To do this, use `glm` and then use `update` to remove what can be removed.

Solution:

This is the same procedure as in the examples in class. Begin with a model containing all the interactions, and hope that some of them can be removed:

```
picnic.1 <- glm(count ~ crabmeat*potato_salad*illness, data=picnic, family = "poisson")
drop1(picnic.1, test = "Chisq")
```

```
## Single term deletions
##
## Model:
## count ~ crabmeat * potato_salad * illness
##                                Df Deviance    AIC    LRT Pr(>Chi)
## <none>                              0.0000 52.332
## crabmeat:potato_salad:illness  1   2.7427 53.074 2.7427   0.0977 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

At $\alpha = 0.05$, the three-way interaction does come out (which saves us having to interpret it):

```
picnic.2 <- update(picnic.1, .~. - crabmeat:potato_salad:illness)
drop1(picnic.2, test = "Chisq")
```

```
## Single term deletions
##
## Model:
## count ~ crabmeat + potato_salad + illness + crabmeat:potato_salad +
##     crabmeat:illness + potato_salad:illness
##                        Df Deviance     AIC    LRT  Pr(>Chi)
## <none>                       2.743  53.074
## crabmeat:potato_salad  1    7.644  55.976  4.901   0.02684 *
## crabmeat:illness       1    6.482  54.813  3.739   0.05316 .
## potato_salad:illness   1   53.683 102.014 50.940 9.524e-13 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The crabmeat by illness interaction (just) comes out:

```
picnic.3 <- update(picnic.2, .~. - crabmeat:illness)
drop1(picnic.3, test = "Chisq")
```

```
## Single term deletions
##
## Model:
## count ~ crabmeat + potato_salad + illness + crabmeat:potato_salad +
##     potato_salad:illness
##                       Df Deviance     AIC    LRT Pr(>Chi)
## <none>                     6.482  54.813
## crabmeat:potato_salad  1   17.157  63.489 10.676 0.001086 **
## potato_salad:illness   1   63.196 109.527 56.714 5.04e-14 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

and here we stop. It begins to look indeed as if it was the potato salad that had to do with the illness.

(c) Investigate each of the significant remaining associations: that is to say, discuss the kind of association that each represents.

Solution:

There are two significant interactions, indicating two significant associations. Let's take the one involving illness first, since that was our major focus here:

```
xt <- xtabs(count ~ potato_salad + illness, data = picnic)
xt
```

```
##             illness
## potato_salad  no yes
##          no   54   4
##          yes 104 142
```

You can stop here if it's clear enough for you already, or you can get a table of proportions as well. The outcome, illness, is in the columns, so you want the *rows* to add up to 1 (as I have done it: if you put `illness` first, your *columns* will have to add up to 1):

```
prop.table(xt, margin = 1)
```

```
##             illness
## potato_salad         no        yes
##          no  0.93103448 0.06896552
##          yes 0.42276423 0.57723577
```

Out of the people who did not eat the potato salad, almost all of them did not get sick. However, over half of the people who *did* eat the potato salad did get sick. There's your smoking gun right there: the food poisoning came from the potato salad.

What about that other association, the one between potato salad and crabmeat? That had nothing to do with food poisoning, but what does it say?

```
xt <- xtabs(count ~ crabmeat + potato_salad, data = picnic)
xt
```

```
##         potato_salad
## crabmeat  no yes
##      no   23  46
##      yes  35 200
```

If you want, you can have a `prop.table` here too. There's no outcome here, so you can do it either way around. Let's have `margin = 2` for a change, but then we have to interpret it the consistent way around as well:

```
prop.table(xt, margin = 2)
```

```
##         potato_salad
## crabmeat        no       yes
##      no  0.3965517 0.1869919
##      yes 0.6034483 0.8130081
```

Out of the people who did not have potato salad, a majority (60%) had crabmeat, but of the people who *did* have potato salad, over 80% of them had crabmeat as well. The short way of saying this is that most people had both, but that doesn't really show up the association; for that, say something like "if a person had potato salad, then they were more likely to have crabmeat as well", the appropriate way around for the way you did the proportions, if you did.

Notes

1. from package `ggbiplot` that you had so much fun installing a couple of weeks ago.

2. There *is* a `parse-integer`, but unfortunately it doesn't work here.

3. The syntax is: thing to make the matrix out of, then one or more of number of rows or columns.

4. I have to say that I didn't think of this the first time. I originally had my function output a vector of eigenvalues rather than a dataframe with them labelled, and by the time I got to the first draft here, I realized that it would be much easier if I knew which eigenvalue was which. So I went back and changed my function to add that. Writing is never a linear process, plus, it's not how many errors you make, but how many you *catch*.