Clustering: k-means

The task: to group Starbucks beverages based on their nutritional characteristics The rationale: make Starbucks customers better aware of the Starbucks offer from health perspective

The algorithm to be used: k-means

Load and inspect the data

The data originates from the Nutrition facts for Starbucks Menu, where further information about it can be found (including the description of all the variables).

Note: the original data set has been adapted for this class.

Load the data

```r
sb_drinks = read.csv("data/starbucks.csv")
```

Inspect the data

```r
str(sb_drinks)
```

```
## 'data.frame':    242 obs. of  11 variables:
##  $ beverage_type: chr  "Coffee" "Coffee" "Coffee" "Coffee" ...
##  $ beverage_prep: chr  "Short" "Tall" "Grande" "Venti" ...
##  $ cholesterol  : int  0 0 0 0 5 15 0 5 25 0 ...
##  $ sodium       : int  5 10 10 10 75 85 65 120 135 105 ...
##  $ total_carbs  : int  0 0 0 0 10 10 6 15 15 10 ...
##  $ sugar        : int  0 0 0 0 9 9 4 14 14 6 ...
##  $ protein      : num  0.3 0.5 1 1 6 6 5 10 10 8 ...
##  $ total_fat    : num  0.1 0.1 0.1 0.1 0.1 3.5 2.5 0.2 6 4.5 ...
##  $ vitamine_A   : int  0 0 0 0 10 10 6 15 15 10 ...
##  $ calcium      : int  0 0 0 2 20 20 20 30 30 30 ...
##  $ iron         : int  0 0 0 0 0 0 8 0 0 15 ...
```

```r
summary(sb_drinks)
```

```
##  beverage_type      beverage_prep       cholesterol         sodium     
##  Length:242         Length:242         Min.   : 0.000   Min.   :  0.0  
##  Class :character   Class :character   1st Qu.: 0.000   1st Qu.: 70.0  
##  Mode  :character   Mode  :character   Median : 5.000   Median :125.0  
##                                        Mean   : 6.364   Mean   :128.9  
##                                        3rd Qu.:10.000   3rd Qu.:170.0  
##                                        Max.   :40.000   Max.   :340.0  
##   total_carbs        sugar          protein          total_fat    
##  Min.   : 0.00   Min.   : 0.00   Min.   : 0.000   Min.   : 0.000  
##  1st Qu.:21.00   1st Qu.:18.00   1st Qu.: 3.000   1st Qu.: 0.200  
##  Median :34.00   Median :32.00   Median : 6.000   Median : 2.250
##  Mean   :35.99   Mean   :32.96   Mean   : 6.979   Mean   : 2.898
##  3rd Qu.:50.75   3rd Qu.:43.75   3rd Qu.:10.000   3rd Qu.: 4.500
##  Max.   :90.00   Max.   :84.00   Max.   :20.000   Max.   :15.000
##    vitamine_A        calcium           iron
##  Min.   : 0.000   Min.   : 0.00   Min.   : 0.000
##  1st Qu.: 4.000   1st Qu.:10.00   1st Qu.: 0.000
##  Median : 8.000   Median :20.00   Median : 2.000
##  Mean   : 9.831   Mean   :20.76   Mean   : 7.446
##  3rd Qu.:15.000   3rd Qu.:30.00   3rd Qu.:10.000
##  Max.   :50.000   Max.   :60.00   Max.   :50.000
```

```r
apply(sb_drinks[, c(1,2)], 2, unique)
```

```
## $beverage_type
## [1] "Coffee"                            "Classic Espresso Drinks"
## [3] "Signature Espresso Drinks"         "Tazo® Tea Drinks"
## [5] "Shaken Iced Beverages"             "Smoothies"
## [7] "Frappuccino® Blended Coffee"       "Frappuccino® Light Blended Coffee"
## [9] "Frappuccino® Blended Crème"
##
## $beverage_prep
##  [1] "Short"              "Tall"               "Grande"
##  [4] "Venti"              "Short Nonfat Milk"  "2% Milk"
##  [7] "Soymilk"            "Tall Nonfat Milk"   "Grande Nonfat Milk"
## [10] "Venti Nonfat Milk"  "Solo"               "Doppio"
## [13] "Whole Milk"
```

Data preparation

To decide how to approach data preparation, we first need to consider how the algorithm that we intend to use - k-means - works with the input data. In short, k-means is based on measuring distances (e.g. Euclidean) between instances and computing mean values of the features that describe the instances (for a detailed explanation of the algorithm, see the lecture slides). This implies the following:

- all features must be numerical
- there should be no outliers

- all features should have the same or similar value ranges

So, we need to:

- select numerical variables, including those that can be transformed to numerical
- check if outliers are present and if so, replace them with less extreme values
- check if all features have the same value ranges and rescale them if their ranges differ

1) Select numerical variables

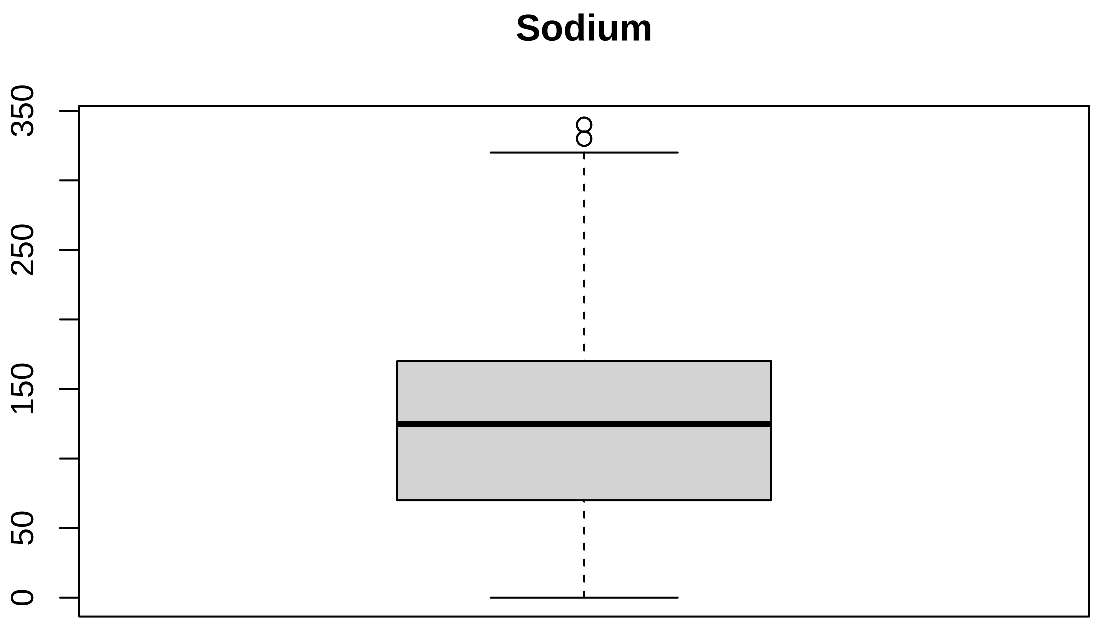
The first two variables are categorical and cannot be used for k-means clustering, hence we use the remaining 9 variables. Note: we will later use these categorical variables to better understand the clusters

```
sb_numeric = sb_drinks[,c(3:11)]
```

2) Check for outliers and handle them, if they are present

We'll start by checking how many outliers each variable has. We'll first do it visually for a couple of variables, to get an idea how outliers can be identified and then will show a more efficient way to do it. While there are different ways for identifying outliers, a frequently applied one is based on the use of boxplot to represent the distribution of a variable, in which case outliers are detected as values that are above the upper whisker or below the bottom whisker - see this annotated boxplot. Let's try it out on one of our variables:

```
boxplot(sb_numeric$sodium, main = "Sodium")
```



The above boxplot indicate that both `sodium` has a couple of extremely high values (dots above the upper whisker).

We will check the same for all variables in a more efficient way, using the `boxplot.stats` function:

```
?boxplot.stats
```

We'll first try it out

```
boxplot.stats(sb_numeric$sodium)
```

```
## $stats
## [1]   0  70 125 170 320
```

```
##
## $n
## [1] 242
##
## $conf
## [1] 114.8434 135.1566
##
## $out
## [1] 330 340 340 330
```

We'll now use `boxplot.stats` to create a function that determines the number of outliers for the given variable

```
check_outliers = function(x) {
  if(length(boxplot.stats(x)$out) == 0)
   return("No outliers")

  top_whisker = boxplot.stats(x)$stats[5]
  bottom_whisker = boxplot.stats(x)$stats[1]
  n_upper_outliers = sum(boxplot.stats(x)$out > top_whisker)
  n_bottom_outliers = sum(boxplot.stats(x)$out < bottom_whisker)

  return(paste0("Upper outliers: N=", n_upper_outliers,
              "; Botton outliers: N=", n_bottom_outliers))
}

apply(sb_numeric, 2, check_outliers) |> as.data.frame()
```

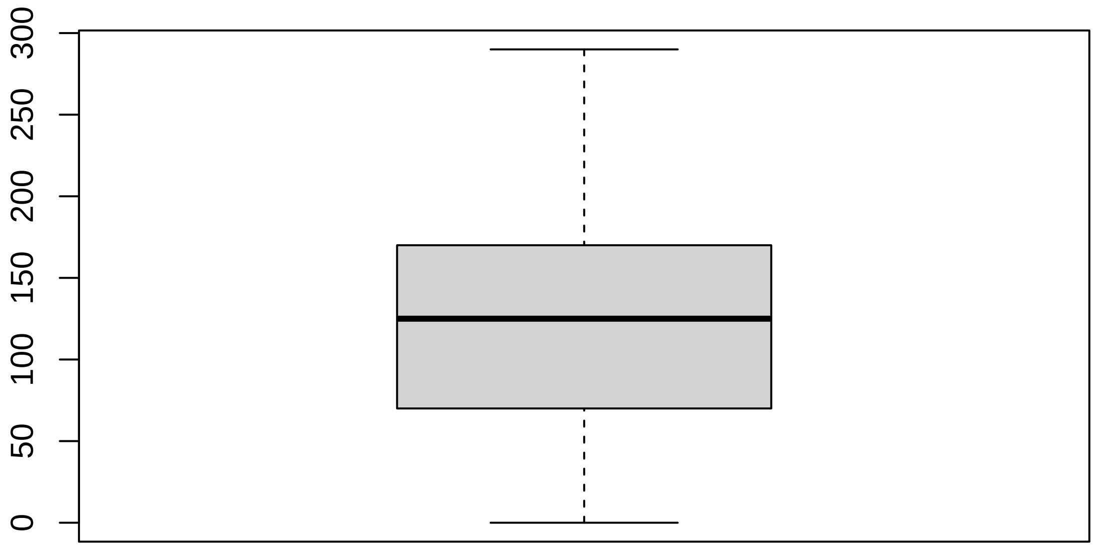
```
##                    apply(sb_numeric, 2, check_outliers)
## cholesterol  Upper outliers: N=9; Botton outliers: N=0
## sodium       Upper outliers: N=4; Botton outliers: N=0
## total_carbs                                No outliers
## sugar        Upper outliers: N=2; Botton outliers: N=0
## protein                                    No outliers
## total_fat    Upper outliers: N=5; Botton outliers: N=0
## vitamine_A   Upper outliers: N=3; Botton outliers: N=0
## calcium                                    No outliers
## iron        Upper outliers: N=17; Botton outliers: N=0
```

As a way of dealing with outliers, we'll use the Winsorizing technique. It is a simple technique that consists of replacing extreme values with a specific percentile of the data, typically 95th for overly high values and 5th percentile for overly low values. For seamless application of this technique to our data, we will use the `Winsorize` function from the *DescTools* package.

```
library(DescTools)

?Winsorize
```

Let's start with `sodium` which we've already examined above:

```
sb_numeric$sodium = Winsorize(sb_numeric$sodium,
                              quantile(sb_numeric$sodium, probs = c(0,0.95)))

boxplot(sb_numeric$sodium)
```



We'll do the same for the other variables with outliers:

```
apply(sb_numeric[,c("cholesterol", "sugar","total_fat", "vitamine_A", "iron")],
      2,
      function(x) Winsorize(x, quantile(x, probs=c(0, 0.95)))) |>
  as.data.frame() -> sb_sub_trimmed
```

Let's check if we have resolved the problem with outliers

```
apply(sb_sub_trimmed, 2, check_outliers) |> as.data.frame()
```

```
##               apply(sb_sub_trimmed, 2, check_outliers)
## cholesterol                                No outliers
## sugar                                      No outliers
## total_fat                                  No outliers
## vitamine_A                                 No outliers
## iron        Upper outliers: N=17; Botton outliers: N=0
```

The iron variable still has outliers -> we need to use a lower percentile for the replacement of extreme values

Let's examine `iron` outliers more closely:

```
boxplot.stats(sb_numeric$iron)$out |> sort(decreasing = T)
```

```
##  [1] 50 50 40 40 40 35 35 35 30 30 30 30 30 30 30 30 30
```

We need to find the percentile that will allow for replacing all values equal or higher than 30

```r
quantile(sb_numeric$iron, probs = seq(0.9, 1, 0.01))
```

```
##   90%   91%   92%   93%   94%   95%   96%   97%   98%   99%  100%
## 25.00 25.00 25.00 25.65 30.00 30.00 30.00 33.85 35.90 40.00 50.00
```

```r
iron_w = Winsorize(sb_numeric$iron, quantile(sb_numeric$iron, probs = c(0,0.92)))
boxplot.stats(iron_w)$out
```

```
## numeric(0)
```

Now, it's OK. What is left is to replace the existing values of the variables with outliers, with Winsorized version of those variables

```r
sb_numeric[,c("sodium", "total_carbs", "protein", "calcium")] |> cbind(sb_sub_trimmed) -> sb_numeric
sb_numeric$iron = iron_w
```

3) Scaling, if needed

```r
apply(sb_numeric, 2, range) |> t()
```

```
##             [,1] [,2]
## sodium         0  290
## total_carbs    0   90
## protein        0   20
## calcium        0   60
## cholesterol    0   25
## sugar          0   71
## total_fat      0    9
## vitamine_A     0   25
## iron           0   25
```

Since value ranges of the variables vary, rescaling is needed. As we have resolved outliers, we can do the re-scaling through normalisation

```r
normalise <- function(var) {
  (var - min(var, na.rm = T))/(max(var, na.rm = T) - min(var,na.rm = T))
}

apply(sb_numeric, 2, normalise) |> as.data.frame() -> sb_norm
```

```r
summary(sb_norm)
```

```
##      sodium        total_carbs        protein          calcium
##  Min.   :0.0000   Min.   :0.0000   Min.   :0.0000   Min.   :0.0000
##  1st Qu.:0.2414   1st Qu.:0.2333   1st Qu.:0.1500   1st Qu.:0.1667
##  Median :0.4310   Median :0.3778   Median :0.3000   Median :0.3333
##  Mean   :0.4403   Mean   :0.3999   Mean   :0.3489   Mean   :0.3459
##  3rd Qu.:0.5862   3rd Qu.:0.5639   3rd Qu.:0.5000   3rd Qu.:0.5000
##  Max.   :1.0000   Max.   :1.0000   Max.   :1.0000   Max.   :1.0000  
##   cholesterol         sugar          total_fat         vitamine_A    
##  Min.   :0.0000   Min.   :0.0000   Min.   :0.00000   Min.   :0.0000  
##  1st Qu.:0.0000   1st Qu.:0.2535   1st Qu.:0.02222   1st Qu.:0.1600  
##  Median :0.2000   Median :0.4507   Median :0.25000   Median :0.3200  
##  Mean   :0.2413   Mean   :0.4597   Mean   :0.31327   Mean   :0.3792  
##  3rd Qu.:0.4000   3rd Qu.:0.6162   3rd Qu.:0.50000   3rd Qu.:0.6000  
##  Max.   :1.0000   Max.   :1.0000   Max.   :1.00000   Max.   :1.0000  
##       iron       
##  Min.   :0.0000  
##  1st Qu.:0.0000  
##  Median :0.0800  
##  Mean   :0.2698  
##  3rd Qu.:0.4000  
##  Max.   :1.0000
```

Feature pruning to avoid high correlations

The presence of highly correlated variables in the data set can negatively affect clustering, so that patterns detected in the data may not be the true ones. Therefore, we need to examine correlations among the variables and remove those that are highly correlated with other variables.

```
library(corrplot)
```

```
## corrplot 0.95 loaded
```

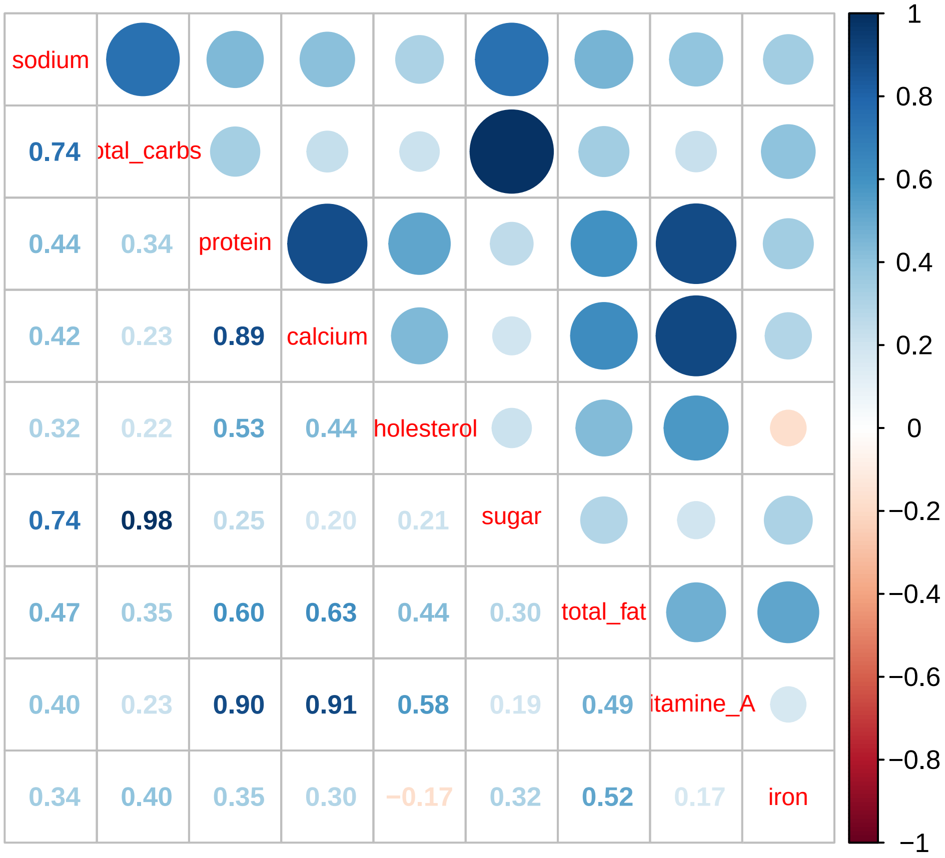
Check if the variables have Normal distribution, to determine which correlation coefficient to compute

```
apply(sb_norm, 2, function(x) shapiro.test(x)$p.value > 0.05)
```

```
##      sodium total_carbs     protein     calcium cholesterol       sugar 
##       FALSE       FALSE       FALSE       FALSE       FALSE       FALSE 
##   total_fat  vitamine_A        iron 
##       FALSE       FALSE       FALSE
```

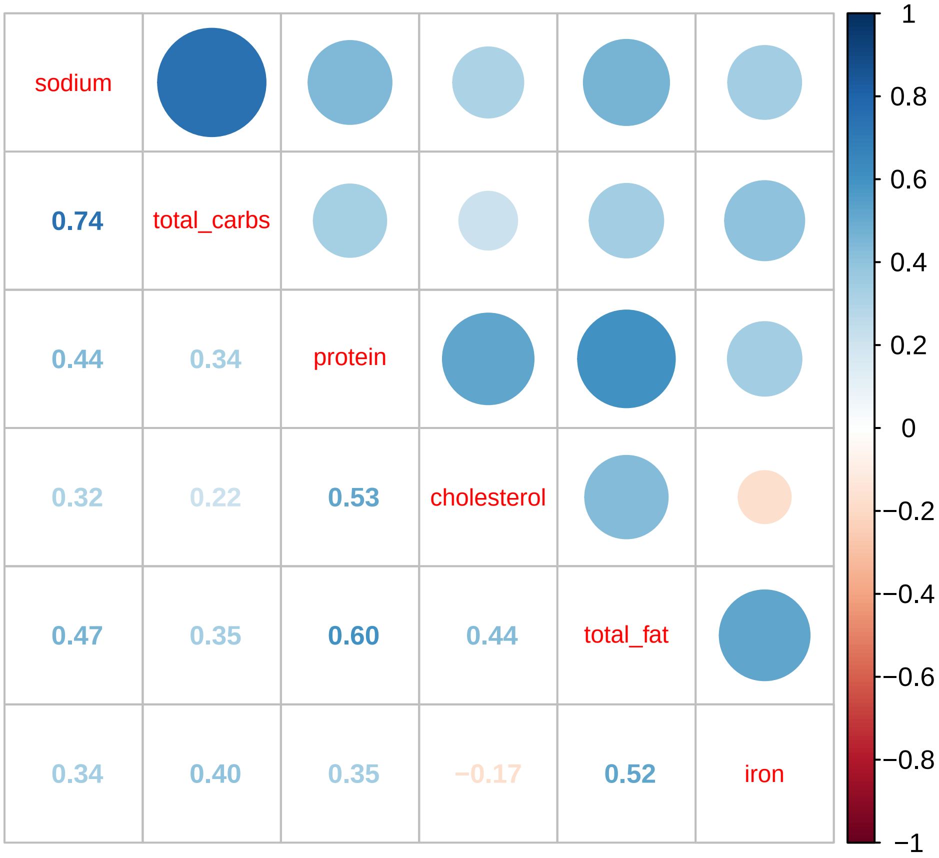
Since none of the variables is normally distributed, we'll compute Spearman correlation coefficient

```
sb_cor = cor(sb_norm, method = "spearman")
corrplot.mixed(sb_cor, tl.cex=0.75, number.cex=0.85)
```



We’ll remove `sugar`, `calcium`, and `vitamine_A`, due to high correlation with other variables Notes: 1) The correlation of 0.74 between `sodium` and `total_carbs` is at a marginal value (0.75 is typically used as the threshold), but we’ll keep these variables for now 2) After we identify the clusters, we can use the variables that we are now putting aside, to interpret and better understand the clusters

```
to_remove = which(colnames(sb_norm) %in% c("sugar", "calcium", "vitamine_A"))
sb_final = sb_norm[,-to_remove]
sb_cor = cor(sb_final, method = "spearman")
corrplot.mixed(sb_cor, tl.cex=0.75, number.cex=0.85)
```



Clustering

Now that we have prepared the data, we can use it as the input to the k-means algorithm, to do the clustering.

Recall that for k-means we need to set the number of clusters as one of the algorithm's hyper parameters. To get a first glimpse at how the algorithm works when applied to the data, we will make an "informed guess" regarding the number of clusters based on the general awareness of the kinds of drinks that Starbucks offers. For example, we can expect 4 clusters:

- one formed of different kinds of regular coffee and tea;
- another of coffees and teas with various kinds of additions;
- the third could be regular soft drinks, and
- the forth one, a variety of customised soft drinks.

Run the k-means algorithm, we will use the *kmeans* function

```
?kmeans
```

```
set.seed(2024)
km_4k <- kmeans(x = sb_final, centers=4, iter.max=20, nstart=1000)
```

Notes:

- Since the initial cluster centers (centroids) are randomly selected, we set the seed to assure replicability of the results.

- As k-means is sensitive to the initial selection of cluster centers, we use the nstart hyper-parameter to indicate that we want to re-run the initial selection many (1000) times (to eventually select the best one).

Examine the results:

```
km_4k
```

```
## K-means clustering with 4 clusters of sizes 39, 63, 91, 49
##
## Cluster means:
##      sodium total_carbs   protein cholesterol total_fat       iron
## 1 0.5870911   0.4900285 0.5423077   0.8153846 0.7165242 0.20102564
## 2 0.6324576   0.5259259 0.3626984   0.1777778 0.1391534 0.13587302
## 3 0.1794240   0.2163614 0.1839560   0.1142857 0.1244200 0.08879121
## 4 0.5608726   0.5070295 0.4836735   0.1020408 0.5668934 0.83265306
##
## Clustering vector:
##   [1] 3 3 3 3 3 3 3 3 1 4 2 1 4 2 1 4 3 3 4 4 1 4 4 1 4 4 1 4 3 3 3 3 1 4 2 1 4
##  [38] 2 1 4 3 3 3 3 3 3 3 3 3 3 3 1 3 3 1 4 3 3 3 3 2 2 3 3 3 3 1 3 2 1 4 2 1 4
##  [75] 2 1 3 2 1 4 1 1 4 1 1 4 3 1 4 4 1 4 4 1 4 4 1 4 3 3 3 3 3 3 3 3 3 3 3 3 1
## [112] 3 2 1 3 2 1 4 3 1 3 2 1 4 2 1 4 2 1 4 3 3 3 3 3 3 3 1 3 2 1 4 3 3 3 3 3 3
## [149] 3 1 3 2 1 4 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 4 4 4 2 2 2 2 2 2 2 2 2
## [186] 2 2 2 1 2 2 2 2 2 2 4 2 1 4 2 2 2 2 2 2 2 1 2 4 4 4 4 4 4 4 4 4 3 2 2 3 2
## [223] 2 3 2 2 4 4 4 2 2 2 2 2 2 2 1 2 2 2 2 2
##
## Within cluster sum of squares by cluster:
## [1] 12.38843 11.46262 12.62957 12.49064
##  (between_SS / total_SS =  58.8 %)
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss"
## [6] "betweenss"    "size"         "iter"         "ifault"
```

Among other things, the output of the kmeans f. includes the following evaluation metrics:

- **within-cluster sum of squares** (within_SS): The sum of squared differences between individual data points in a cluster and the cluster center (centroid). It is computed for each cluster separately; the smaller, the better.
- **total sum of squares** (total_SS): The sum of squared differences of each data point from the global sample mean; this represents the overall measure of dispersion; the smaller, the better
- **between cluster sum of squares** (between_SS): The sum of squared differences between each cluster center (centroid) and the global sample mean; when computing this value, for each cluster center, the squared difference is multiplied by the number of data points in that cluster (to account for the cluster size); the higher this value, the better
- **between_SS / total_SS**: This ratio indicates how well the sample splits into clusters; the higher the ratio, the better the clustering. The maximum is 1.

Now that we have seen what clustering using the k-means function looks like, we can do it "properly", by first choosing the optimal value for K and then running k-means with the chosen value.

Selecting the optimal value for K There are different approaches to determining optimal value for K, among which the so-called *Elbow method* is probably the most widely used one.

The *Elbow method* is based on the sum of squared distances between data points and cluster centers, that is, the sum of *within-cluster sum of squares* (`within_SS`) for all the clusters (`tot.withinss`)

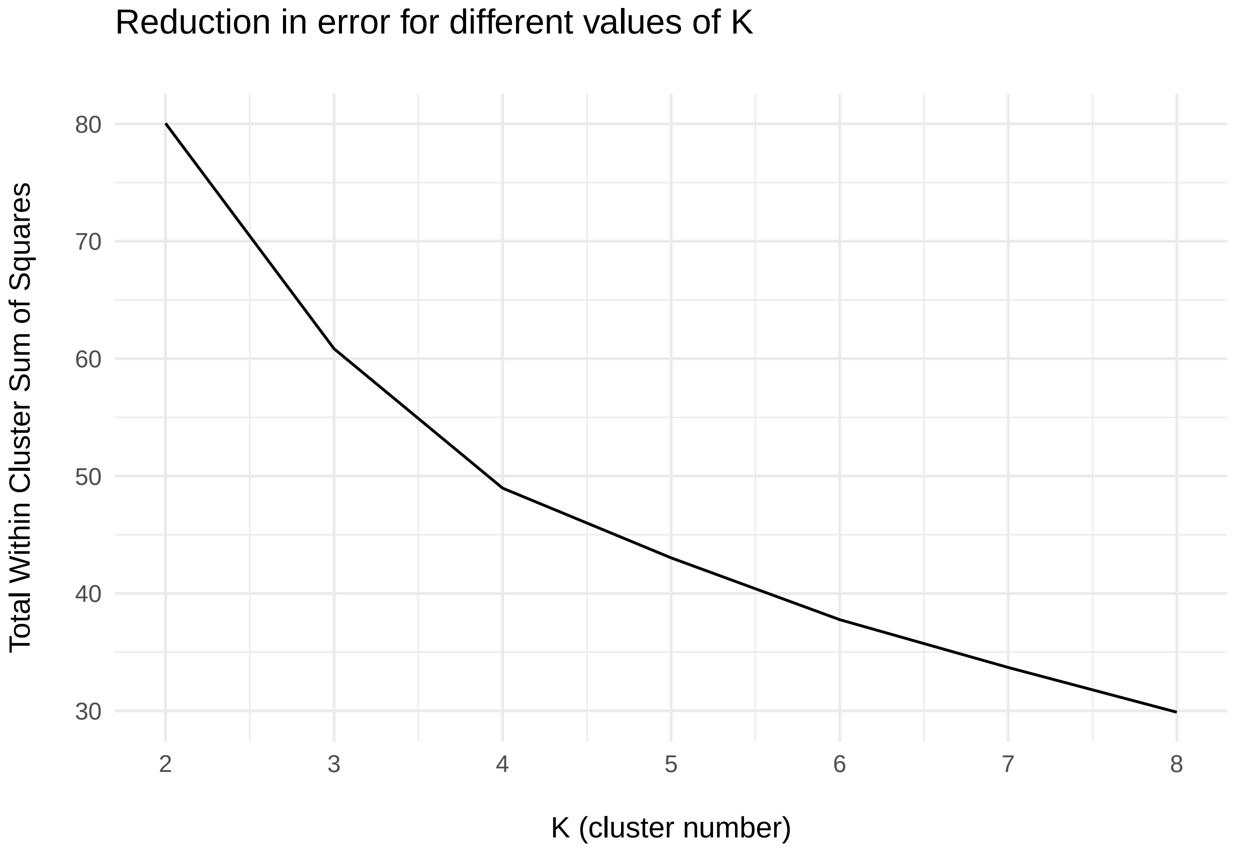
To apply the Elbow method, we will run k-means for different K values (from 2 to 8). For each value of K, we will compute the metric required for the Elbow method namely `tot.withinss`. Along the way, we'll also compute another useful metric - *ratio of between_SS and total_SS*.

```r
eval_metrics = data.frame()

for (k in 2:8) {
  set.seed(2024)
  km_res = kmeans(x=sb_final, centers=k, iter.max=20, nstart = 1000)
  eval_metrics = rbind(eval_metrics,
                       c(k, km_res$tot.withinss, km_res$betweenss/km_res$totss))
}

names(eval_metrics) <- c("k", "tot.within.ss", "ratio")

eval_metrics
```

```
##   k tot.within.ss     ratio
## 1 2      80.05072 0.3262844
## 2 3      60.84067 0.4879583
## 3 4      48.97126 0.5878526
## 4 5      43.03245 0.6378342
## 5 6      37.76970 0.6821261
## 6 7      33.69680 0.7164041
## 7 8      29.88639 0.7484729
```

Draw the Elbow plot based on the computed `tot.within.ss` measure

```r
library(ggplot2)

ggplot(data=eval_metrics, aes(x=k, y=tot.within.ss)) +
  geom_line() +
  labs(x = "\nK (cluster number)",
       y = "Total Within Cluster Sum of Squares\n",
       title = "Reduction in error for different values of K\n") +
  theme_minimal() +
  scale_x_continuous(breaks=seq(from=0, to=8, by=1))
```



To choose optimal value for K, we examine points at the plot where the line “breaks”, forming something that may resemble a bent human elbow. In this case, there are two such points: for k=3 and k=4. We will examine both and choose the one that is more “meaningful”, that is, the one that is easier for interpretation and eventually more useful (e.g., for recommending drinks to customers).

Since we have already done the clustering with k=4, we just need to do the clustering with k=3

```r
set.seed(2024)
km_3k <- kmeans(x = sb_final, centers=3, iter.max=20, nstart=1000)
km_3k
```

```
## K-means clustering with 3 clusters of sizes 55, 73, 114
##
## Cluster means:
##      sodium total_carbs   protein cholesterol total_fat       iron
## 1 0.5570533   0.5070707 0.5045455   0.1927273 0.6020202 0.84363636
## 2 0.6152574   0.5080670 0.4636986   0.5369863 0.4280061 0.09753425
## 3 0.2719298   0.2789474 0.2003509   0.0754386 0.1004873 0.10315789
##
## Clustering vector:
##   [1] 3 3 3 3 3 2 3 3 2 1 3 2 1 2 2 1 3 3 1 1 1 1 1 1 1 1 1 1 3 2 3 3 2 1 3 2 1
##  [38] 2 2 1 3 3 3 3 3 3 3 3 2 3 3 2 3 3 2 1 3 3 3 3 3 2 3 2 3 3 2 3 2 2 1 2 2 1
##  [75] 3 2 3 2 2 1 2 2 1 2 2 1 3 2 1 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3 3 3 2
## [112] 3 3 2 3 3 2 1 3 2 3 3 2 1 2 2 1 2 2 1 3 3 3 3 2 3 3 2 3 3 2 1 3 3 3 3 2 3
## [149] 3 2 3 3 2 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 1 1 1 2 2 2 2 2 3 2 3 3
## [186] 2 3 2 2 2 3 2 3 3 2 1 2 2 1 3 2 3 3 2 3 2 2 2 1 1 1 1 1 1 1 1 1 3 3 2 3 3
## [223] 2 3 3 2 1 1 1 3 2 3 3 2 3 2 2 2 3 2 3 2
```

```
##
## Within cluster sum of squares by cluster:
## [1] 17.46999 25.47072 17.89996
##  (between_SS / total_SS =  48.8 %)
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss"
## [6] "betweenss"    "size"         "iter"         "ifault"
```

We will now interpret and compare the two clustering solutions.

In the output printed above, we can observe the difference in objective metrics of cluster quality. These can also be accessed and examined more directly:

```
data.frame(k = c(rep(3, 3), rep(4, 4)),
           within_ss = c(km_3k$withinss, km_4k$withinss))
```

```
##   k within_ss
## 1 3  17.46999
## 2 3  25.47072
## 3 3  17.89996
## 4 4  12.38843
## 5 4  11.46262
## 6 4  12.62957
## 7 4  12.49064
```

```
eval_metrics[eval_metrics$k %in% c(3,4),]
```

```
##   k tot.within.ss     ratio
## 2 3      60.84067 0.4879583
## 3 4      48.97126 0.5878526
```

Clearly, based on these metrics, the solution with k=4 is better. Note: this is expected since these metrics of cluster quality tend to improve as the number of clusters increases.

More important is to interpret the clusters from the perspective of the features that describe the instances, and compare the values of those features across the clusters. To that end, we will use some auxiliary functions for computing summary statistics of feature values in individual clusters as well as visual comparisons of feature distributions across the clusters.

```
source("clustering_util.R")
```

If any of the features that were excluded due to high correlation with other features, is considered relevant for better understanding of the identified clusters, we can also use them for cluster interpretation and comparison. For example, since the ultimate intention is to identify clusters of Starbucks drinks based on their nutritional values and enable Starbucks customers to make health-conscious decision of their drinks, we can opt for the following set of features:

```
f_selection = which(colnames(sb_norm) %in% c("cholesterol", "sodium", "sugar", "protein", "total_fat"))
```

Let's start with the 4 clusters solution and explore summary statistics (mean and SD) of the feature selection we've made

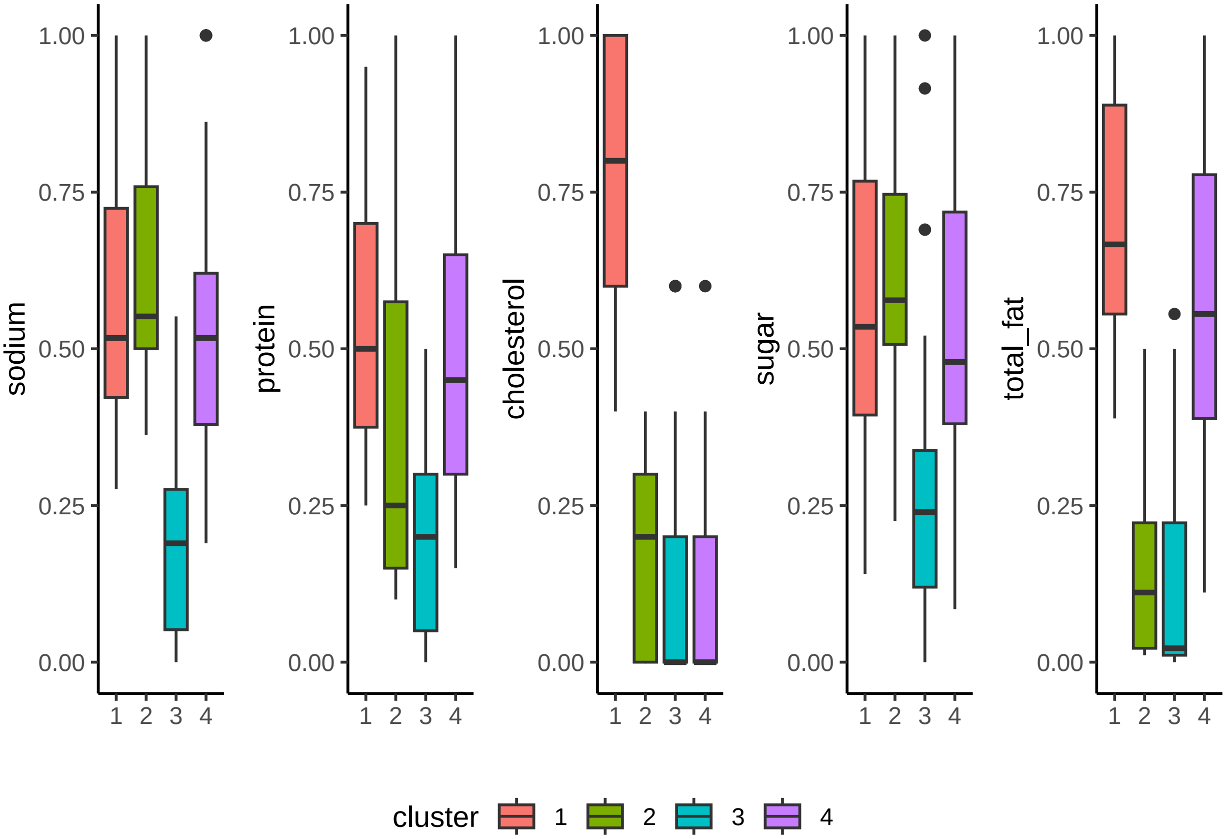
```
# note: for this function to work, you'll need to install the following R packages: dplyr, tidyr, and s
summary_stats(sb_norm[,f_selection], km_4k$cluster)
```

```
## Loading required package: dplyr

##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union

## Loading required package: tidyr

## Loading required package: stringr

##                    CLUST_1       CLUST_2       CLUST_3       CLUST_4
## N                       39            63            91            49
## sodium      0.587 (0.225) 0.632 (0.182) 0.179 (0.134) 0.561 (0.228)
## protein       0.542 (0.2) 0.363 (0.253) 0.184 (0.138) 0.484 (0.218)
## cholesterol 0.815 (0.197) 0.178 (0.157) 0.114 (0.177) 0.102 (0.164)
## sugar       0.576 (0.256)   0.62 (0.19) 0.255 (0.191) 0.542 (0.248)
## total_fat   0.717 (0.199) 0.139 (0.134) 0.124 (0.154) 0.567 (0.237)
```

Due to the high SD of almost all feature-cluster combinations, mean values can be misleading. Therefore, it is better to examine and compare the overall distribution of features across the clusters:

```
# note: for this function to work, you'll need to install the ggpubr package
create_comparison_plots(sb_norm[,f_selection], km_4k$cluster, ncol = length(f_selection))
```

```
## Loading required package: ggpubr
```



We can observe:

- More even distribution of observations (drinks) across the clusters, compared to the solution w/ 3 clusters
- Cluster 1 is exceeding the other two in terms of total fat and cholesterol, and has moderate to high values of the other three nutrients
- Cluster 2 has moderate to high values of sodium and sugar, low values of cholesterol and total fat, and low to moderate values of protein
- Cluster 3 has the lowest values of all the nutrients
- Cluster 4 has moderate values of sodium, sugar, protein, and total fat, and low cholesterol values

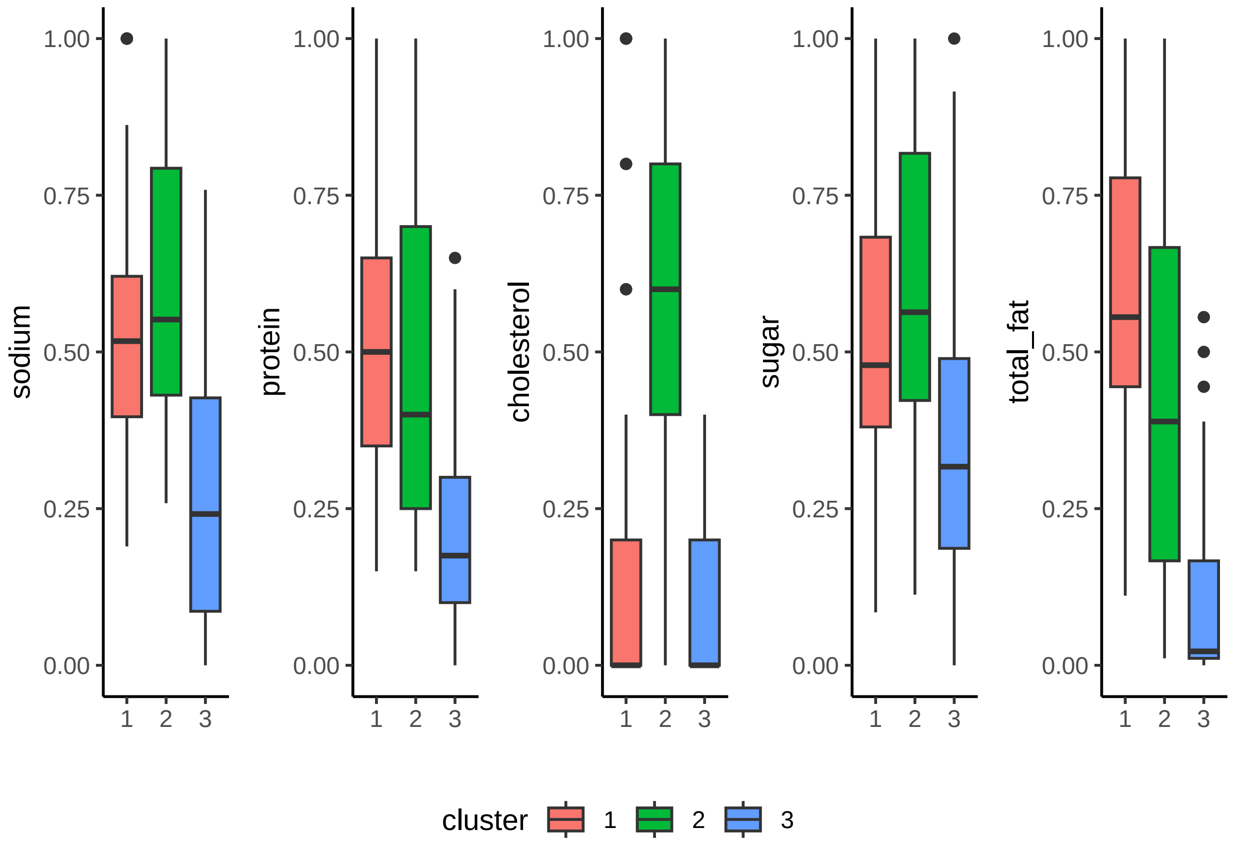
From health perspective, probably the best option are drinks from Cluster 3, while the least desirable ones are those from Cluster 1. If it wasn't for moderate to high values of sugar and total fat, Cluster 4 would also be a healthy option.

Let's now do the same for the 3-cluster solution

```
summary_stats(sb_norm[,f_selection], km_3k$cluster)
```

```
##                     CLUST_1       CLUST_2       CLUST_3
## N                        55            73           114
## sodium      0.557 (0.218) 0.615 (0.239) 0.272 (0.216)
## protein     0.505 (0.219) 0.464 (0.243)   0.2 (0.153)
## cholesterol 0.193 (0.305) 0.537 (0.298) 0.075 (0.117)
## sugar       0.541 (0.237)  0.596 (0.26) 0.333 (0.227)
## total_fat   0.602 (0.249) 0.428 (0.291)   0.1 (0.131)
```

```r
create_comparison_plots(sb_norm[,f_selection], km_3k$cluster, ncol = length(f_selection))
```



We can observe:

- Cluster 1 is low in cholesterol, moderate in sodium and protein, and moderate to high n sugar and total fat
- Cluster 2 is comparable to Cluster 1 in sodium, protein, and sugar, and is somewhat lagging behind in total fat; however, it is far above the other two clusters in terms of cholesterol
- Cluster 3 has the lowest values of all the nutrients

All in all, the solution with 4 clusters is the better both from the perspective of the cluster quality measures and the interpretation / “meaningfulness” perspective.

Let’s also check the distribution of drink preparation variants (the `beverage_prep` variable) in each cluster (of the 4-cluster solution) since drink preparation may help explain the observed difference in nutrient values across the clusters

```r
table(drink_types = sb_drinks$beverage_prep, clust = km_4k$cluster) |> prop.table(margin = 2) |> round(
```

```
##                       clust
## drink_types                1     2     3     4
##   2% Milk              0.846 0.032 0.154 0.020
##   Doppio               0.000 0.000 0.011 0.000
##   Grande               0.000 0.000 0.077 0.000
##   Grande Nonfat Milk   0.026 0.254 0.044 0.102
##   Short              0.000 0.000 0.044 0.000
##   Short Nonfat Milk  0.000 0.016 0.121 0.000
##   Solo               0.000 0.000 0.011 0.000
##   Soymilk            0.000 0.206 0.231 0.653
##   Tall               0.000 0.000 0.077 0.000
##   Tall Nonfat Milk   0.000 0.111 0.132 0.082
##   Venti              0.000 0.000 0.077 0.000
##   Venti Nonfat Milk  0.026 0.238 0.022 0.082
##   Whole Milk         0.103 0.143 0.000 0.061
```

All drinks in cluster 1 are prepared either w/ 2% [fat] milk or with whole milk, which can account for the high values of total fat, cholesterol, protein… On the other hand, drinks in the 3rd cluster (the most "healthy" one) are prepared primarily with non-fat milk and soy milk. The dominance of soy milk among the preparation methods of cluster 4 can explain (at least to an extent) the nutrient values observed in that cluster (barista soy milk is not a healthy choice).