Exercises: Factor analysis and multidimensional scaling

In this exercise, you will know

- How to use FA
- How to use classical and ordinal MDS

Don't forget to change your working directory!

1 Factor analysis

We illustrate how to conduct exploratory data analysis using the data from the classic 1939 study by Karl J. Holzinger and Frances Swineford. In the study, a number of tests intended to measure a general factor and other specific factors, were administered to seventh and eighth grade students in two schools; we will only consider the Grant-White School ($n = 145$). The data used in this example (available from the `lavaan` package) include nine tests intended to measure several domains such us verbal ability, speed, and memory.

```
library(lavaan)
```

```
## This is lavaan 0.6-10
## lavaan is FREE software! Please report any bugs.
```

```
dat = HolzingerSwineford1939[HolzingerSwineford1939$school=="Grant-White",]
dat = dat[, -c(1:6)]
names(dat) = c("visual", "cubes", "lozenge", "paragraph", "sentence", "wordm",
                "add", "counting", "straight")
dat[1:3, ]
```

```
##       visual cubes lozenge paragraph sentence    wordm      add counting
## 157 3.833333  4.75   0.500  3.333333     4.25 1.428571 3.000000      4.1
## 158 5.500000  5.50   2.125  2.666667     4.25 1.428571 2.826087      4.9
## 159 5.666667  6.00   2.750  3.666667     4.75 2.714286 2.173913      4.3
##     straight
## 157 4.333333
## 158 5.416667
## 159 6.333333
```

The variables above are scores deriving from tests on visual perception, cubes, lozenges, paragraph comprehension, sentence completion, word meaning, speeded addition, speeded counting of dots, speeded discrimination between straight and curved capitals.

The classical exploratory factor analysis involves (1) preparing data, (2) determining the number of factors, (3) estimation of the model, (4) factor rotation, (5) factor score estimation and (6) interpretation of the analysis.

1.1 Preparing Data

A correlation matrix is required. The following `R` code calculates the correlation matrix.

```
fa.cor = cor(dat)
```

1.2 Determining the Number of Factors

With the correlation matrix, we first decide the number of factors. There are several ways to do it. But all the methods are based on the eigenvalues of the correlation matrix. From R, we have the eigenvalues below. First, note the number of eigenvalues is the same as the number of variables. Second, the sum of all the eigenvalues is equal to the number of variables.

```
fa.eigen = eigen(fa.cor)
fa.eigen$values
```

```
## [1] 3.6146546 1.5622791 1.2505188 0.7067884 0.5327140 0.4213219 0.3580514
## [8] 0.2979867 0.2556851
```

```
sum(fa.eigen$values)
```

```
## [1] 9
```

The basic idea can be related to the variance explained as in regression analysis. With the correlation matrix, we can take the variance of each variable as 1. For a total of p variables, the total variance is therefore p. For factor analysis, we try to find a small number of factors that can explain a large portion of the total variance. The eigenvalues correspond to the variance of each factor. If the eigenvalue corresponding to a factor is large, that means the variance explained by the factor is large. Therefore, the eigenvalues can be used to select the number of factors.

Rule 1

The first rule to decide the number of factors is to use the number of eigenvalues larger than 1. In this example, we have three eigenvalues larger than 1. Therefore, we can have three factors.

```
fa.eigen$values
```

```
## [1] 3.6146546 1.5622791 1.2505188 0.7067884 0.5327140 0.4213219 0.3580514
## [8] 0.2979867 0.2556851
```

Rule 2

Another way is to select the number of factors with the cumulative eigenvalues accounting for 80% of the total variance. This is to say if we add the eigenvalues of the selected number of factor, the total values should be larger than 80% of the sum of all eigenvalues. We could choose four.

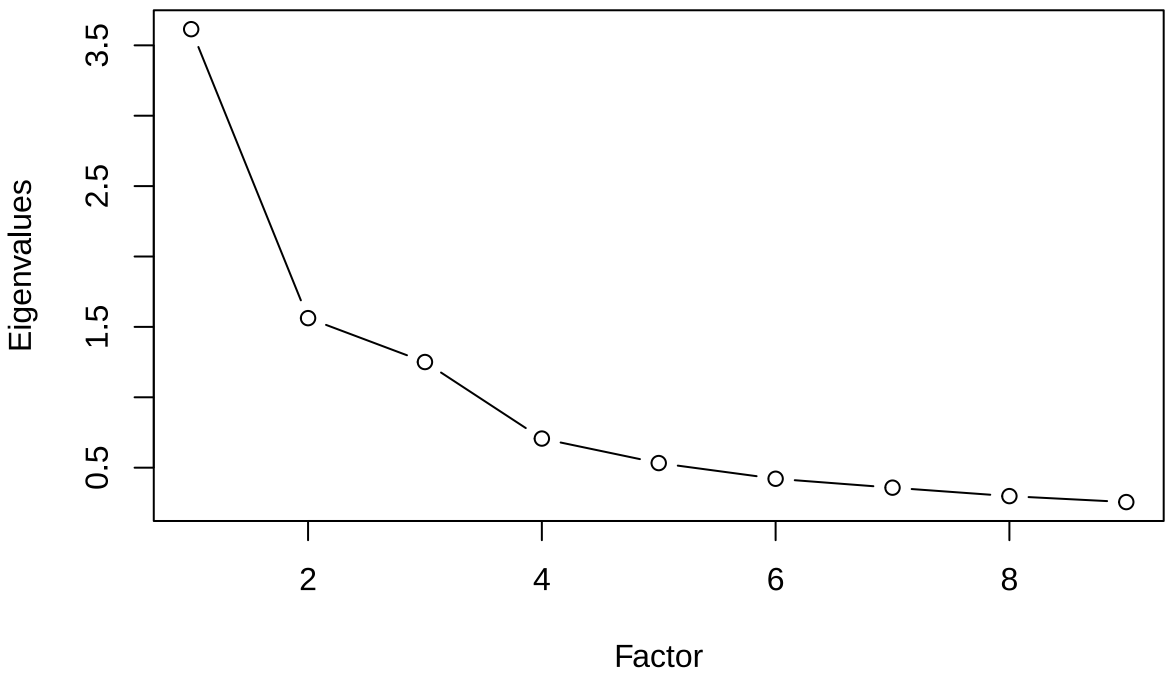
```
cumsum(fa.eigen$values)/9
```

```
## [1] 0.4016283 0.5752149 0.7141614 0.7926934 0.8518839 0.8986974 0.9384809
## [8] 0.9715905 1.0000000
```

Cattell's Scree Plot

The Cattell's Scree plot is a plot of eigenvalues on the Y axis along with the number of factors on the X axis. The plot looks like the side of a mountain, and "scree" refers to the debris fallen from a mountain and lying at its base. As one moves to the right, toward later components/factors, the eigenvalues drop. When the drop ceases and the curve makes an elbow toward less steep decline, Cattell's scree test says to drop all further components/factors after the one starting the elbow. For this example, we can identify four factors based on the scree plot below.

```
plot(fa.eigen$values, type = "b", ylab = "Eigenvalues", xlab = "Factor")
```



1.3 Estimation of Model/Factor Analysis

Once the number of factors is decided, we can conduct exploratory factor analysis using the `R` function `factanal()`. The `R` input and output for this example is given below.

```
fa.res = factanal(x = dat, factors = 4, rotation = "none")
fa.res
```

```
##
## Call:
## factanal(x = dat, factors = 4, rotation = "none")
##
## Uniquenesses:
##    visual     cubes   lozenge paragraph  sentence     wordm       add  counting
##     0.521     0.656     0.478     0.247     0.283     0.316     0.005     0.452
##  straight
##     0.353
##
## Loadings:
##           Factor1 Factor2 Factor3 Factor4
## visual     0.499   0.113   0.465
## cubes      0.326           0.343   0.340
## lozenge    0.499           0.423   0.294
## paragraph  0.811   0.220  -0.215
## sentence   0.769   0.265  -0.194  -0.135
## wordm      0.778   0.189  -0.201
## add                0.997
## counting   0.109   0.592   0.404  -0.148
## straight   0.401   0.428   0.495  -0.241
##
##                Factor1 Factor2 Factor3 Factor4
## SS loadings      2.631   1.707   1.046   0.303
## Proportion Var   0.292   0.190   0.116   0.034
## Cumulative Var   0.292   0.482   0.598   0.632
##
## Test of the hypothesis that 4 factors are sufficient.
## The chi square statistic is 2.59 on 6 degrees of freedom.
## The p-value is 0.858
```

In this type of analysis it is assumed that the observed data consist of two parts, the common factor part and the uniqueness part. The common factor part is based on the four factors, which are also called the common factors. The uniqueness part is also called uniqueness factor, which is specific to each observed variable.

Using the variable `visual` as an example, we have

$$visual = 0.499 * Factor1 + 0.113 * Factor2 + 0.465 * Factor3 + e_{visual}.$$

Note that the factor loadings are from the Loadings section of the output. The loadings are the coefficients of the latent factors on the manifest indicators or observed variables. The variance of the uniqueness is in the Uniquenesses section. For e_{visual}, the variance is 0.521. For the other variables, we apply the same reasoning.

The other section is related to the variance explained by the factors. The SS loadings row provides the sum squared loadings related to each factor. It is the overall variance explained in all the 9 variables by each factor. Therefore, the first factor explains the total of 2.631 variance, that's about $29.2\% = 2.631/9$. Proportion Var includes the variances in the observed variables/indicators explained by each factor. Cumulative Var gives the cumulative proportion of variance explained by all factors.

A test is conducted to test whether the factor model is sufficient to explain the observed data. The null hypothesis that a 4-factor model is sufficient. For this model, the chi-square statistic is 2.59 with degrees of freedom 6. The p-value for the chi-square test is 0.858 which is larger than .05. Therefore, we do not reject the null hypothesis.% that the factor model have a good fit to the data.

1.4 Factor Rotation

Although we have identified four factors and found that the 4-factor model is a good model, we cannot find a clear pattern in the factor loadings to have a deep understanding of the factors. Through factor rotation, we can make the output more understandable; this is usually necessary to facilitate the interpretation of factors. The aim here is to find a simple solution that each factor has a small number of large loadings and a large number of zero (or small) loadings. There are many different rotation methods such as the varimax rotation, quadtimax rotation, equimax rotation, oblique rotation, etc. The PROMAX rotation is one kind of oblique rotation and is widely used. After PROMAX rotation, the factor will be correlated.

```r
fa.res = factanal(x = dat, factors = 4, rotation = "promax")
print(fa.res, cut = 0.2)
```

```
##
## Call:
## factanal(x = dat, factors = 4, rotation = "promax")
##
## Uniquenesses:
##    visual     cubes   lozenge paragraph  sentence     wordm       add  counting
##     0.521     0.656     0.478     0.247     0.283     0.316     0.005     0.452
##  straight
##     0.353
##
## Loadings:
##           Factor1 Factor2 Factor3 Factor4
## visual                     0.470   0.327
## cubes                              0.656
## lozenge                            0.674
## paragraph  0.845
## sentence   0.833
## wordm      0.801
## add                0.996
## counting           0.334   0.526
## straight                   0.775
##
##                 Factor1 Factor2 Factor3 Factor4
## SS loadings       2.085   1.151   1.146   1.024
## Proportion Var    0.232   0.128   0.127   0.114
## Cumulative Var    0.232   0.360   0.487   0.601
##
## Factor Correlations:
##         Factor1 Factor2 Factor3 Factor4
## Factor1  1.0000  -0.168   0.542  0.0742
## Factor2 -0.1679   1.000  -0.379 -0.4578
## Factor3  0.5415  -0.379   1.000  0.5592
## Factor4  0.0742  -0.458   0.559  1.0000
##
## Test of the hypothesis that 4 factors are sufficient.
## The chi square statistic is 2.59 on 6 degrees of freedom.
## The p-value is 0.858
```

In the output, we use `print(fa.res, cut = 0.2)` to show factor loadings that are greater than 0.2. Note that after rotation, many loading are actually smaller than 0.2. The pattern of the factor loadings are clearer now. For example, the variable `visual` has two large loadings (0.470 and 0.327) corresponding to Factors 3 and 4 but small than 0.2 loadings on the other two. In this case, we might say that `visual` is mainly influenced by Factors 3 and 4. A similar reasoning can be applied to the other variables, for instance `paragraph` only has one large loading for Factor 1 and all the others are smaller than 0.2.

We can also see that the primary indicators for Factor 1 are `paragraph`, `sentence` and `wordm`. %And for Factor 4, the indictors include add, code, counting, and straight.

The correlations among the factors are given in the section of Factor Correlation. For example, the correlation between Factor 1 and Factor 2 is 0.17. Note that after rotation, the test of the model is the same as without rotation.

1.5 Interpretation of the Results

Based on the rotated factor loadings, we can name the factors in the model. This can be done by identifying significant loadings. For example, Factor 1 is characterised by `paragrap`, `sentence` and `wordm`, all of which are related to verbal perspective of cognitive ability. One way to name the factor is to call it a verbal factor. Similarly, the second and third can be called speed factor, and the last one can be called the spatial factor.

Rule 1 suggested the presence of three factors, please repeat the analysis using 3 factors.

1.6 Factor Scores

Sometimes, the purpose of factor analysis is to estimate the score of each latent construct/factor for each participant. Factor scores can be used in further data analysis. In general, there are two methods for estimating factor scores: the regression method and the Bartlett method. The second method generally works better. For example, the following code obtains the Bartlett factor scores. %As an example, the linear regression is also fitted.

```
fa.res = factanal(x = dat, factors = 4, rotation = "promax", scores = "Bartlett")
head(fa.res$scores)
```

```
##        Factor1    Factor2    Factor3    Factor4
## 157 -0.3653920 -0.9298950 -1.3138375 -1.9841111
## 158 -0.6757907 -1.0189768  0.1733558 -0.2113125
## 159  0.3167543 -1.6998613  0.5133377  0.6076599
## 160 -0.3702624  0.9922177 -1.0177805 -0.9272467
## 161  0.4850997  0.8545549 -0.7596543 -1.3413952
## 162  0.3827841  0.1453375  0.4200798  0.1356463
```

```
summary(lm(Factor2 ~ Factor1, data = as.data.frame(fa.res$scores)))
```

```
##
## Call:
## lm(formula = Factor2 ~ Factor1, data = as.data.frame(fa.res$scores))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.02587 -0.78857 -0.06933  0.68516  2.46676
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.240e-15  8.266e-02   0.000   1.0000
## Factor1      1.396e-01  7.786e-02   1.792   0.0752 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9954 on 143 degrees of freedom
## Multiple R-squared:  0.02197,    Adjusted R-squared:  0.01513
## F-statistic: 3.213 on 1 and 143 DF,  p-value: 0.07518
```

2 Multidimensional scaling

2.1 Classical MDS

We illustrate an example of classical multimensional scaling using a dataset (`cerealnut.dta`) which consists of eight variables with nutrition data on 25 breakfast cereals. Data are available on Moodle.

```
#install.packages("haven")
library(haven)
cerealnut = read_dta("cerealnut.dta")
apply(cerealnut[,-1], 2, var)
```

```
##     calories      protein          fat           Na        fiber        carbs
##  454.0000000    1.7266667    0.5766667 5086.8333333    4.2291667   16.2291667
##        sugar            K
##   21.2500000 6006.9166667
```

Function `apply()` shows that `K`, `Na`, and `calories` - having much larger variances - will largely determine the Euclidean distances. Let's simply calculate the Euclidean distances with the default `dist()` command

```
cereal.dist = dist(cerealnut[,-1])
```

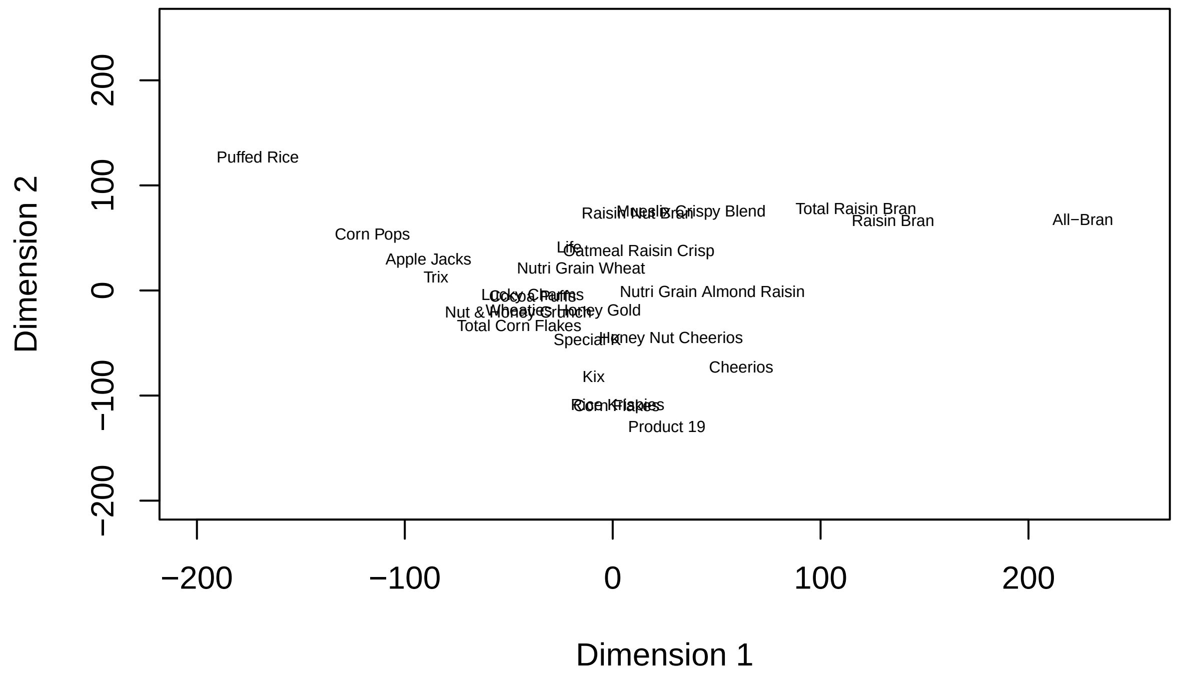
A procedure to find an MDS solution for a distance matrix from metric data is `cmdscale()`:

```
cereal.mds = cmdscale(cereal.dist)
cereal.mds
```

```
##             [,1]        [,2]
##  [1,]   61.827062  -72.553393
##  [2,]  -38.509435   -5.103719
##  [3,]   28.051534  -46.066717
##  [4,]   -9.169321  -81.494241
##  [5,]  -38.502389   -5.135628
##  [6,]   12.563541   37.089677
## [7,]   12.004028   73.780012
## [8,]  -44.982725  -33.250224
## [9,]  117.006685   77.996224
## [10,]  -85.003269   12.933042
## [11,]  -23.736733  -19.718219
## [12,]  226.179107   67.675191
## [13,]  -88.619865   28.432322
## [14,]    1.806912 -109.376969
## [15,] -115.536595   52.707239
## [16,]   37.744911   74.472656
## [17,]  -45.388641  -21.939256
## [18,]   47.944052   -0.608227
## [19,]  -15.226070   21.729050
## [20,]   26.087538 -129.479796
## [21,]  134.858669   66.725522
## [22,]    2.371034 -109.611540
## [23,]  -12.167038  -47.953993
## [24,]  -20.903633   41.451500
## [25,] -170.699361  127.299487
```

`cmdscale()` has performed classical metric scaling and extracted two dimensions, which is the default action. The result of `cmdscale()` is a list of two dimensions indicating coordinates for entities (in this case, cereals brands). Given those coordinates, we can simply `plot()` the values and label them:

```
rownames(cereal.mds) = cerealnut$brand
plot(cereal.mds, type = "n", xlab = "Dimension 1", ylab = "Dimension 2",
     xlim = c(-200, 250), ylim = c(-200, 250))
text(cereal.mds, rownames(cereal.mds), cex=0.5)
```



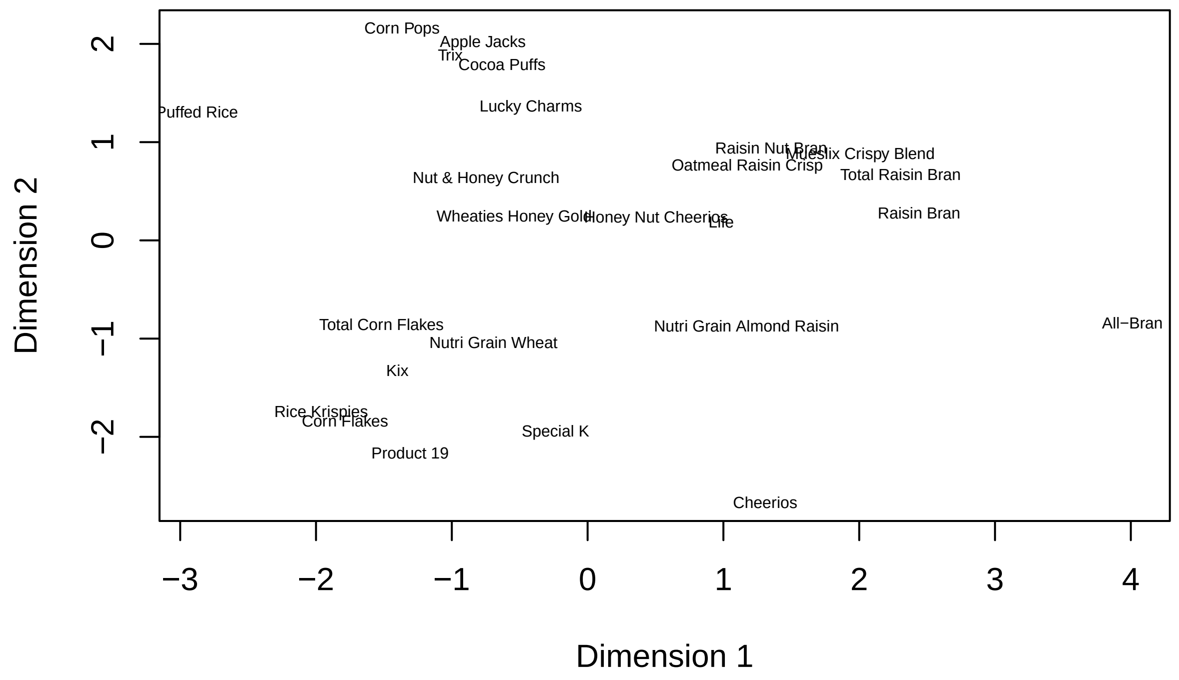
Classical MDS has placed the cereals so that all the brands fall within a triangle defined by Product 19, All-Bran, and Puffed Rice.

But, as we saw from the variable summary, three of the eight variables are controlling the distances. If we want to provide for a more equal footing for the eight variables, we compute MDS on standardized variables.

```
cerealnut.sc =scale(cerealnut[,-1])
```

We now compute classical MDS on the standardised variables:

```
cereal.sc.dist = dist(cerealnut.sc)
cereal.sc.mds = cmdscale(cereal.sc.dist)
rownames(cereal.sc.mds) = cerealnut$brand
plot(cereal.sc.mds, type = "n", xlab = "Dimension 1", ylab = "Dimension 2")
text(cereal.sc.mds, rownames(cereal.sc.mds), cex=0.5)
```



This configuration plot, based on the standardized variables, better incorporates all the nutrition data. If you are familiar with these cereal brands, spotting groups of similar cereals appearing near each other is easy. The bottom-left corner has several of the most sweetened cereals. The brands containing the word "Bran" all appear to the right of center. Rice Krispies and Puffed Rice are the farthest to the left.

2.2 Ordinal MDS

We investigate the use of non-metrical MDS using a simulated data set that is typical of consumer brand perception surveys. This data reflects consumer ratings of brands with regard to perceptual adjectives as expressed on survey items with the following form:

On a scale from 1 to 10 — where 1 is least and 10 is most — how [ADJECTIVE] is [BRAND A]?

In this data, an observation is one respondent's rating of a brand on one of the adjectives. Two such items might be:

1. How trendy is Intelligentsia Coffee?
2. How much of a category leader is Blue Bottle Coffee?

Such ratings are collected for all the combinations of adjectives and brands of interest.

The data here comprise simulated ratings of 10 brands ("a" to "j") on 9 adjectives ("performance", "leader", "latest", "fun", and so forth), for $N = 100$ simulated respondents. We start by loading and checking data:

```
brand.ratings = read.csv("http://goo.gl/IQl8nc")
head(brand.ratings)
```

```
##   perform leader latest fun serious bargain value trendy rebuy brand
## 1       2      4      8   8       2       9     7      4     6     a
## 2       1      1      4   7       1       1     1      2     2     a
## 3       2      3      5   9       2       9     5      1     6     a
## 4       1      6     10   8       3       4     5      2     1     a
## 5       1      1      5   8       1       9     9      1     1     a
## 6       2      8      9   5       3       8     7      1     2     a
```

```
tail(brand.ratings)
```

```
##      perform leader latest fun serious bargain value trendy rebuy brand
## 995        4      2      8   7       1       3     3      5     2     j
## 996        2      2      3   6       4       8     5      1     2     j
## 997        3      2      6   7       1       3     3      2     1     j
## 998        1      1     10  10       1       6     5      5     2     j
## 999        1      1      7   5       1       1     2      5     1     j
## 1000       7      4      7   8       4       1     2      5     1     j
```

Each of the 100 simulated respondents has observations on each of the 10 brands, so there are 1,000 total rows.

Perhaps the simplest business question in these data is: "What is the average (mean) position of the brand on each adjective?" We can use `aggregate()` to find the mean of each variable by brand:

```
brand.mean = aggregate(. ~ brand, data=brand.ratings, mean)
brand.mean
```

```
##    brand perform leader latest  fun serious bargain value trendy rebuy
## 1      a    1.65   3.04   7.46 7.87    1.77    4.83  4.78   3.78  2.21
## 2      b    7.47   7.21   8.43 3.40    7.61    4.37  4.70   7.25  4.33
## 3      c    6.57   7.45   5.88 3.75    7.72    2.64  3.28   5.29  3.39
## 4      d    2.31   2.87   7.28 6.58    2.40    1.91  2.10   7.24  2.47
## 5      e    2.68   4.92   7.60 6.88    4.44    5.73  5.34   5.60  3.82
## 6      f    4.30   5.12   2.31 5.47    5.96    6.59  6.79   2.99  7.18
## 7      g    7.43   3.98   2.24 4.65    2.84    6.65  7.35   1.72  7.19
## 8      h    4.44   3.64   7.74 8.03    3.93    2.29  2.46   7.59  2.19
## 9      i    5.56   3.58   7.29 7.20    3.91    3.58  2.41   6.84  3.21
## 10     j    2.47   2.36   5.72 6.85    2.65    4.00  4.16   3.90  1.28
```

Before proceeding, we perform a bit of housekeeping on the new `brand.mean` object. We name the rows with the brand labels that `aggregate()` put into the `brand` column, and then we remove that column as redundant:

```
rownames(brand.mean) = brand.mean[, 1] # use brand for the row names
brand.mean = brand.mean[, -1] # remove brand name column
```

The resulting matrix is now nicely formatted with brands by row and adjective means in the columns:

```
brand.mean
```

```
##   perform leader latest  fun serious bargain value trendy rebuy
## a    1.65   3.04   7.46 7.87    1.77    4.83  4.78   3.78  2.21
## b    7.47   7.21   8.43 3.40    7.61    4.37  4.70   7.25  4.33
## c    6.57   7.45   5.88 3.75    7.72    2.64  3.28   5.29  3.39
## d    2.31   2.87   7.28 6.58    2.40    1.91  2.10   7.24  2.47
## e    2.68   4.92   7.60 6.88    4.44    5.73  5.34   5.60  3.82
## f    4.30   5.12   2.31 5.47    5.96    6.59  6.79   2.99  7.18
## g    7.43   3.98   2.24 4.65    2.84    6.65  7.35   1.72  7.19
## h    4.44   3.64   7.74 8.03    3.93    2.29  2.46   7.59  2.19
## i    5.56   3.58   7.29 7.20    3.91    3.58  2.41   6.84  3.21
## j    2.47   2.36   5.72 6.85    2.65    4.00  4.16   3.90  1.28
```

Let's convert the mean ratings to rankings instead of raw values; this will be non-metric, ordinal data. We apply `rank()` to the columns using `lapply()` and code each resulting column as an ordinal factor variable using `ordered()`:

```
brand.rank = data.frame(lapply(brand.mean, function(x) ordered(rank(x))))
brand.rank
```

```
##    perform leader latest fun serious bargain value trendy rebuy
## 1        1      3      7   9       1       7     7      3     3
## 2       10      9     10   1       9       6     6      9     8
## 3        8     10      4   2      10       3     4      5     6
## 4        2      2      5   5       2       1     1      8     4
## 5        4      7      8   7       7       8     8      6     7
## 6        5      8      2   4       8       9     9      2     9
## 7        9      6      1   3       4      10    10      1    10
## 8        6      5      9  10       6       2     3     10     2
## 9        7      4      6   8       5       4     2      7     5
## 10       3      1      3   6       3       5     5      4     1
```

To find distances between the ranks, we use an alternative to `dist()`, `daisy()` from the `cluster` package, which can handle non-metric data such as rank ordering. In `daisy()`, we compute distance with the `gower` metric, which handles mixed numeric, ordinal, and nominal data:

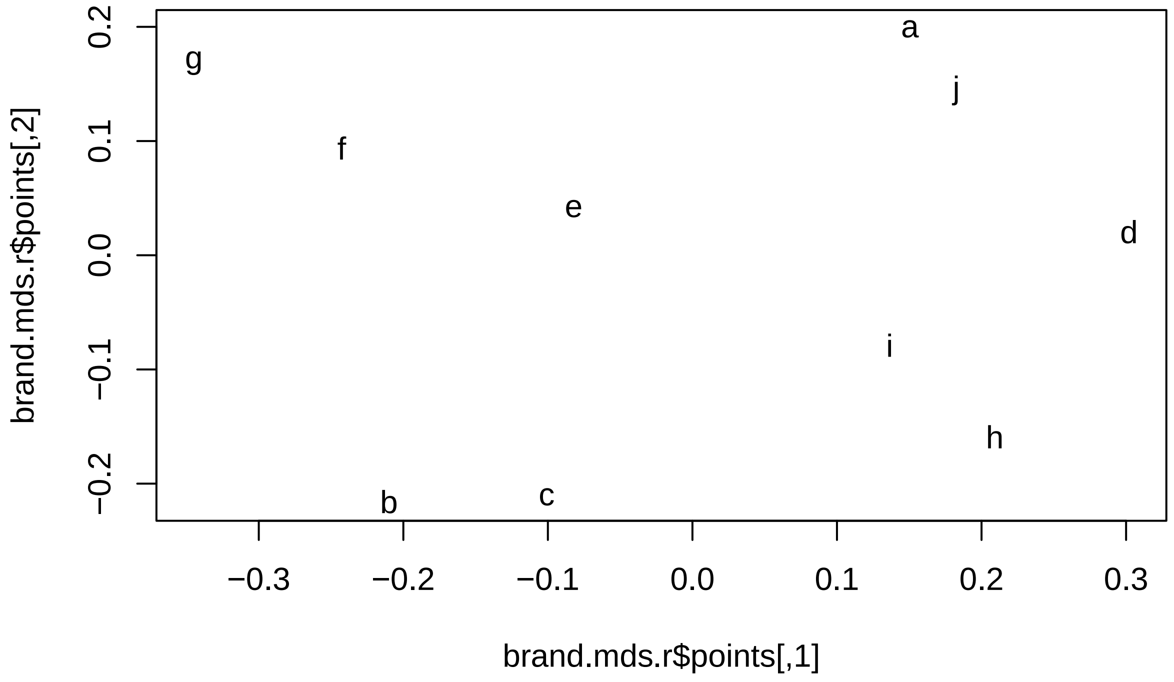
```
library(cluster)
brand.dist.r = daisy(brand.rank, metric="gower")
```

Now that we have a distance matrix we apply the non-metric MDS function `isoMDS()` from package `MASS` to the data. Then we plot the result:

```
library(MASS)
brand.mds.r = isoMDS(brand.dist.r)
```

```
## initial  value 9.063777
## iter   5 value 7.918224
## iter  10 value 7.772503
## final  value 7.655470
## converged
```

```
plot(brand.mds.r$points, type="n")
text(brand.mds.r$points, rownames(brand.mds.r), cex=1)
```


g
f
e
a
j
d
i
h
b
c
brand.mds.r$points[,1]
brand.mds.r$points[,2]