Principal Component Analysis

S.Tonini, F. Chiaromonte (special thanks to J. Di Iorio and L. Insolia)

February 16th 2023

Contents

Introduction

Working in high-dimensional spaces can be undesirable for several reasons so it could be useful to project the features to a space of fewer dimensions.

Principal Component Analysis (PCA) is a *Dimension Reduction* technique for unsupervised problems. It projects/transforms the data into a lower-dimensional space so that the lower-dimensional representation retains some meaningful properties of the original data, ideally close to its intrinsic dimension.

Libraries

We are going to use:

- **mvtnorm**: *Multivariate Normal and t Distributions*

- **factoextra**: *Extract and Visualize the Results of Multivariate Data Analyses*
- **scales**: *Scale Functions for Visualization*
- **ellipse**: *Functions for drawing ellipses and ellipse-like confidence regions*
- **corrplot**: *Visualization of a Correlation Matrix*

```
library(mvtnorm)    # for the toy simulated example
library(factoextra) # contains also decathlon data
library(scales)     # to create ggplot-like figures in base R (color transparency)
library(ellipse)    # to add elliptical confidence regions in base R plots
library(corrplot)   # correlation plots
```

Data

Today we are going to use the **decathlon2** data set, which is available in the **factoextra** package. It consists of 27 observations (athletes) and the following 13 variables (performance).

```
help(decathlon2)
head(decathlon2)
```

```
##           X100m Long.jump Shot.put High.jump X400m X110m.hurdle Discus
## SEBRLE    11.04      7.58    14.83      2.07 49.81        14.69  43.75
## CLAY      10.76      7.40    14.26      1.86 49.37        14.05  50.72
## BERNARD   11.02      7.23    14.25      1.92 48.93        14.99  40.87
## YURKOV    11.34      7.09    15.19      2.10 50.42        15.31  46.26
## ZSIVOCZKY 11.13      7.30    13.48      2.01 48.62        14.17  45.67
## McMULLEN  10.83      7.31    13.76      2.13 49.91        14.38  44.41
##           Pole.vault Javeline X1500m Rank Points Competition
## SEBRLE          5.02    63.19  291.7    1   8217    Decastar
## CLAY            4.92    60.15  301.5    2   8122    Decastar
## BERNARD         5.32    62.77  280.1    4   8067    Decastar
## YURKOV          4.72    63.44  276.4    5   8036    Decastar
## ZSIVOCZKY       4.42    55.37  268.0    7   8004    Decastar
## McMULLEN        4.42    56.37  285.1    8   7995    Decastar
```

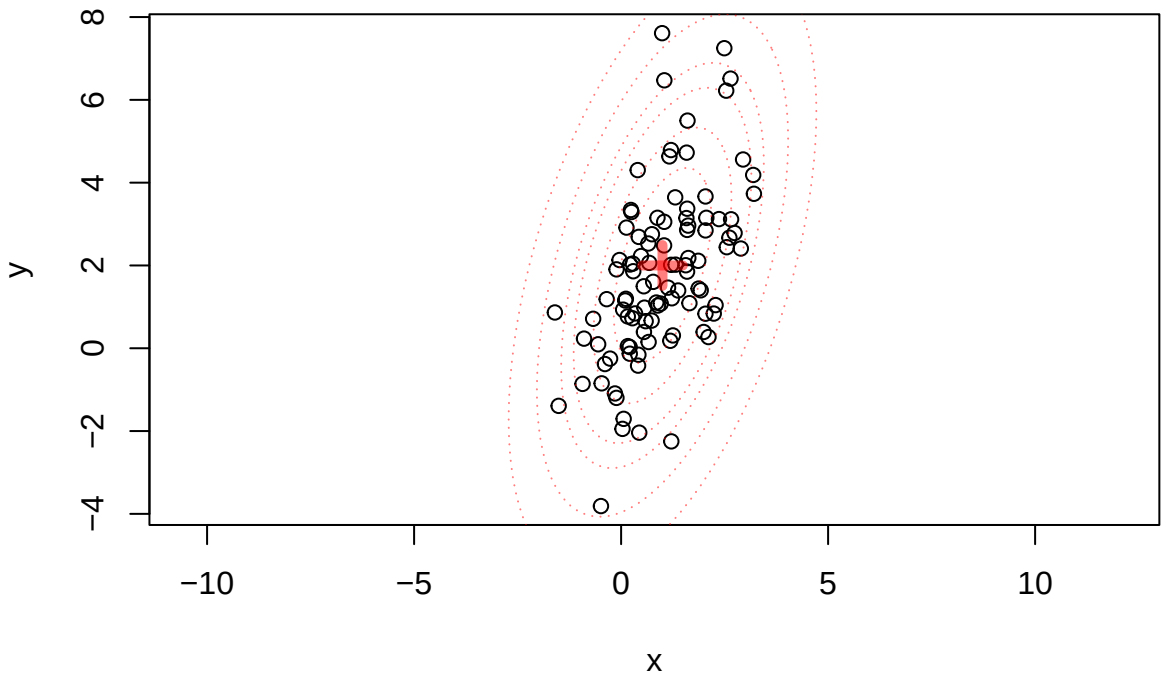
A Toy Example of Dimension Reduction

Let's simulate a 2-dimensional data set from a Gaussian distribution with higher dispersion along the y-axis.

```
set.seed(1234)                # set seed for reproducibility
mu  <- c(1, 2)                # location/mean vector (p times 1)
sig <- cbind(c(1,1), c(1,4))  # covariance matrix (p times p)
n   <- 100                    # number of points

X <- rmvnorm(n, mu, sig)      # data generation
colnames(X) <- c("x","y")     # renaming
head(X)                       # visualize our data
```

```
##                x           y
## [1,] -0.03816118  2.13170559
## [2,]  1.21209499 -2.24884418
## [3,]  1.57692190  3.14455298
## [4,]  0.27238812  0.72545111
## [5,]  0.16401245  0.05239442
## [6,]  0.20871312 -0.13108549
```

```r
plot(X, asp=1)                # plot our data (with fixed aspect ratio)

points(mu[1], mu[2],          # add centroid
       col=alpha("red", 0.5), # marker color and transparency
       pch=3,                 # marker type
       lwd=5,                 # marker width
       cex=2)                 # marker size

# iterate across confidence levels
conflev <- c(0.5, 0.75, 0.9, 0.95, 0.99, 0.999)
for (confi in conflev){
  lines(ellipse(sig, centre=mu, level=confi),    # add elliptical confidence regions
        col = alpha("red", 0.5),                 # add color and transparency
        lty = 3)                                 # dashed lines
}
```



How can we reduce the dimension of these data?

We can reduce the number of features (columns) in different ways. Let's see some.

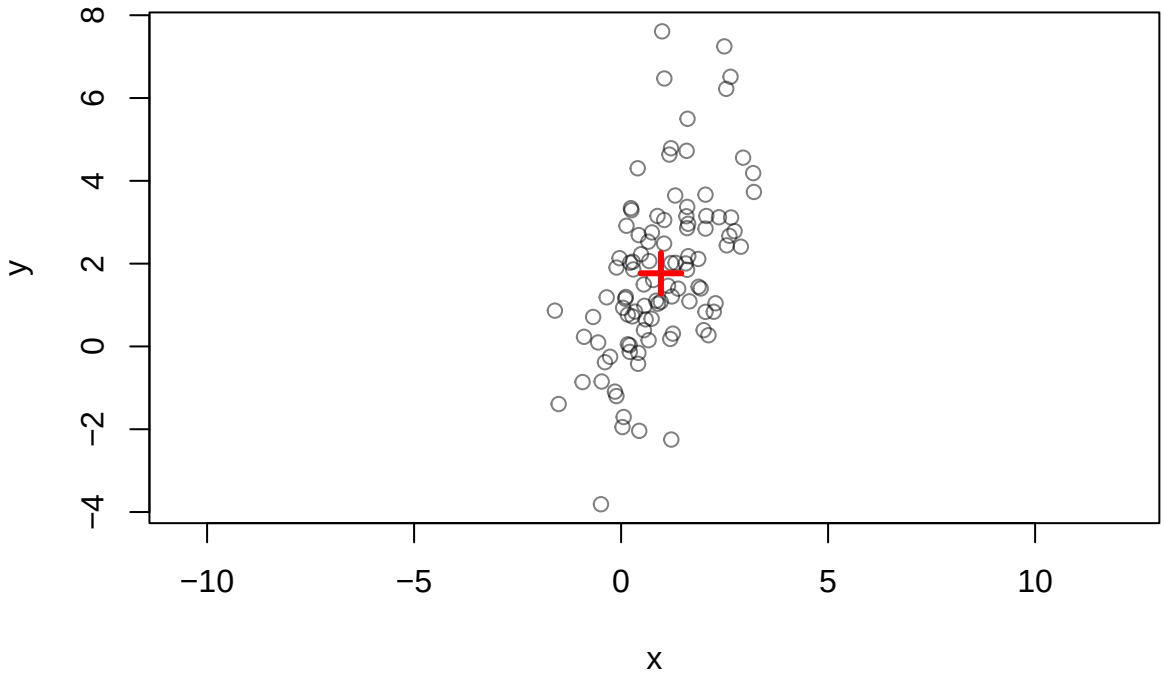
Using the sample mean

The sample mean is the easiest 0-dimensional data reduction method, because it allows to collapse all the information into a single point.

```
med <- colMeans(X)              # column-wise means
med
```

```
##         x         y
## 0.9623671 1.7667355
```

```
plot(X, asp=1,                  # plotting the points
     col=alpha('black', 0.5))   # reduce the transparency

points(med[1], med[2],          # plotting their centroid
       col='red', pch=3,        # color, marker type
       cex = 2, lwd=3)          # size, width
```



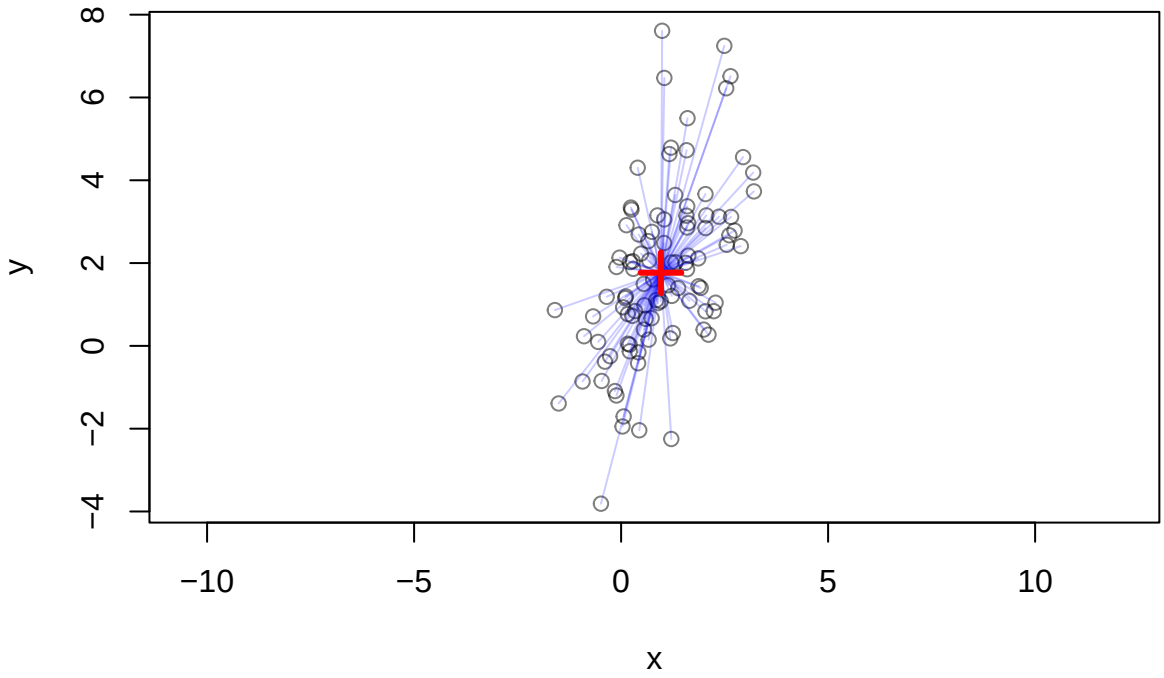
What is the variance?

What is the associated loss of information?

What is the error?

We can think of the error in this way:

```
plot(X, asp=1,                    # plotting the points
     col=alpha('black', 0.5))     # reduce the transparency

for(i in 1:n){                    # for each point
  lines(rbind(X[i,], med),        # plot the "error"
        col=alpha('blue',0.2))
}

points(med[1], med[2],            # plotting their centroid
       col='red', pch=3,          # color, marker type
       cex = 2, lwd=3)            # size, width
```



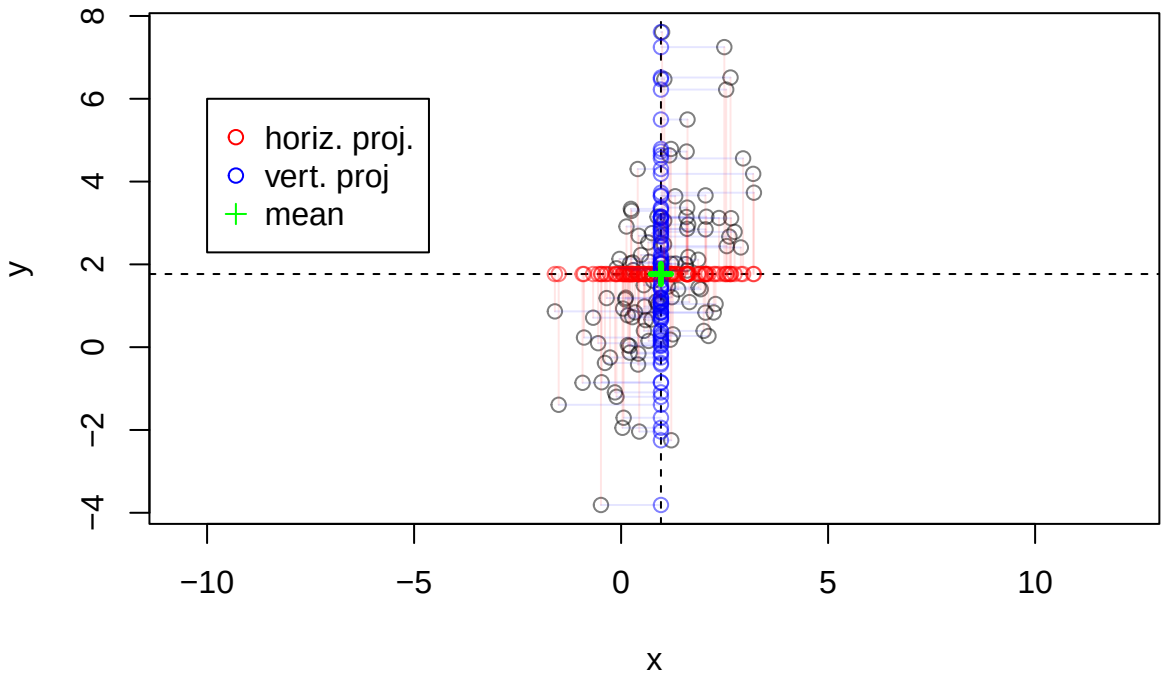
We are collapsing our data to one point, the sample mean (also known as PC0). The error is high.

Projecting the data on a new axis

We can easily identify two axes from the sample mean: an *horizontal*-axis, and a *vertical*-axis.

```
plot(X, asp=1, col=alpha('black', 0.5))     # data

abline(h=med[2], lty=2)                     # horizontal axis
points(X[,1], rep(med[2], n),               # projected points
       col=alpha('red',0.5))

abline(v=med[1], lty=2)                     # vertical axis
```

```
points(rep(med[1], n), X[,2],                # projected points
       col=alpha('blue',0.5))

for(i in 1:n){
  lines(rbind(X[i,], c(X[i,1],med[2])),      # plot the horiz. proj. "error"
        col=alpha('red',0.1))

  lines(rbind(X[i,], c(med[1], X[i,2])),     # plot the vert. proj. "error"
    col=alpha('blue',0.1))
}

points(med[1], med[2], col='green',          # centroid
       pch=3, cex = 1, lwd=3)

legend(-10,6,                                # legend location
       c("horiz. proj.","vert. proj","mean"),
       col=c("red","blue","green"),
       pch=c(1,1,3))
```




```
# comparing variances
var(X[,1])                                   # red dots variance
```

```
## [1] 1.092818
```

```
var(X[,2])                                    # blue dots variance
```

```
## [1] 4.22376
```

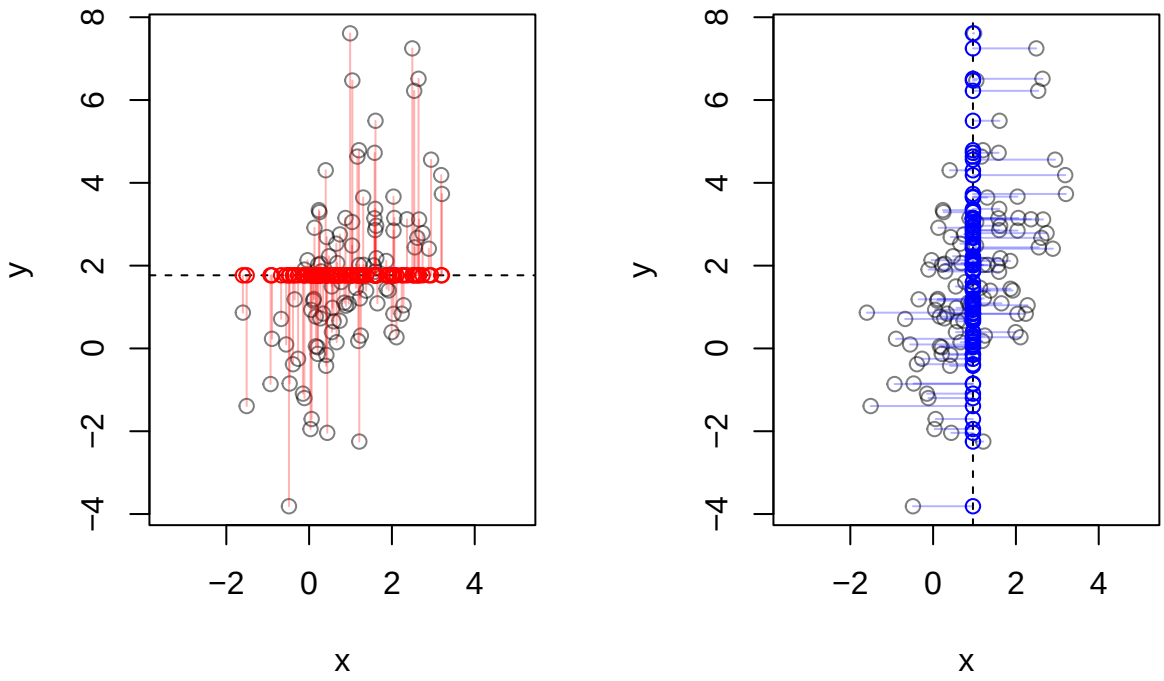
Which of the two axis maximizes the variance?

```
var(X[,2]) > var(X[,1])
```

```
## [1] TRUE
```

The vertical axis. Blue points are more scattered and the error (sum of the lenghts of blue segments) is lower.

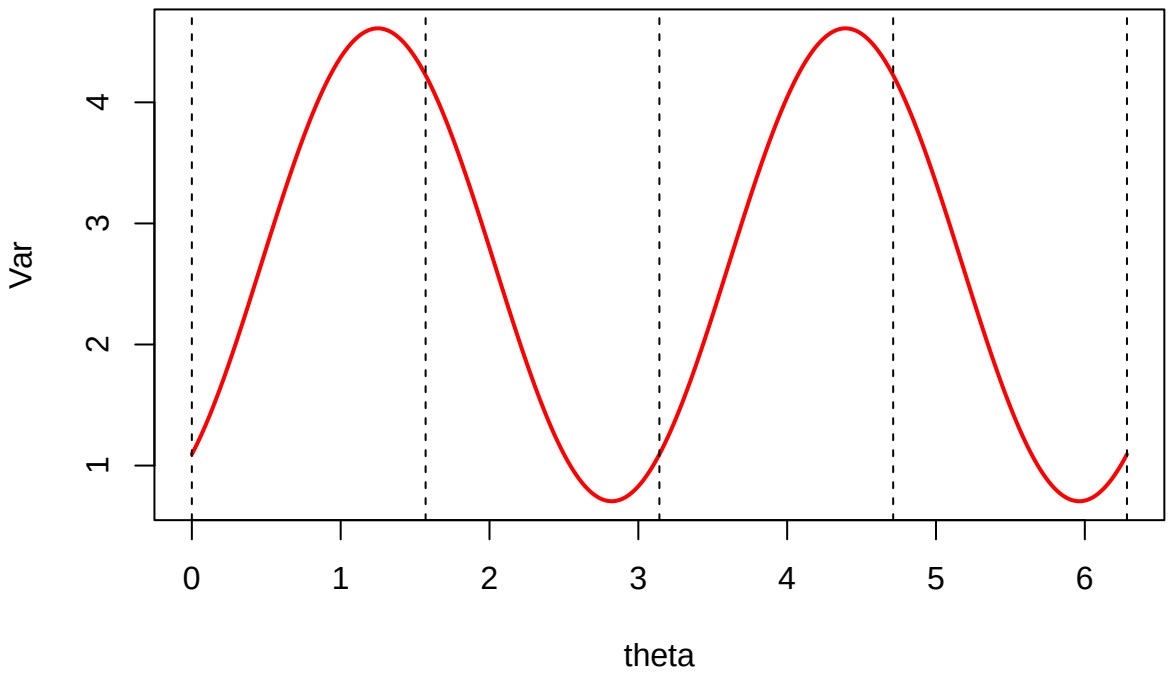
```
# 2 times 1 panels in a figure
par(mfrow=c(1,2))

# left panel: horiz. axis
plot(X, asp=1, col=alpha('black', 0.5))
abline(h=med[2], lty=2)
points(X[,1], rep(med[2], n), col='red')
for(i in 1:n)
  lines(rbind(X[i,], c(X[i,1], med[2])), col=alpha('red',0.3))

# right panel: vert. axis
plot(X, asp=1, col=alpha('black', 0.5))
abline(v=med[1], lty=2)
points(rep(med[1], n), X[,2], col='blue')
for(i in 1:n)
  lines(rbind(c(med[1],X[i,2]),X[i,]), col=alpha('blue',0.3))
```




```
# reset it as 1 x 1
par(mfrow=c(1,1))
```

Using this strategy we can find the "best" axis – i.e., the one maximizing the variance. Let's try a brute force approach.

```
# Compute the variance across (almost) all possible directions
theta   <- seq(0, 2*pi, by = 2*pi/360)     # angles
Var     <- rep(NA, length(theta))

# centered data
Xc = scale(X, scale=F)

for(i in 1:length(theta))
{
  a      <- c(cos(theta[i]), sin(theta[i]))  # directional vector
  proj   <- Xc %*% a                         # projecting points
  Var[i] <- var(proj)                        # computing variance
}

# plotting the variance for each direction/angle
plot(theta, Var, type = 'l',
     col='red', lwd = 2)
abline(v=c(0, pi/2, pi, 3/2*pi, 2*pi),       # fundamental angles
       lty=2)
```



The direction with highest variability is identified by the maximun of the function.

```
max.var   <- max(Var) # maximum variance
# max.theta <- theta[which.max(Var)] # theta angle with maximum variance (only the 1st)
max.theta <- theta[Var==max.var] # theta angle with maximum variance (not only the 1st)
max.theta
```

```
## [1] 1.256637 4.398230
```

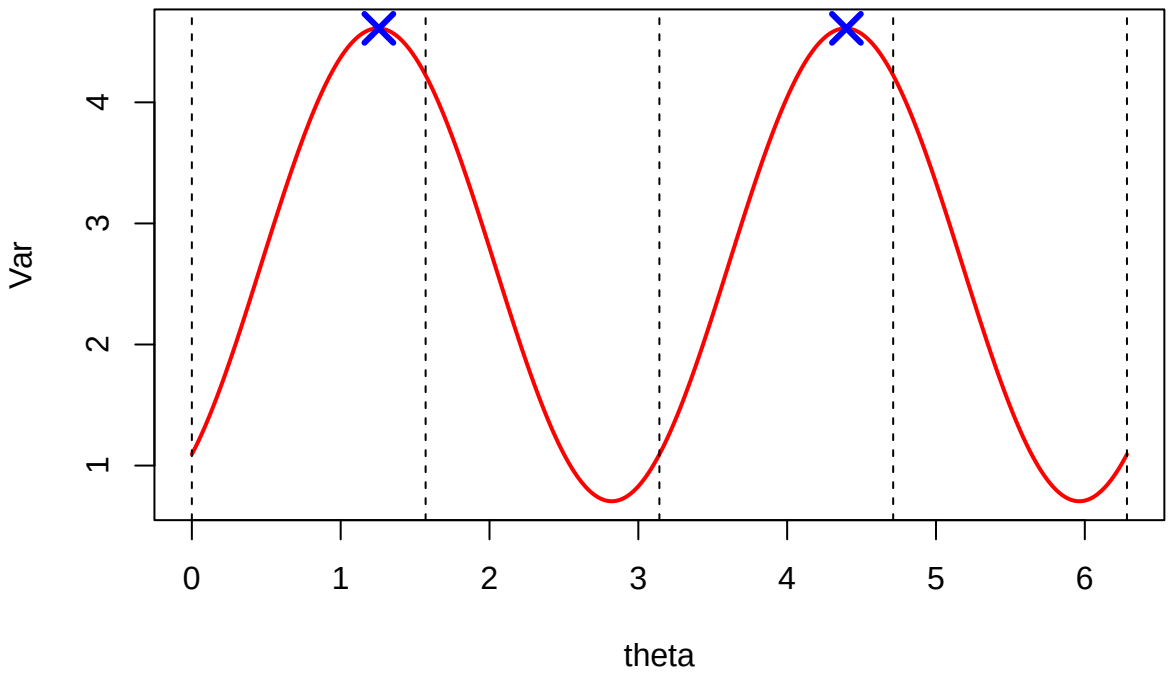
```
# projected data (see that their variance is equal to the maximum)
projx <- X %*% c(cos(max.theta[1]), sin(max.theta[1]))
var(projx)
```

```
##          [,1]
## [1,] 4.611209
```

```
max.var
```

```
## [1] 4.611209
```

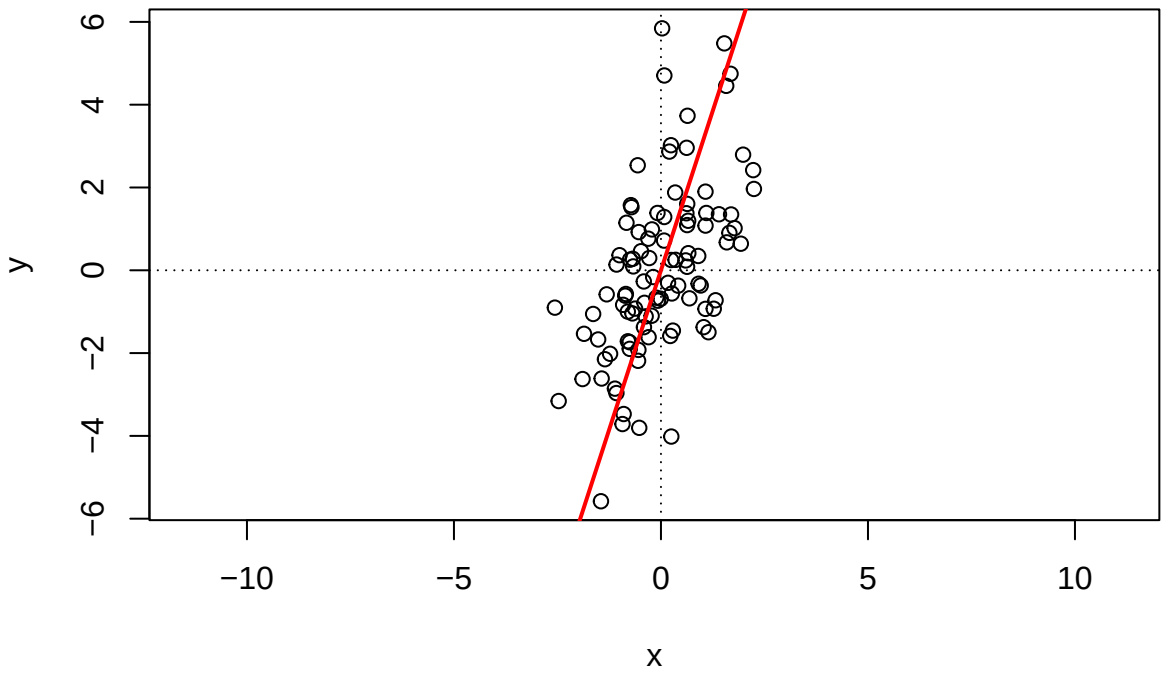
```
# plotting the Variance for each direction/angle
plot(theta, Var, type = 'l', col='red', lwd = 2, lty=1)
abline(v=c(0, pi/2, pi, 3/2*pi, 2*pi), lty=2) # fundamental angles
points(max.theta, rep(max.var, length(max.theta)),
       pch=4, col='blue', lwd=3, cex=2)
```



We just found the first principal component (PC1) as the axis maximizing the variance. Let's plot the solution we have found.

Remark: there is rotational invariance, so we can pick any maximizer theta (e.g., the first one).

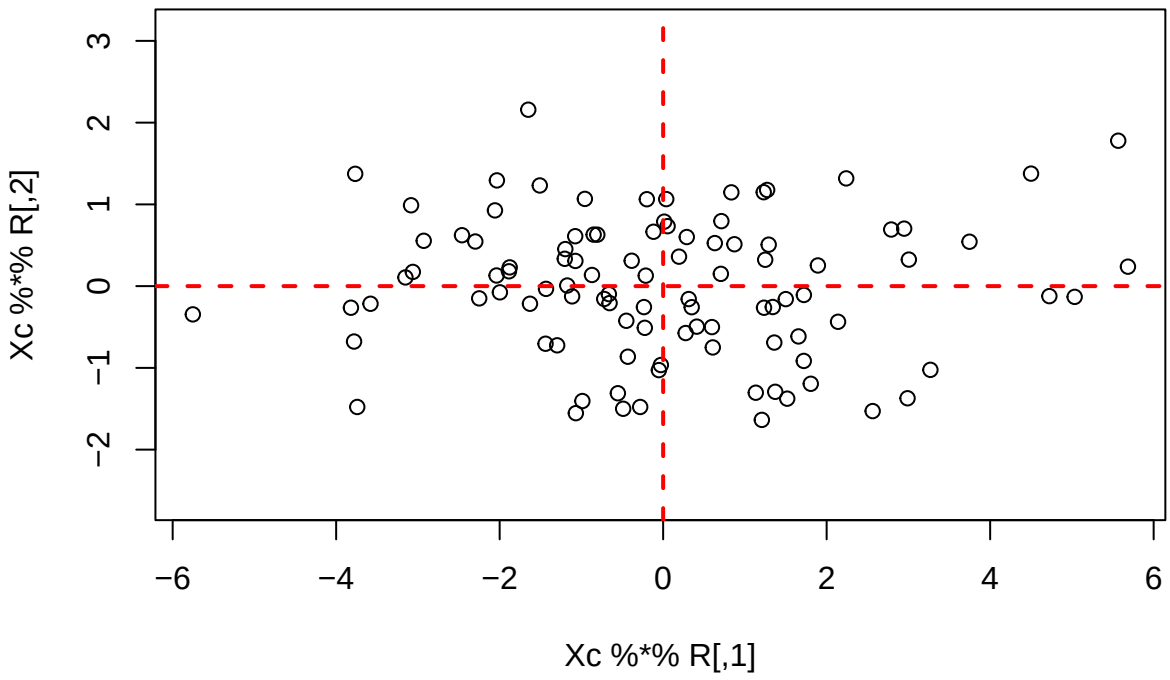
```
# slope
slopepc1 <- sin(max.theta[1])/cos(max.theta[1])

# plot centered data
plot(Xc, asp=1, col='black')
abline(v=0, lty=3)
abline(h=0, lty=3)
# plot PC1
abline(a=0,b=slopepc1, col = "red", lty=1, lwd=2)
```



Note that there is only another component orthogonal to PC1 (and centered at 0) since we are in a 2-dimensional setting.

Thus, we can also view PCA as a change of bases/rotation.

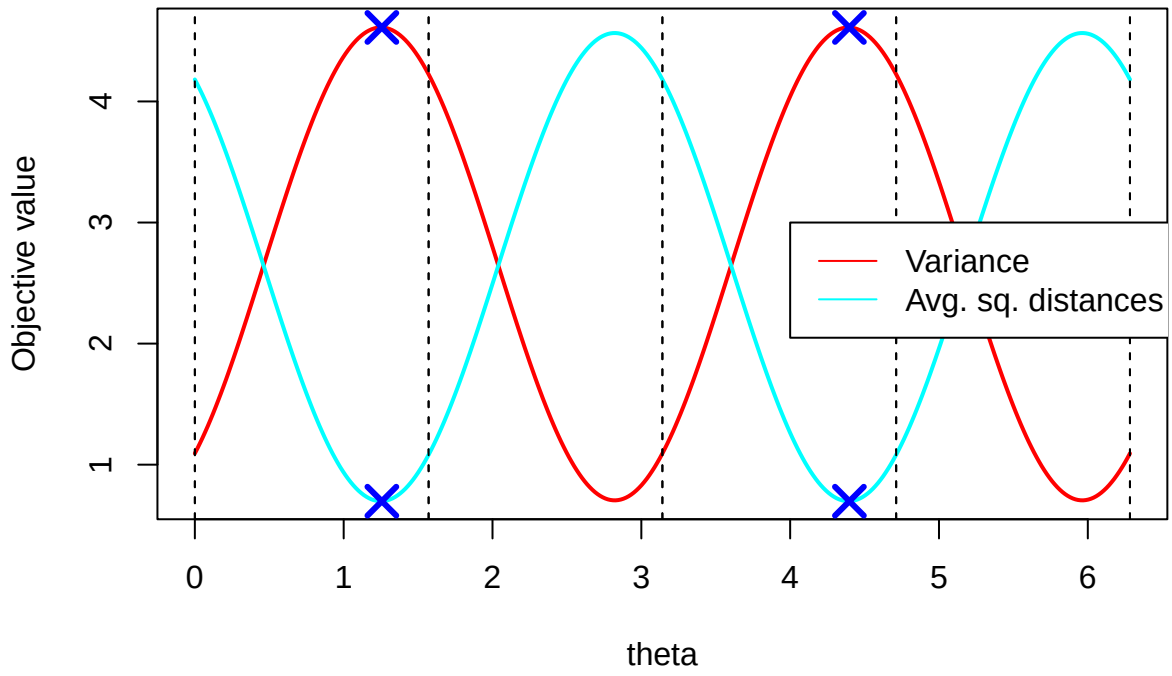
```
# rotation matrix
R1 <- c(cos(max.theta[1]), -sin(max.theta[1]))
R2 <- c(sin(max.theta[1]), cos(max.theta[1]))
R <- rbind(R1, R2)

# plot rotated data such that PC1 and PC2 are now the "standard" bases
plot(Xc %*% R, asp=1)
abline(v=0, col="red",lty=2, lwd=2)
abline(h=0, col="red",lty=2, lwd=2)
```



Finally, notice that maximizing the variance is equivalent to minimizing squared distances.

```
# Compute the squared distances across (almost) all possible directions
theta2   <- seq(0, 2*pi, by = 2*pi/360) # angles
Dist     <- rep(NA, length(theta))

for(i in 1:length(theta))               # for each angle theta
{
  mi = sin(theta2[i])/cos(theta2[i])    # slope of the associated line
  distk = rep(NA, n)                    # initialize point-to-line distances
  for (k in 1:n){                       # compute them
    distk[k] = abs(-mi*Xc[k,1]+Xc[k,2])/sqrt(mi^2+1)
  }
  Dist[i] = mean(distk^2)               # store average of squared distances
}

# min and argmin for the distance method
min.dist   <- min(Dist)                             # min distance
min.theta2 <- theta2[Dist==min.dist]                # theta angle with min distances

# compare the two approaches

# plotting the Variance for each direction/angle
plot(theta, Var, type = 'l', col='red',
     lwd = 2, lty=1, ylab = "Objective value")
abline(v=c(0, pi/2, pi, 3/2*pi, 2*pi), lty=2)       # fundamental angles
```

```
points(max.theta, rep(max.var, length(max.theta)),
       pch=4, col='blue', lwd=3, cex=2)

# plotting the avg squared distances for each direction/angle
points(theta2, Dist, type = 'l',
       col='cyan', lwd = 2, lty=1)
abline(v=c(0, pi/2, pi, 3/2*pi, 2*pi), lty=2)      # fundamental angles
points(min.theta2, rep(min.dist, length(min.theta2)),
       pch=4, col='blue', lwd=3, cex=2)

legend(4,3,
       c("Variance","Avg. sq. distances"),
       col=c("red","cyan"),
       lty=c(1,1),
       bg="white")
```



Global minima/maxima do differ, but are the arguments (theta values) minimizing/maximizing these functions the same?

```
max.theta
```

```
## [1] 1.256637 4.398230
```

```
min.theta2
```

```
## [1] 1.256637 4.398230
```

The same result can be obtained through low-rank approximations.

The functions **princomp()** and **prcomp()** in the **stats** package use the spectral decomposition and the singular value decomposition (SVD), respectively. We will focus on the latter.

PCA on decathlon2 data

We are going to perform PCA to the first 10 columns of the **decathlon2** data set to analyze athletes performance.

```
# select only the first 10 columns
decathlon2<- decathlon2[, 1:10]
head(decathlon2)
```

```
##           X100m Long.jump Shot.put High.jump X400m X110m.hurdle Discus
## SEBRLE    11.04      7.58    14.83      2.07 49.81        14.69  43.75
## CLAY      10.76      7.40    14.26      1.86 49.37        14.05  50.72
## BERNARD   11.02      7.23    14.25      1.92 48.93        14.99  40.87
## YURKOV    11.34      7.09    15.19      2.10 50.42        15.31  46.26
## ZSIVOCZKY 11.13      7.30    13.48      2.01 48.62        14.17  45.67
## McMULLEN  10.83      7.31    13.76      2.13 49.91        14.38  44.41
##           Pole.vault Javeline X1500m
## SEBRLE          5.02    63.19  291.7
## CLAY            4.92    60.15  301.5
## BERNARD         5.32    62.77  280.1
## YURKOV          4.72    63.44  276.4
## ZSIVOCZKY       4.42    55.37  268.0
## McMULLEN        4.42    56.37  285.1
```

```
dim(decathlon2)
```

```
## [1] 27 10
```

```
summary(decathlon2)
```

```
##      X100m         Long.jump        Shot.put       High.jump
##  Min.   :10.44   Min.   :6.800   Min.   :12.68   Min.   :1.860
##  1st Qu.:10.84   1st Qu.:7.210   1st Qu.:14.17   1st Qu.:1.930
##  Median :10.97   Median :7.310   Median :14.57   Median :1.980
##  Mean   :10.99   Mean   :7.365   Mean   :14.54   Mean   :1.998
##  3rd Qu.:11.13   3rd Qu.:7.545   3rd Qu.:15.01   3rd Qu.:2.080
##  Max.   :11.64   Max.   :7.960   Max.   :16.36   Max.   :2.150
##      X400m         X110m.hurdle       Discus        Pole.vault
##  Min.   :46.81   Min.   :13.97   Min.   :37.92   Min.   :4.400
##  1st Qu.:48.70   1st Qu.:14.15   1st Qu.:42.27   1st Qu.:4.660
##  Median :49.20   Median :14.34   Median :44.72   Median :4.900
```

```
##  Mean   :49.31   Mean   :14.50   Mean   :44.85   Mean   :4.836
##  3rd Qu.:49.86   3rd Qu.:14.87   3rd Qu.:46.93   3rd Qu.:5.000
##  Max.   :51.16   Max.   :15.67   Max.   :51.65   Max.   :5.400
##     Javeline         X1500m
##  Min.   :50.31   Min.   :262.1
##  1st Qu.:55.32   1st Qu.:271.6
##  Median :57.19   Median :278.1
##  Mean   :58.32   Mean   :278.5
##  3rd Qu.:62.05   3rd Qu.:283.6
##  Max.   :70.52   Max.   :301.5
```

To perform PCA we use the function **prcomp**.

```
help(prcomp)
```

Among other things, the function takes as input:

- **data:** a data frame
- **scale:** a logical value (TRUE/FALSE) indicating whether the variables should be scaled to have unit variance before the analysis takes place

Note that by default variables are centered to have zero mean.

Therefore, to perform PCA after scaling the data, we run:

```
res <- prcomp(decathlon2, scale = TRUE)
str(res)
```

```
## List of 5
##  $ sdev    : num [1:10] 1.936 1.321 1.232 1.016 0.786 ...
##  $ rotation: num [1:10, 1:10] -0.423 0.392 0.369 0.314 -0.332 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:10] "X100m" "Long.jump" "Shot.put" "High.jump" ...
##   .. ..$ : chr [1:10] "PC1" "PC2" "PC3" "PC4" ...
##  $ center  : Named num [1:10] 10.99 7.36 14.54 2 49.31 ...
##   ..- attr(*, "names")= chr [1:10] "X100m" "Long.jump" "Shot.put" "High.jump" ...
##  $ scale   : Named num [1:10] 0.2817 0.2944 0.8364 0.0956 0.9773 ...
##   ..- attr(*, "names")= chr [1:10] "X100m" "Long.jump" "Shot.put" "High.jump" ...
##  $ x       : num [1:27, 1:10] 0.273 0.888 -1.347 -0.911 -0.102 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : chr [1:27] "SEBRLE" "CLAY" "BERNARD" "YURKOV" ...
##   .. ..$ : chr [1:10] "PC1" "PC2" "PC3" "PC4" ...
##  - attr(*, "class")= chr "prcomp"
```

The result is a list containing 5 elements:

- **sdev:** the standard deviations of the principal components (square root of the eigenvalues)
- **rotation:** the matrix of variable loadings (eigenvectors)
- **center:** the centering used if **scale=TRUE**
- **scale:** the scaling used if **scale=TRUE**
- **x:** the scores, i.e. the rotated data

Selecting the number of components

The selection of a "suitable" number of components is non-trivial since in unsupervised learning tasks no ground-truth/labels/response variable is available (which may be used to assess and validate our results otherwise, e.g. through cross-validation – more on this later).

However, some "rules of thumb" have been developed. For instance, one may consider the **percentage of variance explained** (**PVE**), or the **cumulative PVE**.

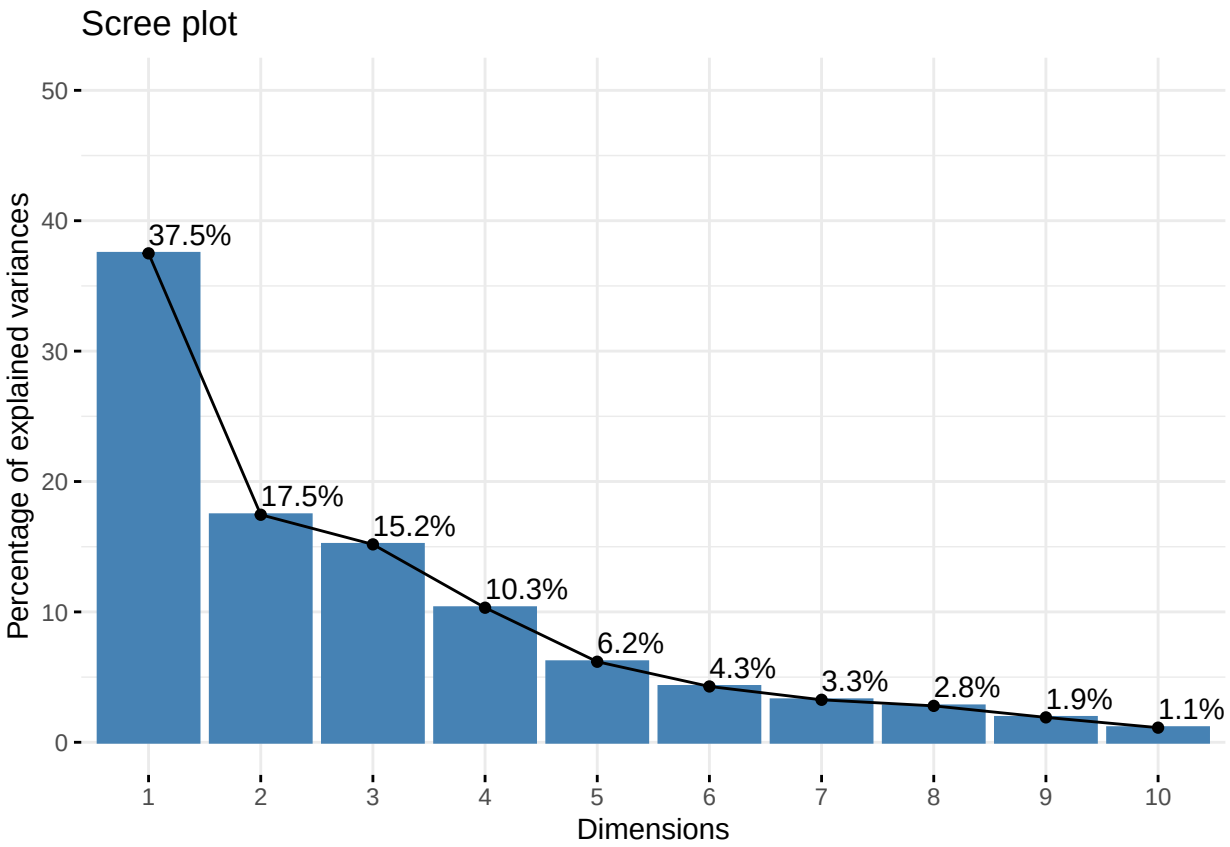
Here we use the `get_eig` function of the **factoextra** package.

```
help(get_eig)
# eigenvalue, PVE and cumulative PVE for each PC
get_eig(res)
```

```
##        eigenvalue variance.percent cumulative.variance.percent
## Dim.1   3.7499727        37.499727                     37.49973
## Dim.2   1.7451681        17.451681                     54.95141
## Dim.3   1.5178280        15.178280                     70.12969
## Dim.4   1.0322001        10.322001                     80.45169
## Dim.5   0.6178387         6.178387                     86.63008
## Dim.6   0.4282908         4.282908                     90.91298
## Dim.7   0.3259103         3.259103                     94.17209
## Dim.8   0.2793827         2.793827                     96.96591
## Dim.9   0.1911128         1.911128                     98.87704
## Dim.10  0.1122959         1.122959                    100.00000
```

Based on this information we can create a **scree plot**, and try to find an *elbow* (i.e. an inflection point) therein.

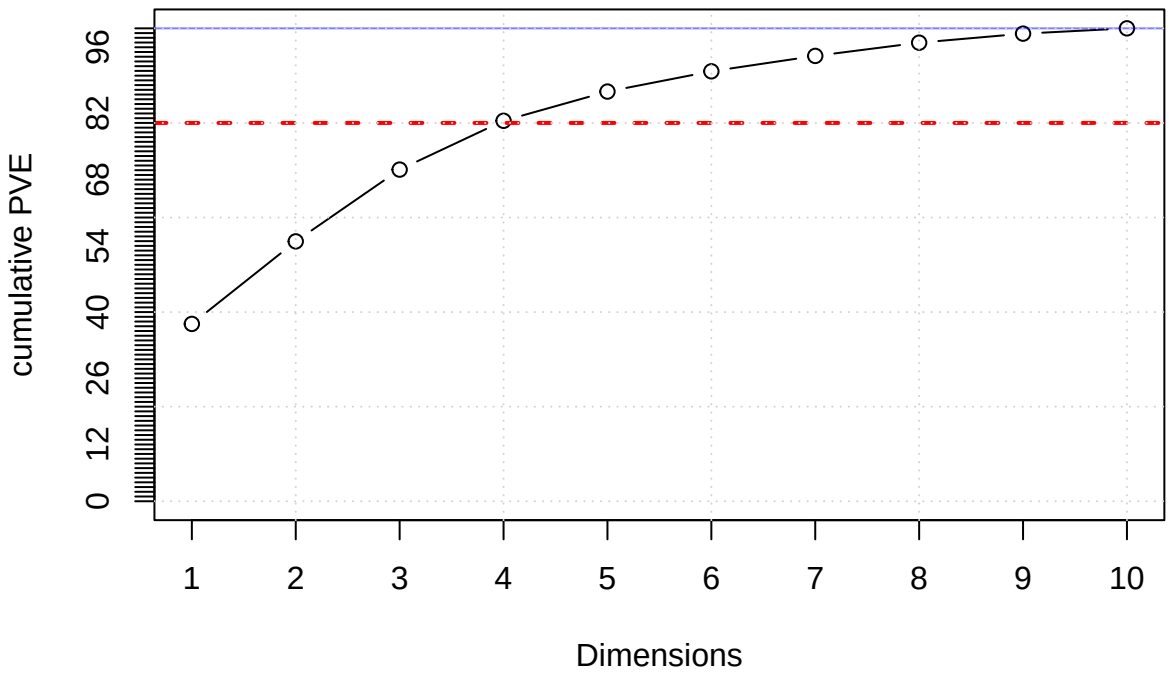
```
fviz_eig(res, addlabels = TRUE, ylim = c(0, 50))
```



Sometimes, it is not easy to identify an elbow, or this might be associated to a particularly low cumulative PVE.

Thus, it is common to fix an **acceptance threshold** a priori (e.g. 80%), and then look at the **cumulative PVE**.

```
plot(get_eig(res)$cumulative.variance.percent,
     type='b', axes=F, xlab='Dimensions', ylab='cumulative PVE', ylim=c(0,100))
abline(h=100, col=alpha('blue',0.5))
abline(h=80, lty=2, col='red', lwd=2) # thresholding
box()
axis(2, at=0:100,labels=0:100)
axis(1,at=1:ncol(decathlon2),labels=1:ncol(decathlon2))
grid()
```



Loadings interpretation

Let's focus on the loadings, i.e., the eigenvectors representing the directions of the PCs.

```
loadings <- res$rotation
loadings
```

```
##                     PC1        PC2          PC3         PC4        PC5
## X100m        -0.42290657  0.2594748 -0.081870461  0.09974877 -0.2796419
## Long.jump     0.39189495 -0.2887806  0.005082180 -0.18250903  0.3355025
## Shot.put      0.36926619  0.2135552 -0.384621732  0.03553644 -0.3544877
## High.jump     0.31422571  0.4627797 -0.003738604  0.07012348  0.3824125
## X400m        -0.33248297  0.1123521 -0.418635317  0.26554389  0.2534755
## X110m.hurdle -0.36995919  0.2252392 -0.338027983 -0.15726889  0.2048540
## Discus        0.37020078  0.1547241 -0.219417086  0.39137188 -0.4319091
## Pole.vault   -0.11433982 -0.5583051 -0.327177839 -0.24759476 -0.3340758
## Javeline      0.18341259  0.0745854 -0.564474643 -0.47792535  0.1697426
## X1500m        0.03599937 -0.4300522 -0.286328973  0.64220377  0.3227349
##                     PC6         PC7         PC8         PC9        PC10
## X100m         0.16023494 -0.03227949  0.35266427 -0.71190625  0.03272397
## Long.jump     0.07384658  0.24902853  0.72986071 -0.12801382  0.02395904
## Shot.put      0.32207320  0.23059438 -0.01767069  0.07184807 -0.61708920
## High.jump     0.52738027  0.03992994 -0.25003572 -0.14583529  0.41523052
## X400m        -0.23884715  0.69014364 -0.01543618  0.13706918  0.12016951
```

```
## X110m.hurdle  0.26249611 -0.42797378  0.36415520  0.49550598 -0.03514180
## Discus       -0.28217086 -0.18416631  0.26865454  0.18621144  0.48037792
## Pole.vault    0.43606610  0.12654370 -0.16086549  0.02983660  0.40290423
## Javeline     -0.42368592 -0.23324548 -0.19922452 -0.33300936  0.02100398
## X1500m        0.10850981 -0.34406521 -0.09752169 -0.19899138 -0.18954698
```

We can plot the first two PCs (PC1 and PC2) in the **graph of variables** (also called correlation circle).

Here the importance of the original features is represented by the **color code**:

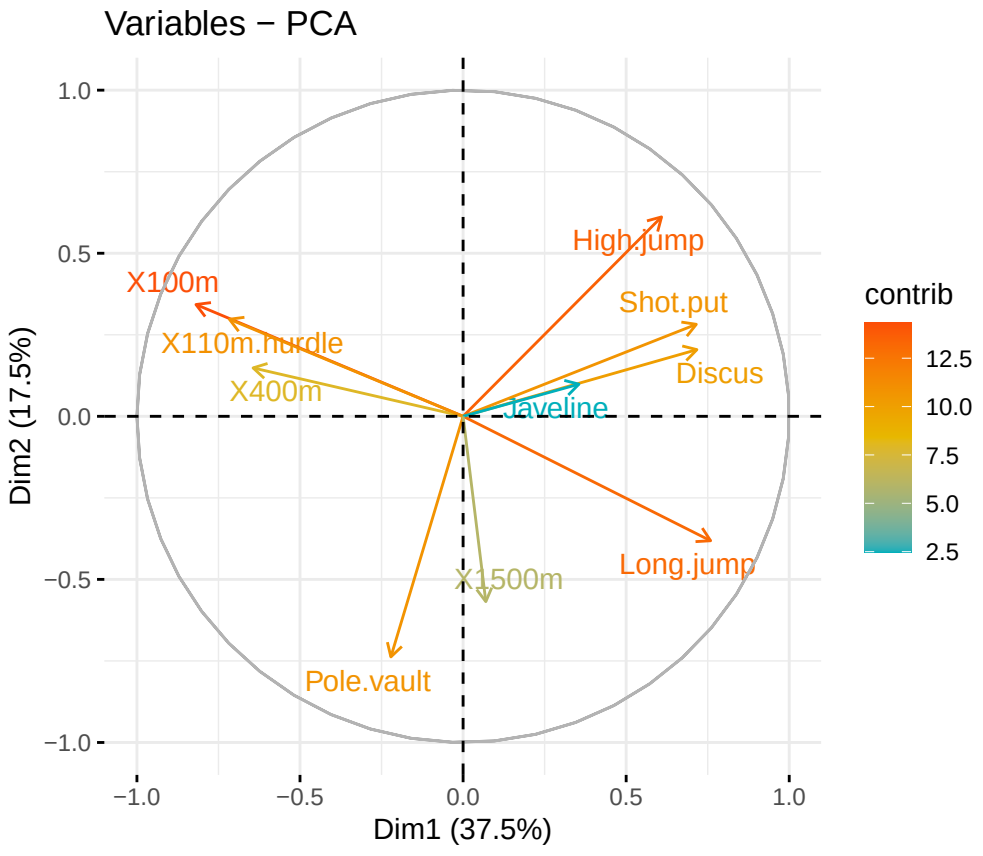
- red: *high*
- blue: *medium*
- white: *low*

and by the lenght of the vector (its *closeness* to the circle).

In these plots, positively correlated variables have same direction (they are close to each other, i.e. they have a small angle in between). Negatively correlated variables have opposite directions. Uncorrelated variables are orthogonal.

Remark: a lower value for "X100m", "X400m", "X110m hurdle" means a better performance. So it makes sense that they are negatively correlated with "long jump".

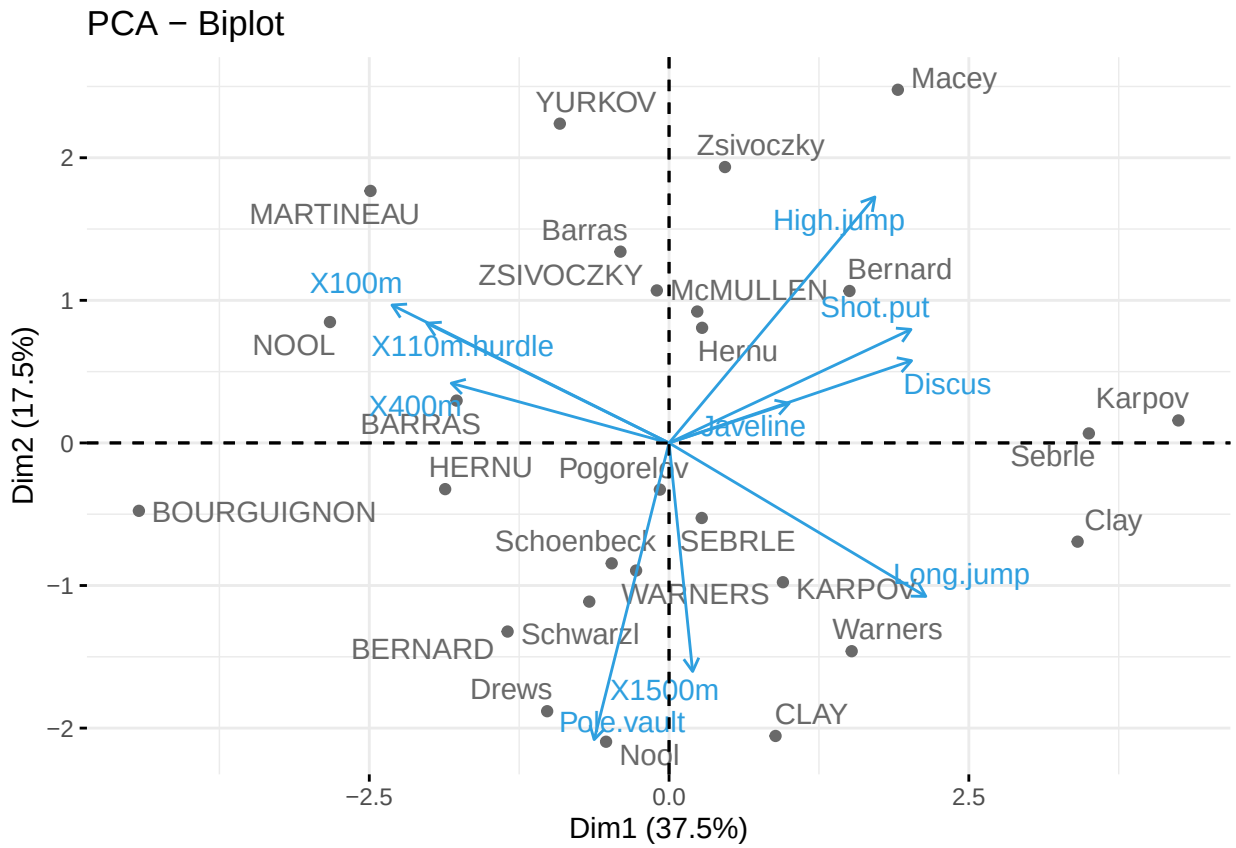
```
fviz_pca_var(res,
             col.var = "contrib", # Color by contributions to the PC
             gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"),
             repel = TRUE )     # Avoid text overlapping
```



We also see that “X100m”, “Long jump” and “High jump” and “Pole vault” contribute the most to PC1 and PC2.

If we want to show also the individuals we can run the **biplot of individuals and variables**. Here similar athletes are grouped together.

```
fviz_pca_biplot(res, repel = TRUE,
                col.var = "#2E9FDF", # Variables color
                col.ind = "#696969"  # Individuals color
)
```

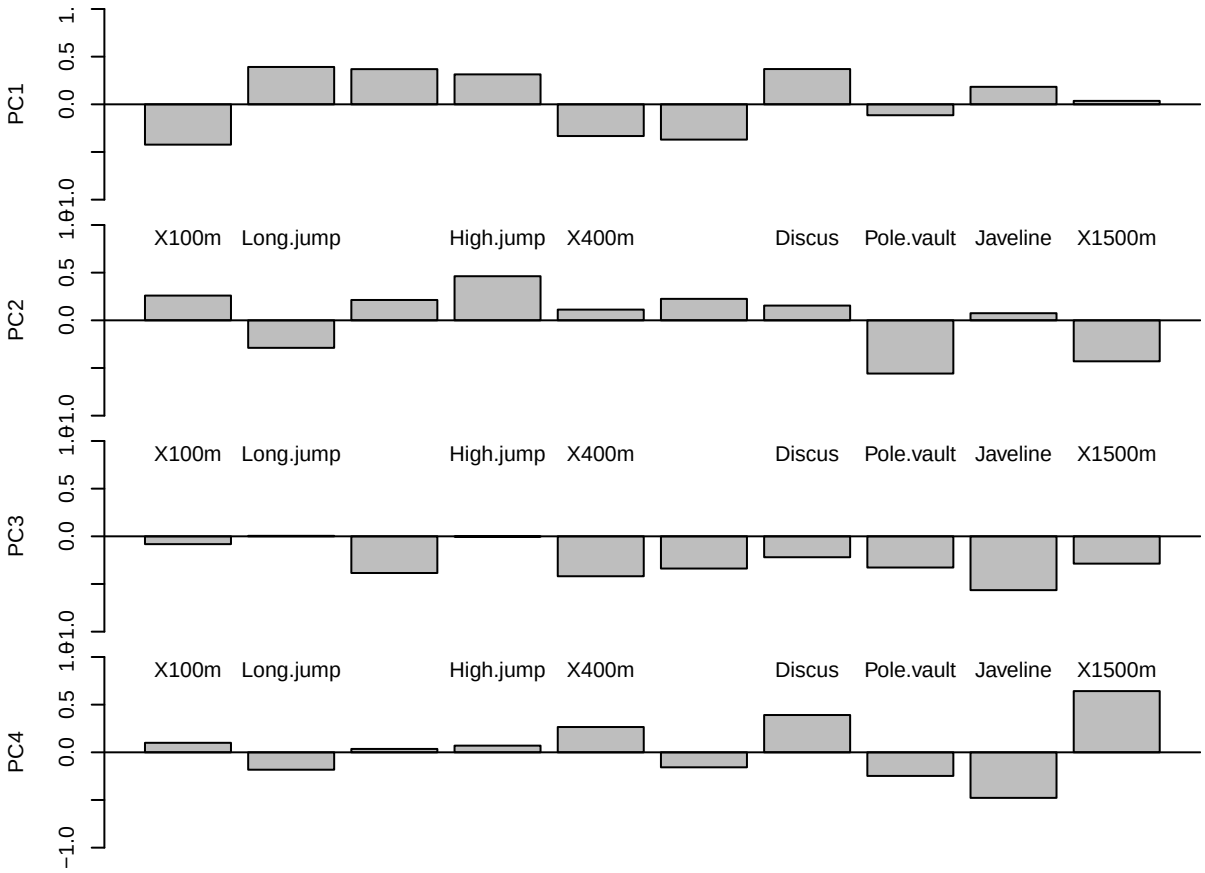


Along PC1 we tend to find athletes with higher/lower overall performance, which are mainly related to agility. On PC2 we can distinguish stronger athletes from others.

Remark: observations are represented by their projections, but variables are represented by their correlations.

Information regarding all the PCs (e.g., the first 4) can be obtained in the following way:

```
plot.new()
par(mar = c(1,4,0,2), mfrow = c(4,1))
for(i in 1:4)
{
  barplot(loadings[,i], ylim = c(-1, 1),
          ylab=paste0("PC", i))           # this is not the axis label, but the plot title
  abline(h=0)
}
```



The "importance" of each variable across different PCs is also contained in the field **cos2** of the function `get_pca_var`:

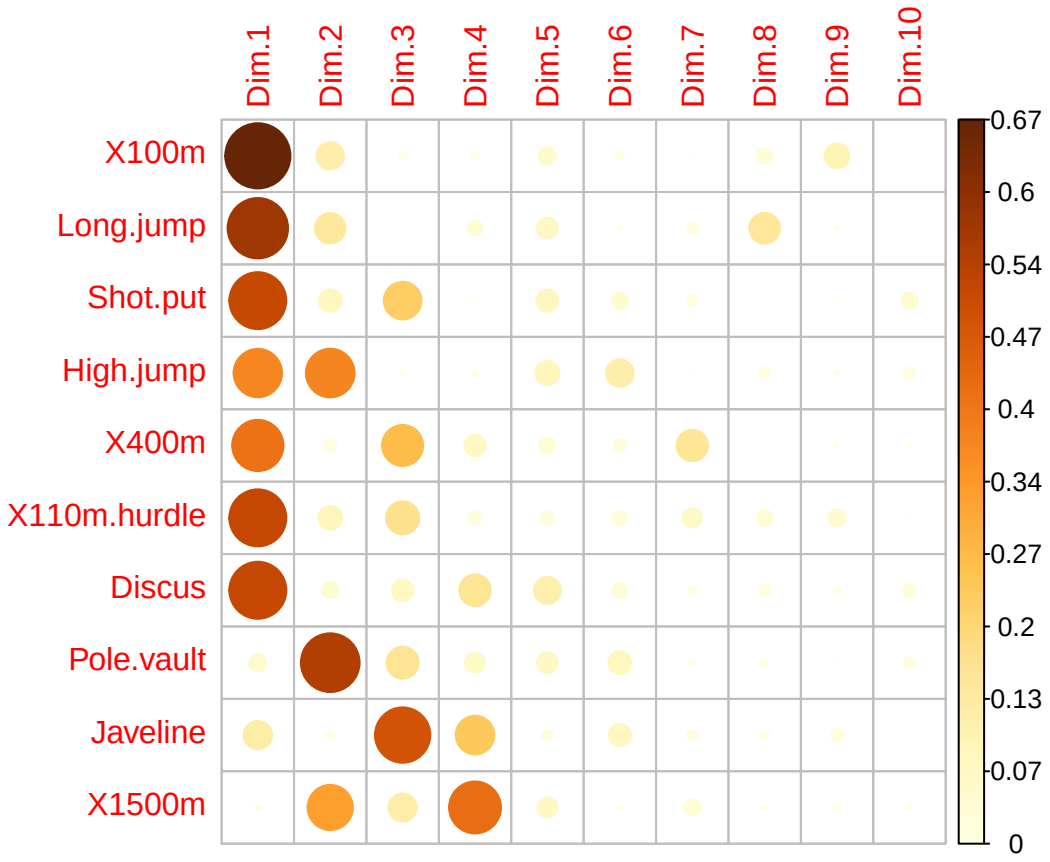
- A high cos2 indicates a good representation of the variable on the principal component. In this case the variable is positioned close to the circumference of the correlation circle.
- A low cos2 indicates that the variable is not perfectly represented by the PCs. In this case the variable is close to the center of the circle.

For a given variable, the sum of the cos2 on all the principal components is equal to one.

```
help(get_pca_var)
varpca <- get_pca_var(res)
varpca
```

```
## Principal Component Analysis Results for variables
##  ===================================================
##   Name       Description
## 1 "$coord"   "Coordinates for the variables"
## 2 "$cor"     "Correlations between variables and dimensions"
## 3 "$cos2"    "Cos2 for the variables"
## 4 "$contrib" "contributions of the variables"
```

```
corrplot(varpca$cos2, is.corr=FALSE)
```



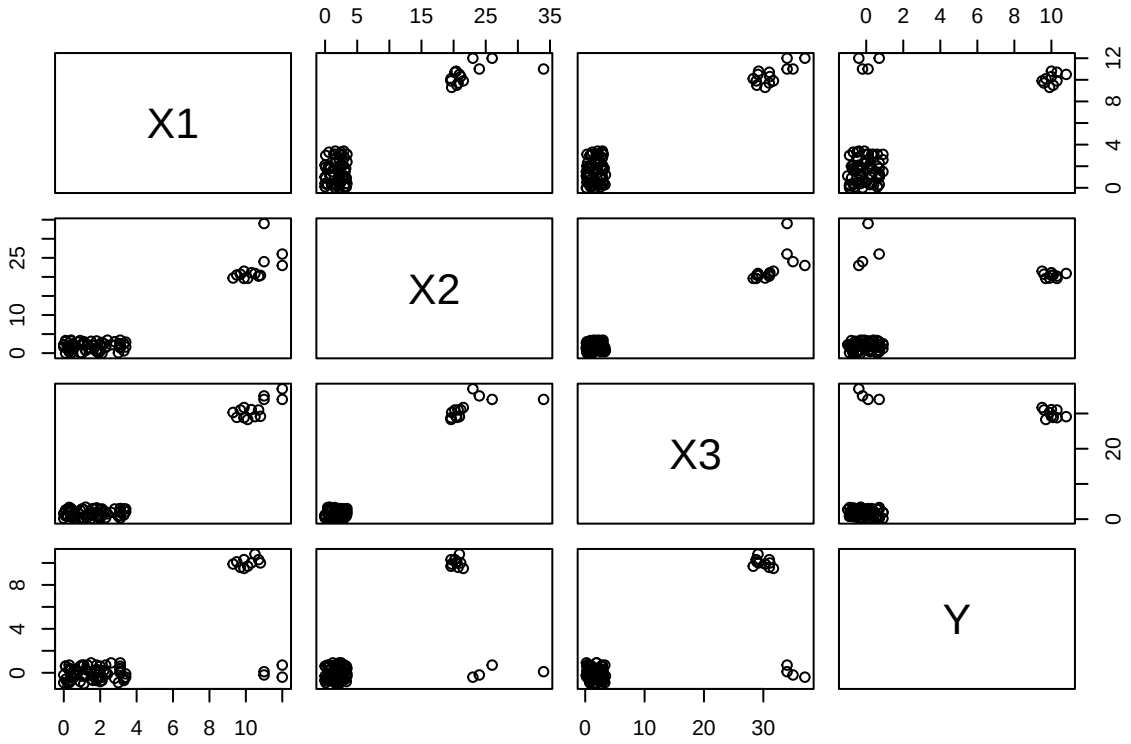
Robust PCA

Unlike classical PCA, there exist robust versions of PCA (e.g., ROBPCA) that are resistant to outliers in the data.

Here robust loadings are computed in a robust manner (e.g., through projection-pursuit techniques and the minimum covariance determinant method) to remove the effects of outliers.

We will focus on the **PcaHubert** function within the **rrcov** package, and itdepends on a trimming proportion $\alpha = h/n$ denoting the fraction of points (with larger distances) that do not contribute to the objective function (in this case h out of n).

```
##

set.seed(123)
library(rrcov)
help(PcaHubert)

# PCA of the Hawkins Bradu Kass's Artificial Data
# The first 14 observations are outliers, created in two groups:
# 1--10 (more difficult to detect) and 11--14.
data(hbk)
pairs(hbk)
```




```
pcaRob <- PcaHubert(hbk, k=4, alpha=0.75)
print("Robust output")
```

```
## [1] "Robust output"
```

```
pcaRob
```

```
## Call:
## PcaHubert(x = hbk, k = 4, alpha = 0.75)
##
## Standard deviations:
## [1] 1.3660302 1.2401095 1.1601012 0.6450163
```

```
pcaRob$loadings
```

```
##            PC1         PC2        PC3         PC4
## X1 -0.51016971  0.78654102 -0.3364001 -0.08896652
## X2 -0.62522656 -0.61389365 -0.4814074 -0.02175382
## X3 -0.58980262 -0.02635976  0.7927732  0.15148832
## Y   0.03084628  0.06158714 -0.1630721  0.98420680
```

```
## Compare with the classical PCA
pca <- prcomp(hbk)
cat("\n\n")               # line break
```

```
print("Classical output")
```

```
## [1] "Classical output"
```

```
pca
```

```
## Standard deviations (1, .., p=4):
## [1] 14.9371883  2.3532250  1.2992998  0.9559814
##
## Rotation (n x k) = (4 x 4):
##           PC1          PC2        PC3         PC4
## X1 -0.2362481  0.030060216  0.1671421  0.95673754
## X2 -0.5450798 -0.324339001 -0.7730323  0.01064233
## X3 -0.7843094  0.000936946  0.5486521 -0.28954926
## Y  -0.1786999  0.945462700 -0.2710450 -0.02648092
```

```
## Compare with the classical PCA on clean data
pcaclean <- prcomp(hbk[15:nrow(hbk), ])
cat("\n\n")               # line break
```

```
print("Classical output on clean data")
```

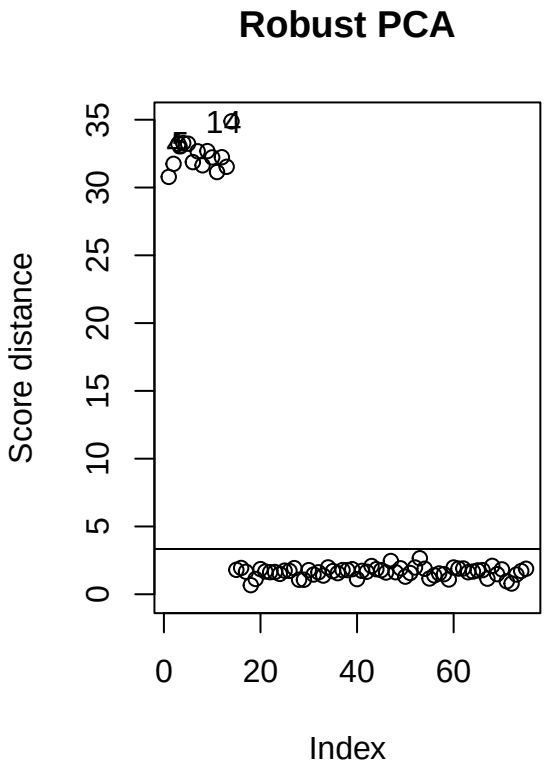
```
## [1] "Classical output on clean data"
```

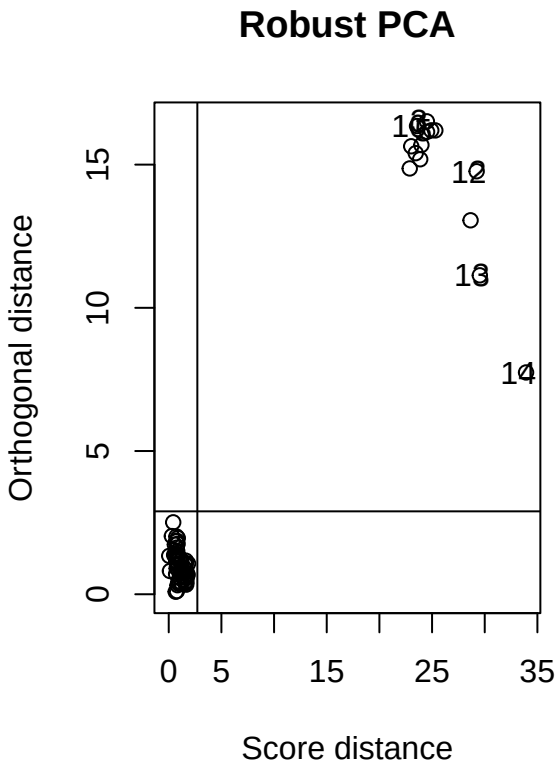
```
pcaclean
```

```
## Standard deviations (1, .., p=4):
## [1] 1.1516768 1.0455152 0.9780616 0.5438023
##
## Rotation (n x k) = (4 x 4):
##            PC1         PC2        PC3         PC4
## X1 -0.51016971 -0.78654102  0.3364001  0.08896652
## X2 -0.62522656  0.61389365  0.4814074  0.02175382
## X3 -0.58980262  0.02635976 -0.7927732 -0.15148832
## Y   0.03084628 -0.06158714  0.1630721 -0.98420680
```

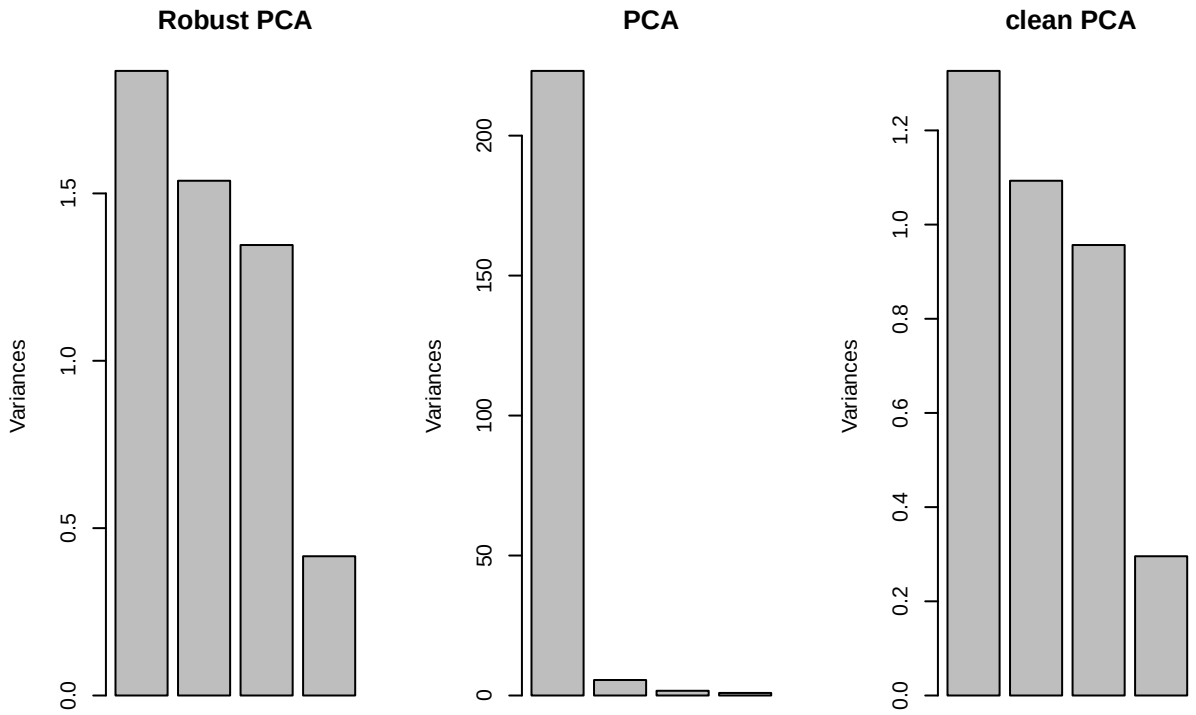
Let's plot their solutions:

```
par(mfrow=c(1,2))

plot(pcaRob)                    # distance plot

pca2 <- PcaHubert(hbk, k=2)
plot(pca2)                      # PCA diagnostic plot (or outlier map)
```

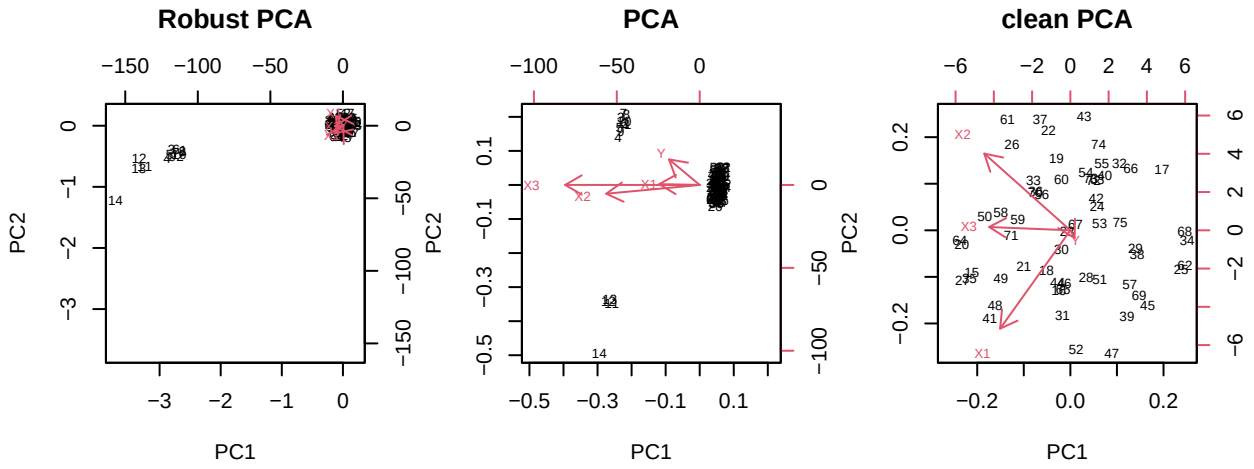





```
par(mfrow=c(1,3))

## Use the standard plots available for prcomp and princomp
screeplot(pcaRob, main="Robust PCA")
screeplot(pca ,main="PCA")
screeplot(pcaclean ,main=" clean PCA")
```




```
par(mfrow=c(1,3))

biplot(pcaRob)
title("Robust PCA", line = 3)

biplot(pca)
title("PCA", line = 3)

biplot(pcaclean)
title("clean PCA", line = 3)
```



Retrieve the robust covariance matrix:

```
print("Cov by Robust PCA (25% trimming)")
```

```
## [1] "Cov by Robust PCA (25% trimming)"
```

```
py <- PcaHubert(hbk, alpha=0.75)
cov.1 <- py@loadings %*% diag(py@eigenvalues) %*% t(py@loadings)
cov.1
```

```
##            X1         X2         X3           Y
## X1 1.58938055 0.07060011  0.1706859  0.11895931
## X2 0.07060011 1.62092195  0.1993732  0.01152154
## X3 0.17068586 0.19937318  1.4960449 -0.21043430
## Y  0.11895931 0.01152154 -0.2104343  0.04339776
```

```
cat("\n\n")
```

```
print("Cov by another robust method")
```

```
## [1] "Cov by another robust method"
```

```
CovRobust(hbk, CovControlSest(method="auto",))
```

```
##
## Call:
## CovSest(x = x, control = obj)
## -> Method:  S-estimates: DET-S
##
## Robust Estimate of Location:
## [1]   1.54001   1.83624   1.67311  -0.07768
##
## Robust Estimate of Covariance:
##     X1        X2        X3        Y
## X1   1.73686   0.05954   0.25298   0.13609
## X2   0.05954   1.74166   0.23390  -0.01709
## X3   0.25298   0.23390   1.66731  -0.19090
## Y    0.13609  -0.01709  -0.19090   0.48481
```

```
cat("\n\n")
```

```
print("Cov on clean data")
```

```
## [1] "Cov on clean data"
```

```
cov(hbk[15:nrow(hbk), ])
```

```
##           X1          X2         X3            Y
## X1 1.1320546 0.050754098  0.1173361  0.058661202
## X2 0.0507541 1.152273224  0.1407377  0.001857923
## X3 0.1173361 0.140737705  1.0701585 -0.105483607
## Y  0.0586612 0.001857923 -0.1054836  0.317300546
```

```
cat("\n\n")
```

```
print("Empirical output (i.e. 0% trimming)")
```

```
## [1] "Empirical output (i.e. 0% trimming)"
```

```
cov(hbk)
```

```
##           X1       X2        X3         Y
## X1 13.341712 28.46921  41.24398  9.477306
## X2 28.469207 67.88297  94.66562 20.388456
## X3 41.243982 94.66562 137.83486 31.032420
## Y   9.477306 20.38846  31.03242 12.199809
```

Multidimensional Scaling

Multidimensional scaling (MDS) is useful to visualize the level of similarity of individual cases in a dataset. MDS is used to translate "information about the pairwise 'distances' among a set of"n" objects or individuals" into a configuration of "n" points mapped into an abstract Cartesian space.

See for instance:

- cmdscale() [stats package]: Classical (metric) multidimensional scaling.
- isoMDS() [MASS package]: Kruskal's non-metric multidimensional scaling (one form of non-metric MDS).
- sammon() [MASS package]: Sammon's non-linear mapping (one form of non-metric MDS).

These functions require a distance object as input and the number of dimensions in the reduced space (by default equal to 2).