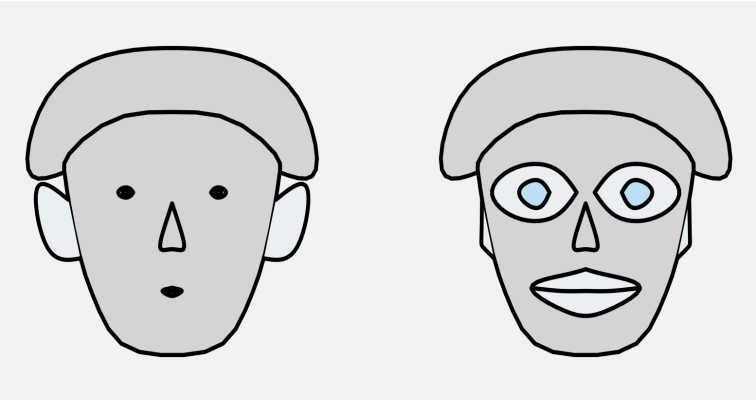
Giving your data a face – Chernoff plots in R

Annika Hamachers
German Police University, Münster
annika.hamachers@dhpol.de



Abstract

This short tutorial demonstrates the use of the `faces()` function from the R package `aplpack` to visualize multidimensional data in the pictogram style introduced by Herman Chernoff in 1973.

1 Intro

So called "Chernoff faces" are a quite extraordinary way to visualize multidimensional data in a creative, almost playful way that yet can be easily interpreted.

The concept was first promoted by Herman Chernoff in his 1973 paper, "The Use of Faces to Represent Points in k- Dimensional Space Graphically". Chernoff's basic idea was to make use of the fact that humans can 'read' other human faces quite intuitively: We easily discern and memorize a large variety of facial features and expressions. Hence, presenting data as faces – rendering the single variables as facial features – would facilitate to "represent multivariate data, subject to strong but possibly complex relationships, in such a way that an investigator can quickly comprehend relevant information."

2 Set-up

In R, we can map our data into the ranges of the Chernoff face parameters with the `faces()` function from the `aplpack` package. This acronym stands for 'Another Plot Package' and is worth checking out ways beyond its ability to produce Chernoff faces: it holds functions for summary plots, icon plots, bagplots etc.

Alongside this package, we would want to load `readr` into the current R session for conveniently importing our data that we hold in a csv file (if those packages are not in your library, yet, uncomment the first two lines).

```
#install.packages('readr')
#install.packages('aplpack')
library(readr) # package to read in the CSV
library(aplpack) #package containing faces()
```

3 Quick Chernoff Faces

For this tutorial we read in data that stem from a meta analytical literature review that compared communication science research papers in international journals with those in German journals. For each scope, the data table contains count data for the different research methods, subdisciplines, and topics that were encountered:

	A	B	C	D	E
1	Scope	observation	interview	contentAnal	mediaSyste
2	national	65	678	394	4
3	internationa	100	774	251	1
4					

Reading in those data, make sure to keep the headers, so you can use the variable names as labels later:

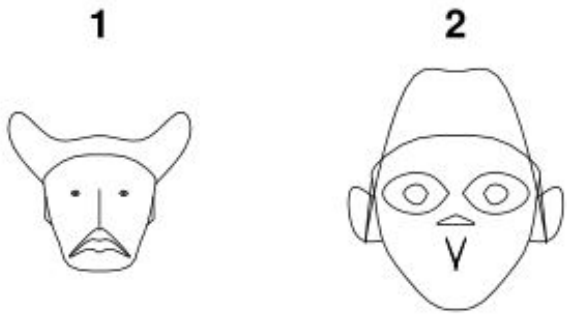
```
myData <- read.csv("Chernoff.csv", sep=";", header=TRUE)
```

Producing Chernoff faces from such a dataframe, is – at least in theory – super easy: We just call `faces()` on the dataframe and specify the face type, we would like to get (0 standing for line drawings, 1 for color-filled faces, and 2 for a little bit silly Santa Claus faces). However, in our case this results in an error:

```
## Error in x - min(x):  nicht-numerisches Argument für binären Operator
```

This error is due to the fact, that our dataframe holds more variables than there are Chernoff face features available: The maximum number of different traits, we can display, is 15, so, we can only depict a subsection of our variables:

```
faces(myData[,2:16], face.type = 0)
```



This produced two quite distinct faces – one for each of the two rows in our dataframe. If you have a lot of units of observation, you would, of course, get a hole bunch of faces (see for instance this blog post on *fansided.com* that neatly depicts NBA team stats for the season of 2017 as Chernoff faces). However, we would not recommend applying `faces()` to a dataframe with too many rows because it will get very challenging for your audience to intuitively grasp the meaning of the different faces.

4 Editing the Faces

Without additional information on how the function operates, it is, of course, impossible to interpret and further style those faces in a meaningful way. We need to know, which variable in the dataset is being associated with which facial feature.

In general, `faces()` renders the variables in the order of their appearance as depicted in this table:

Column	Feature
1st	height of face
2nd	width of face
3rd	shape of face
4th	height of mouth
5th	width of mouth
6th	curve of smile
7th	height of eyes
8th	width of eyes
9th	height of hair
10th	width of hair
11th	styling of hair
12th	height of nose
13th	width of nose
14th	width of ears
15th	height of ears

In color-filled faces, the distinct colors of facial features are generated by averaging over sets of variables (1st & 2nd = face; 1st, 2nd, & 3rd = lips; 7th & 8th = eyes; 8th, 9th, & 10th = hair; 12th & 13th = nose; 14th & 15th = ears)

When we want to know, what exactly this means in the case of our data, we can call the function again, adding the argument `print.info=TRUE`. If we are only interested in this info and do not need the faces themselves again, we can set `plot.faces` to `FALSE`:

```
faces(myData[,2:16], print.info=TRUE, plot.faces=FALSE)

## effect of variables:
##  modified item       Var
##  "height of face   " "observation"
##  "width of face    " "interview"
##  "structure of face" "contentAnalysis"
##  "height of mouth  " "mediaSystems"
##  "width of mouth   " "mediaContent"
##  "smiling          " "mediaUse"
##  "height of eyes   " "mediaEffects"
##  "width of eyes    " "interpersonalCommunication"
##  "height of hair   " "methods"
##  "width of hair   "  "theDiscipline"
##  "style of hair   "  "print"
##  "height of nose  "  "A.V"
##  "width of nose   "  "internet"
##  "width of ear    "  "games"
##  "height of ear   "  "individualCom"
```

As you can see, there is a lot going on in our faces... To make it easier for the spectator to infer meaning from them, we should try to reduce the variables we want to display and aggregate some of the facial features.

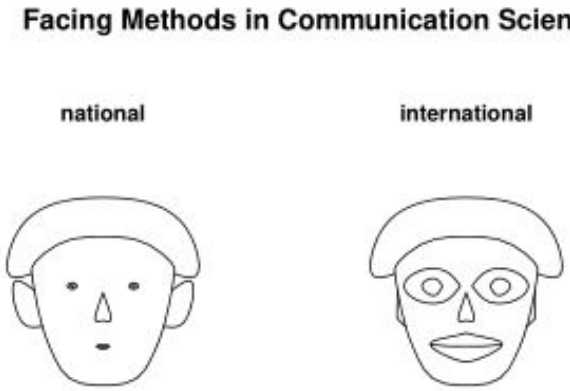
Maybe, it would be a good start to solely focus on the differences in research methods. Accordingly, we would only have to choose three associated features and set the rest to zero. Since we are dealing with count variables in our case, we might want to choose features that change gradually. So, the shape of the face or the styling of the hair would be rather bad choices; those would be more appropriate for categorial variables. Also, the curve of the smile will very likely be associated with a positive or negative evaluation and we'd like to keep it neutral. Instead, since our everyday understanding of an observation has something to do with 'watching', we want this variable to be represented by the size of the eyes, interviews (which involve talking) by the size of the mouth, and content analyses (which do not necessarily have to but could involve listening) by the size of the ears. Therefore, we simultaneously want to adjust their heights and widths:

```
newData <- myData[,1:16]#copy the original data
newData[,2:16] <- 0 #set everything except the scope variable to zero

# override columns that will be associated with the height and width
# of the eyes with the original counts for interviews
newData[,5:6] <- myData$interview

# override columns that will be associated with the height and width
# of the eyes with the original counts for observations
newData[,8:9] <- myData$observation

# override columns that will be associated with the height and width
# of the eyes with the original counts for content analyses
newData[,15:16] <- myData$contentAnalysis
```

Make sure to add 1 to the column values from the table above because we will derive features starting from the second column of our new data, again leaving out the scope variable.

Plotting these new data, we also take the chance to add a title (`main`), set the `labels` to the journal scope and adjust their font size with `cex` (for the full list of options available for the `faces()` function, please refer to the aplpack documentation):

```
faces(newData[,2:16], face.type = 0,
      main="Facing Methods in Communication Science",
      labels=newData$Scope, cex=1.1)
```



Those faces tell you right away, that the German research landscape in communication science is clearly dominated by content analyses whereas internationally surveys and observations are more prevalent.

You can either export your final faces – just as any other R figure – from the plot panel in R Studio or by wrapping the `faces()` function into a device function matching the desired output format where you specify the name and path to which it is to be saved, e.g. like this for a pdf file:

```
pdf(file = "/myFaces.pdf")

faces(newData[,2:16], face.type = 0,
      main="Facing Methods in Communication Science",
      labels=newData$Scope, cex=1.1)

dev.off()
```

If you want to edit this file later as described in the next section, we strongly recommend to choose a device that supports scalable vector graphics (such as svg, pdf, or postscript).

5 Postproduction

Since R is primarily a tool for data analysis and has limited means for styling its output, we would always recommend to perform some further editing steps outside of R to make the faces appear more professional – just as also data viz superstar Nathan Yau suggests on his awesome blog Flowingdata.

Particularly, we would want to use a professional graphic design tool like Adobe's Illustrator Illustrator or its free dupe Inkscape in order to retrace the lines, color the faces, style the fonts, and add a legend to the plot.

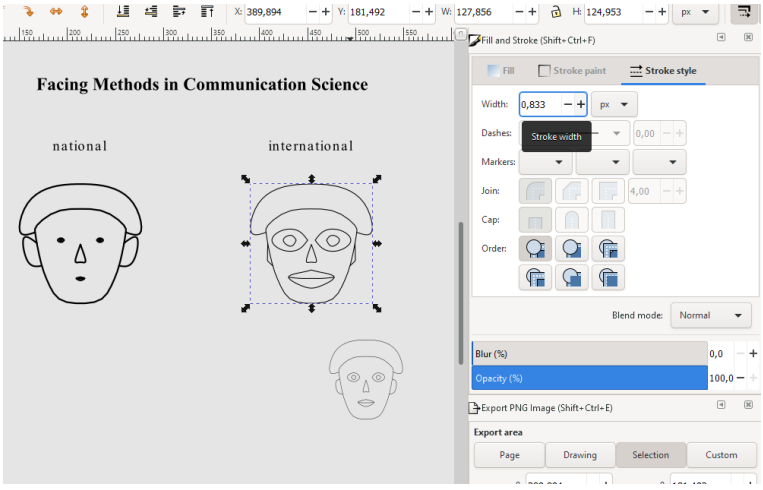
Before switching to this tool (in my case Inkscape), we'll have to do one more thing in R: We create an additional Chernoff face from a 1x15 matrix where all values are set to zeros. This baseline face will serve as our legend:

```
dummyData <- matrix(0, ncol = 15, nrow = 1)
faces(dummyData, face.type = 0, labels="")
```

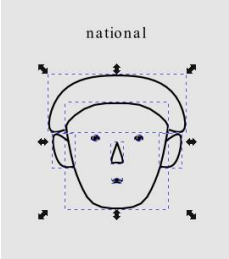
Now, we can open our favorite design tool and simply drag and drop our faces and the legend onto the canvas and start editing. What I for instance did to touch up the graphic a bit was:

1. Set the background to a neutral gray (open up the 'Document Properties' menu by pressing `Control + Shift + D`, then click on 'Background color')

2. Make the lines for our original faces thicker to set them apart from the legend face (open up the 'Fill and Stroke' menu by pressing `Control + Shift + F`, then select 'Stroke style' and adjust the 'Width')



3. Fill in the faces with colors (in the same menu switch to 'Fill', mark the filled square ('flat color') and choose a color you like). If you want different features to appear in different colors, make sure to select the face and hit `Control + Shift + G` several times until you see frames like this that tell you that the elements of the face have been 'ungrouped':



Hint: Though it makes the faces less anthropomorphic, it is nowadays considered good style not to choose skin-toned colors for Chernoff faces in order to avoid racial stereotyping (unless of course the skin-color conveys an actual meaning in your data). So, instead, I chose to make the faces look a bit like the white walkers from GoT – giving them blue irises and gray hair and skin – which also goes nicely with the background and seemed neutral enough. Moreover, I colored the features of interest (the mouths, eyes, and ears) in a lighter gray to make them pop.

4. Adjust texts (double-click text items to style them and/or hit `T` to change your cursor to text insertion mode to add more text elements)

5. Align all elements (change the cursor back by hitting `S`, drag everything in place, and/or hit `Control + Shift + A` to access the 'Align' menu that helps you to space out elements equally).

So, this is what my final graphic looked like: