Preparation: installing R studio

https://posit.co/products/open-source/rstudio/

Preparation: installing Excel reader and loading it into the library

```
> install.packages("readxl")
> library(readxl)
```

Step 1. Creation of dissimilarity matrix based on the file “input_for_distance_matrix.xlsx”

```
> df<-read_excel("path-to-input_for_distance_matrix.xlsx")
> df_factors<-df[ ,-1]
> custom_dissimilarity<-function(data){
+ n<-nrow(data)
+ k<-ncol(data)
+ dist_matrix<-matrix(0,n,n)
+ for(i in 1:n){
+ for(j in 1:n){
+ diff_count <- sum(data[i, ] != data[j, ])
+ dist_matrix[i, j] <- diff_count / k
+ }
+ }
+ dist_df<-as.data.frame(dist_matrix)
+ rownames(dist_df) <- df$ID
+ colnames(dist_df) <- df$ID
+ return(dist_df)}
> dissimilarity_matrix<-custom_dissimilarity(df_factors)
> write.csv2(dissimilarity_matrix, file = "path-to-new-file-distance_matrix.csv")
```

If everything went well, the above code should have led to a new file “distance_matrix.csv” that looks as in *Figure 1*.

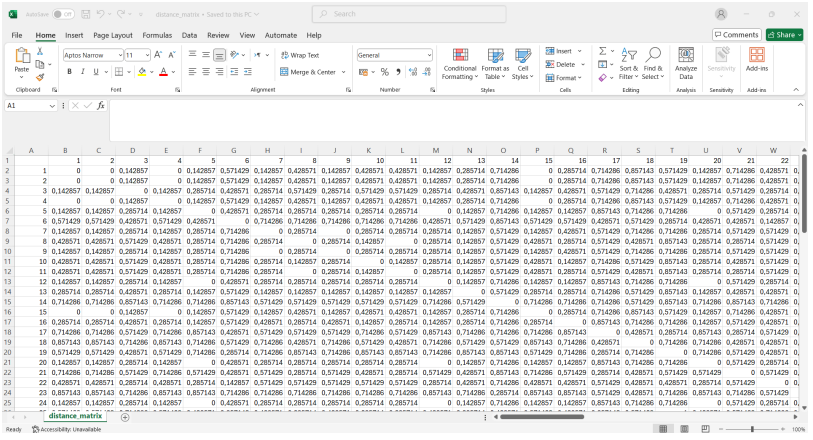

Figure 1: Output of dissimilarity matrix creation

Step 2. Running mds based on the newly created file “distance_matrix.csv”

> dist_matrix <- read.csv2("path-to-new-file-distance_matrix.csv")

> dist_matrix <- dist_matrix[, -1]

> dist_object <- as.dist(dist_matrix)

> mds_result <- cmdscale(dist_object, k = 2)

> write.csv2(mds_result, file = "path-to-mds_result.csv")

If everything went fine, the above code should have led to a new file “mds_result.csv” that looks as in *Figure 2*.

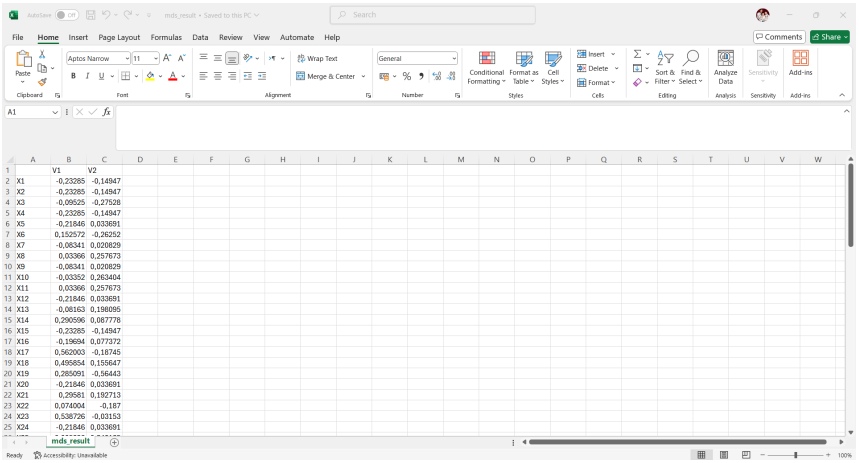

Figure 2: Output of running MDS

Step 3. Visualization of semantic maps

Preparation: copy-pasting the MDS output into the file “input_for_distance_matrix.xlsx” and saving the files as “input_for_ggplot.xlsx”

Copy the V1 and V2 columns from the file “mds_result.csv” to “input_for_distance_matrix.xlsx” and save as the new file “input_for_ggplot.xlsx”. The results looks as in *Figure 3*.

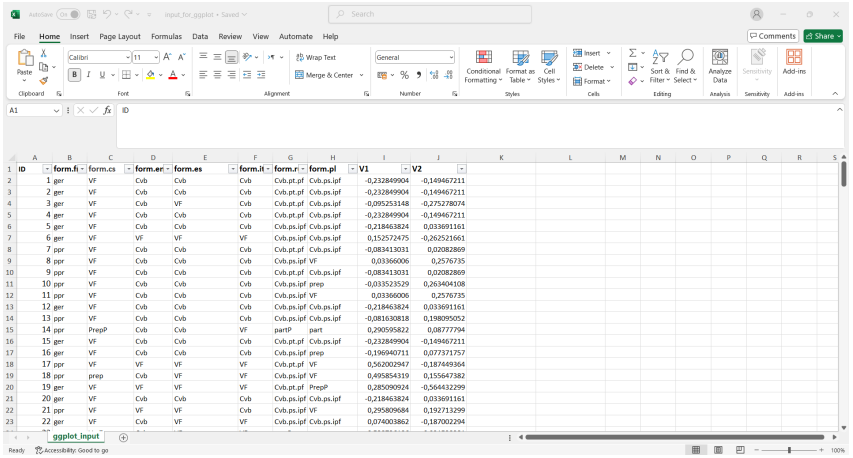

Figure 3: Merge of original data with MDS output

Preparation: downloading and loading the relevant packages into the library

> install.packages("ggplot2")

> library(ggplot2)

Preparation: loading the data

Here, I'm using the simplest possible route, *viz.* importing data directly from the excel file "input_for_ggplot.xlsx". The only setting I adapt from the standard is the name in *Import Options*: from 'input_for_ggplot' to 'data' (shorter to type later on).

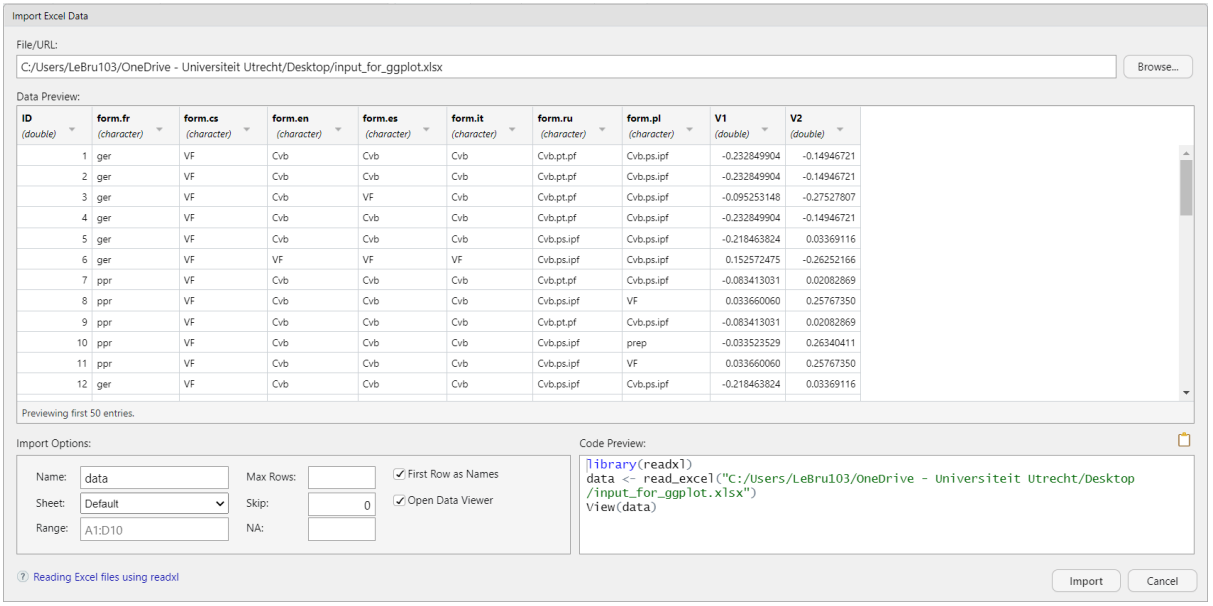

Illustration 4: The Import Excel Data window

Preparation: defining colors

In this step, we define the colors we will be working with. Even though the MDS algorithm is insensitive to there being the same labels across languages (and this is the way we want it to be), maps are easier to interpret if we keep the color coding constant for the different languages. We do this by defining color codes as follows:

> my_colors<-c(

+ "ger"="red",

+ "ppr"="green",

+ "VF"="orange",

+ "Cvb"="purple",

+ "InfP"="blue")

The above list can be extended as needed. You can find an overview of the available colors in R online.

Preparation: adding jitter

In order to make sure that we see all our datapoints, it is practical to add some jitter (=random space between datapoints with the same coordinates). It is important, though, to make sure we can always reproduce our jitter. This requires two steps.

The first step is to fix the randomness we are going to introduce. We do this with the following command:

```
> set.seed(129)
```

The number you choose is irrelevant, as long as you remember which number you picked.

The next step is to create a jittered dataset that we can use as the basis for our visualization. We do this with the following commands:

```
> jitter_amount<-0.05 # with this command, you set the amount of jitter that will be applied

> data$dimension1 <- as.numeric(data$V1) + runif(nrow(data), -jitter_amount, jitter_amount)
```

with this command, you create a variation of the coordinates in V1 by adding the amount of jitter defined before and you save them under the new header 'dimension1'.

```
> data$dimension2 <- as.numeric(data$V2) + runif(nrow(data), -jitter_amount, jitter_amount)
```

with this command, you create a variation of the coordinates in V2 by adding the amount of jitter defined before and you save them under the new header 'dimension2'.

The actual visualization

With all the preparations in place, you can start working on your visualization. You do this with the following commands:

```
> m <- ggplot(data, aes(x = dimension1, y = dimension2,color=form.fr))
```

With this command, you will be able to create the semantic map with the coloring based on French. If you want variations in colors, you just rerun this command with the coloring based on another language (for Italian, you would then change 'form.fr' to 'form.it').

```
> m+geom_point()+

+ scale_color_manual(values=my_colors)
```

With this command, you create the semantic map and you make sure to use the color scheme you prepared before.

The result with the above commands looks as in *Illustration 5*. If you rerun the same commands but with the Italian coloring scheme, the output looks as in *Illustration 6*.

If you want to add ID numbers to the different datapoints, you can use the following variation for the final command:

```
> m+geom_point()+

+ geom_text(aes(label =ID), vjust = -0.5, size = 2)+

+ scale_color_manual(values=my_colors)
```

This will give you the result in *Illustration 7*.

Note that you can do a lot more with the visualizations than I indicate here. You can e.g. have the color scheme depend on the forms in a language and vary the shapes of the datapoints (dots, triangles, squares, etc.) based on the functions in a language.

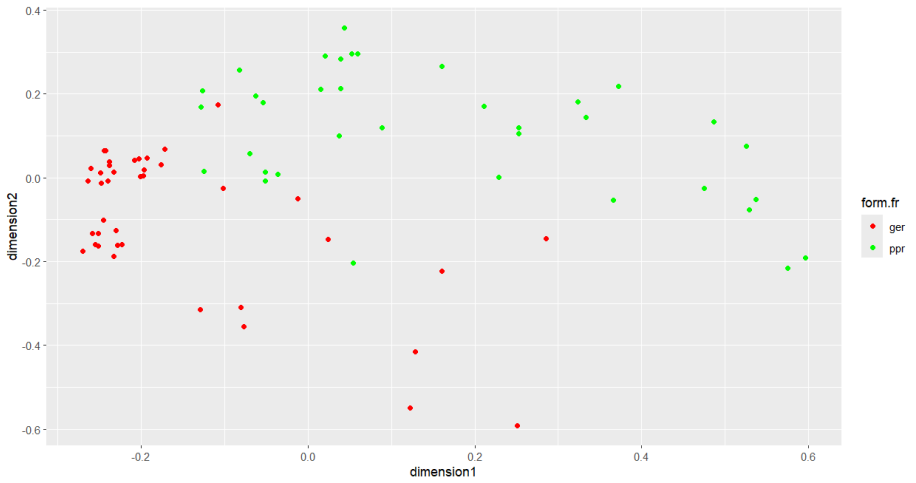

Illustration 5: The semantic map with coloring for French

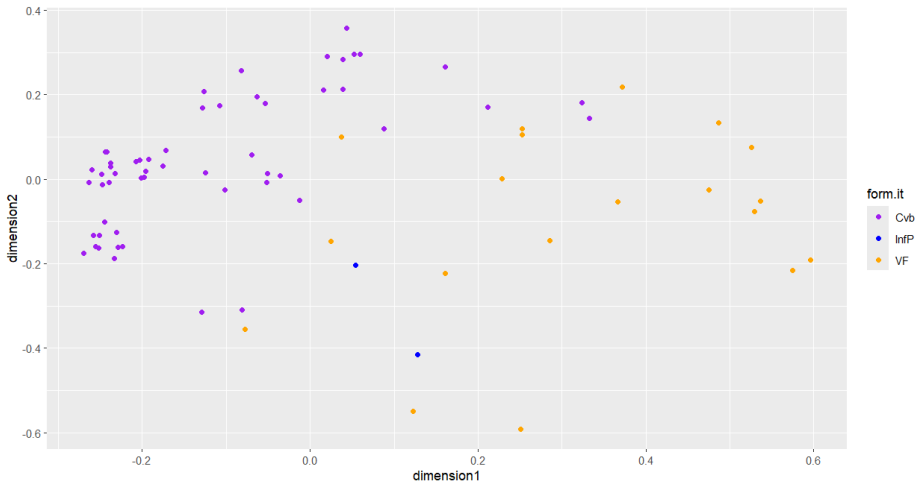

Illustration 6: The semantic map with coloring for Italian

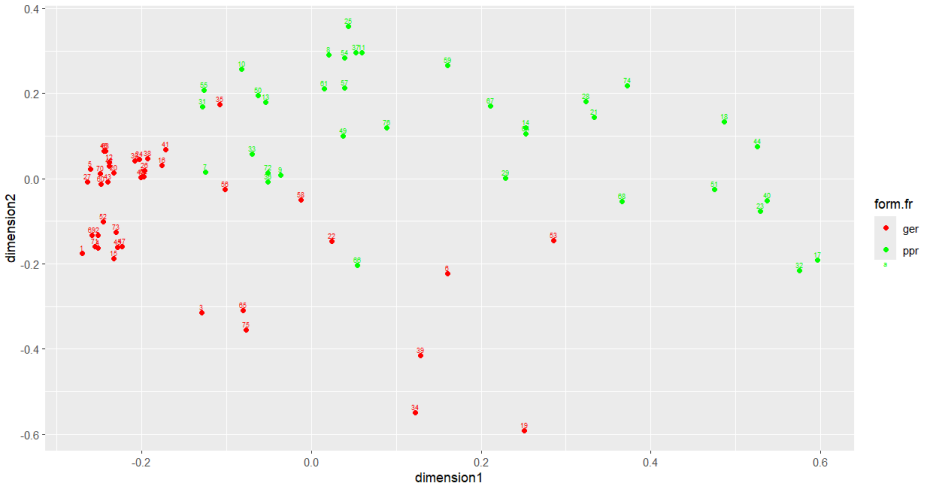

Illustration 7: The semantic map with coloring for French and ID numbers added as labels