Machine Learning Meta

Alfonso R. Reyes

2019-09-18

Contents

Prerequisites

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

```
install.packages("bookdown")
# or the development version
# devtools::install_github("rstudio/bookdown")
```

Remember each Rmd file contains one and only one chapter, and a chapter is defined by the first-level heading `#`.

To compile this example to PDF, you need XeLaTeX. You are recommended to install TinyTeX (which includes XeLaTeX): https://yihui.name/tinytex/.

Chapter 1

PCA: prcomp vs princomp

http://www.sthda.com/english/articles/31-principal-component-methods-in-r-practical-guide/118-principal-component-ana

1.1 General methods for principal component analysis

There are two general methods to perform PCA in R :

- Spectral decomposition which examines the covariances / correlations between variables
- Singular value decomposition which examines the covariances / correlations between individuals

The function princomp() uses the spectral decomposition approach. The functions prcomp() and PCA()[FactoMineR] use the singular value decomposition (SVD).

1.2 prcomp() and princomp() functions

The simplified format of these 2 functions are :

```
prcomp(x, scale = FALSE)
princomp(x, cor = FALSE, scores = TRUE)
```

1. Arguments for prcomp():
   `x`: a numeric matrix or data frame
   `scale`: a logical value indicating whether the variables should be scaled to have unit variance before the analysis takes place
2. Arguments for princomp():
   `x`: a numeric matrix or data frame `cor`: a logical value. If TRUE, the data will be centered and scaled before the analysis `scores`: a logical value. If TRUE, the coordinates on each principal component are calculated

1.3 factoextra

```
# install.packages("factoextra")
```

```
library(factoextra)
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
#> Welcome! Related Books: `Practical Guide To Cluster Analysis in R` at https://goo.gl/13EFCZ
```

1.4 demo dataset

We'll use the data sets `decathlon2` [in factoextra], which has been already described at: PCA - Data format.

Briefly, it contains:

- Active individuals (rows 1 to 23) and active variables (columns 1 to 10), which are used to perform the principal component analysis
- Supplementary individuals (rows 24 to 27) and supplementary variables (columns 11 to 13), which coordinates will be predicted using the PCA information and parameters obtained with active individuals/variables.

```
library("factoextra")
data(decathlon2)
decathlon2.active <- decathlon2[1:23, 1:10]
head(decathlon2.active[, 1:6])
#>           X100m Long.jump Shot.put High.jump X400m X110m.hurdle
#> SEBRLE     11.0      7.58     14.8      2.07  49.8         14.7
#> CLAY       10.8      7.40     14.3      1.86  49.4         14.1
#> BERNARD    11.0      7.23     14.2      1.92  48.9         15.0
#> YURKOV     11.3      7.09     15.2      2.10  50.4         15.3
#> ZSIVOCZKY  11.1      7.30     13.5      2.01  48.6         14.2
#> McMULLEN   10.8      7.31     13.8      2.13  49.9         14.4
```

```
decathlon2.supplementary <- decathlon2[24:27, 1:10]
head(decathlon2.supplementary[, 1:6])
#>         X100m Long.jump Shot.put High.jump X400m X110m.hurdle
#> KARPOV   11.0      7.30     14.8      2.04  48.4         14.1
#> WARNERS  11.1      7.60     14.3      1.98  48.7         14.2
#> Nool     10.8      7.53     14.3      1.88  48.8         14.8
#> Drews    10.9      7.38     13.1      1.88  48.5         14.0
```

1.5 Compute PCA in R using prcomp()

In this section we'll provide an easy-to-use R code to compute and visualize PCA in R using the prcomp() function and the factoextra package.

‘. Load factoextra for visualization

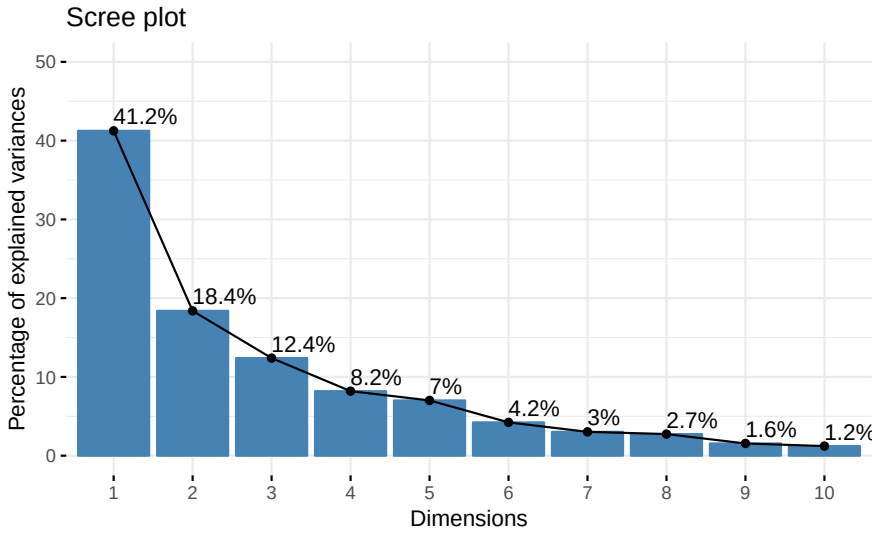
```
library(factoextra)
```

2. compute PCA

```
# compute PCA
res.pca <- prcomp(decathlon2.active, scale = TRUE)
```

3. Visualize eigenvalues (scree plot). Show the percentage of variances explained by each principal component.

```
# Visualize eigenvalues (scree plot).
fviz_eig(res.pca, addlabels = TRUE, ylim = c(0, 50))
```

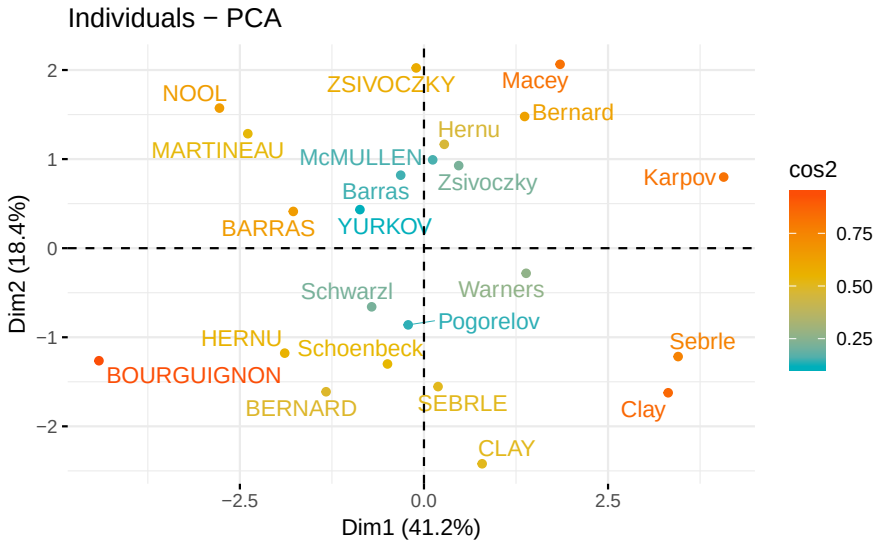


From the plot above, we might want to stop at the fifth principal component. 87% of the information (variances) contained in the data are retained by the first five principal components.

1.6 Plots: quality and contribution

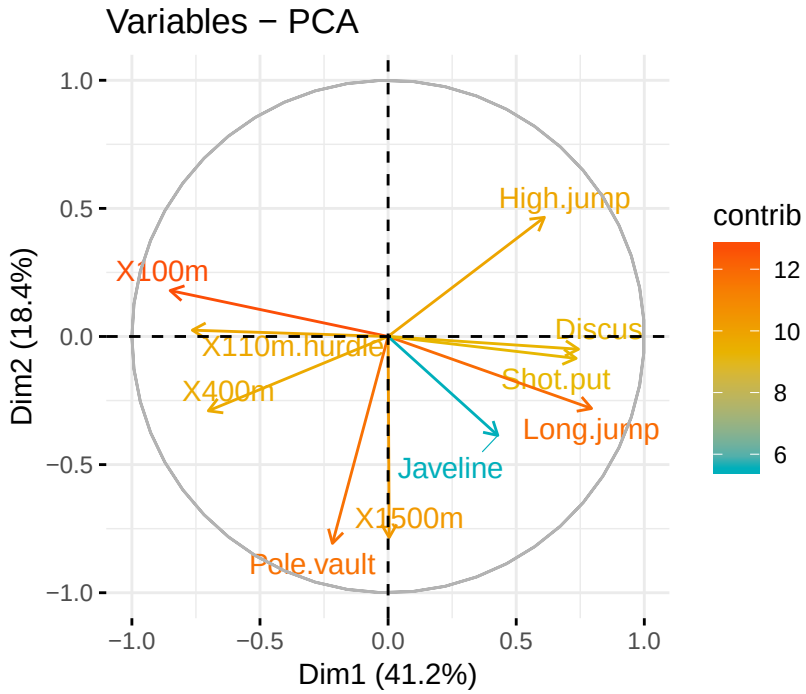
4. Graph of individuals. Individuals with a similar profile are grouped together.

```
# Graph of individuals.
fviz_pca_ind(res.pca,
             col.ind = "cos2", # Color by the quality of representation
             gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"),
             repel = TRUE     # Avoid text overlapping
             )
```



5. Graph of variables. Positive correlated variables point to the same side of the plot. Negative correlated variables point to opposite sides of the graph.

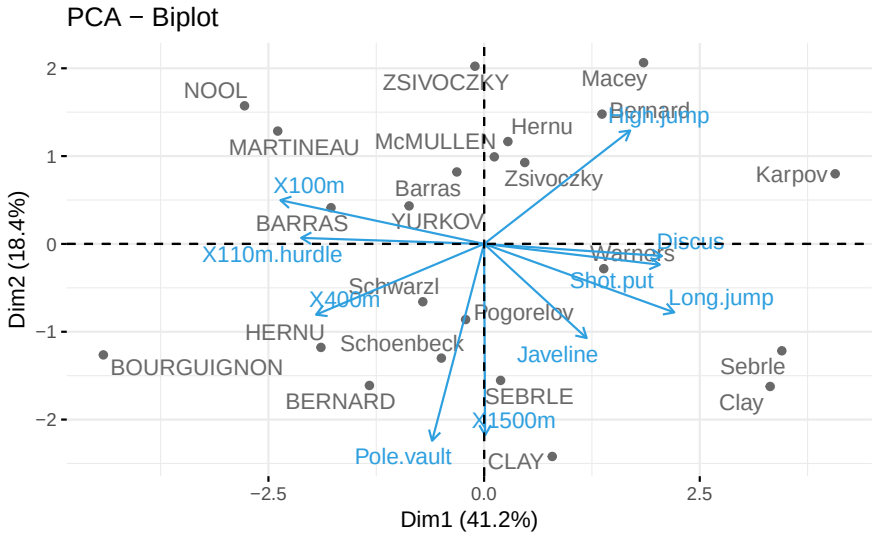
```
# Graph of variables.
fviz_pca_var(res.pca,
             col.var = "contrib", # Color by contributions to the PC
             gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"),
             repel = TRUE     # Avoid text overlapping
             )
```



6. Biplot of individuals and variables

```
# Biplot of individuals and variables
fviz_pca_biplot(res.pca, repel = TRUE,
                col.var = "#2E9FDF", # Variables color
                col.ind = "#696969"  # Individuals color
```

```
)
```



1.7 Access to the PCA results

```
library(factoextra)
# Eigenvalues
eig.val <- get_eigenvalue(res.pca)
eig.val
#>        eigenvalue variance.percent cumulative.variance.percent
#> Dim.1       4.124            41.24                        41.2
#> Dim.2       1.839            18.39                        59.6
#> Dim.3       1.239            12.39                        72.0
#> Dim.4       0.819             8.19                        80.2
#> Dim.5       0.702             7.02                        87.2
#> Dim.6       0.423             4.23                        91.5
#> Dim.7       0.303             3.03                        94.5
#> Dim.8       0.274             2.74                        97.2
#> Dim.9       0.155             1.55                        98.8
#> Dim.10      0.122             1.22                       100.0

# Results for Variables
res.var <- get_pca_var(res.pca)
res.var$coord          # Coordinates
#>                 Dim.1   Dim.2   Dim.3   Dim.4  Dim.5    Dim.6    Dim.7
#> X100m        -0.85063  0.1794 -0.3016  0.0336 -0.194  0.03537 -0.09134
#> Long.jump     0.79418 -0.2809  0.1905 -0.1154  0.233 -0.03373 -0.15433
#> Shot.put      0.73391 -0.0854 -0.5176  0.1285 -0.249 -0.23979 -0.00989
#> High.jump     0.61008  0.4652 -0.3301  0.1446  0.403 -0.28464  0.02816
#> X400m        -0.70160 -0.2902 -0.2835  0.4308  0.104 -0.04929  0.28611
#> X110m.hurdle -0.76413  0.0247 -0.4489 -0.0169  0.224  0.00263 -0.37007
#> Discus        0.74321 -0.0497 -0.1765  0.3950 -0.408  0.19854 -0.14273
#> Pole.vault   -0.21727 -0.8075 -0.0941 -0.3390 -0.222 -0.32746 -0.01039
#> Javeline      0.42823 -0.3861 -0.6041 -0.3317  0.198  0.36210  0.13356
#> X1500m        0.00428 -0.7845  0.2195  0.4480  0.263  0.04205 -0.11137
```

```
#>                 Dim.8   Dim.9   Dim.10
#> X100m        -0.10472 -0.3031  0.04442
#> Long.jump    -0.39738 -0.0516  0.02972
#> Shot.put      0.02436  0.0478  0.21745
#> High.jump     0.08441 -0.1121 -0.13357
#> X400m        -0.23355  0.0822 -0.03417
#> X110m.hurdle -0.00834  0.1618 -0.01563
#> Discus       -0.03956  0.0134 -0.17259
#> Pole.vault    0.03291 -0.0258 -0.13721
#> Javeline      0.05284 -0.0405 -0.00385
#> X1500m        0.19447 -0.1022  0.06283
res.var$contrib        # Contributions to the PCs
#>                 Dim.1   Dim.2  Dim.3   Dim.4 Dim.5    Dim.6   Dim.7
#> X100m        1.75e+01  1.7505  7.339  0.1376  5.39  0.29592  2.7571
#> Long.jump    1.53e+01  4.2904  2.930  1.6249  7.75  0.26900  7.8716
#> Shot.put     1.31e+01  0.3967 21.620  2.0141  8.82 13.59686  0.0323
#> High.jump    9.02e+00 11.7716  8.793  2.5499 23.12 19.15961  0.2620
#> X400m        1.19e+01  4.5799  6.488 22.6509  1.54  0.57451 27.0527
#> X110m.hurdle 1.42e+01  0.0333 16.261  0.0348  7.17  0.00164 45.2616
#> Discus       1.34e+01  0.1341  2.515 19.0413 23.76  9.32175  6.7323
#> Pole.vault   1.14e+00 35.4619  0.714 14.0231  7.01 25.35762  0.0357
#> Javeline     4.45e+00  8.1087 29.453 13.4296  5.58 31.00496  5.8957
#> X1500m       4.44e-04 33.4729  3.887 24.4939  9.88  0.41813  4.0989
#>                Dim.8  Dim.9  Dim.10
#> X100m         3.9952 59.174  1.6176
#> Long.jump    57.5332  1.715  0.7241
#> Shot.put      0.2162  1.471 38.7677
#> High.jump     2.5957  8.102 14.6265
#> X400m        19.8734  4.349  0.9573
#> X110m.hurdle  0.0254 16.858  0.2003
#> Discus        0.5702  0.115 24.4217
#> Pole.vault    0.3947  0.428 15.4356
#> Javeline      1.0173  1.054  0.0122
#> X1500m       13.7787  6.734  3.2370
res.var$cos2           # Quality of representation
#>                 Dim.1    Dim.2   Dim.3    Dim.4  Dim.5    Dim.6    Dim.7
#> X100m        7.24e-01 0.032184 0.09094 0.001127 0.0378 1.25e-03 8.34e-03
#> Long.jump    6.31e-01 0.078881 0.03631 0.013315 0.0544 1.14e-03 2.38e-02
#> Shot.put     5.39e-01 0.007294 0.26791 0.016504 0.0619 5.75e-02 9.77e-05
#> High.jump    3.72e-01 0.216424 0.10896 0.020895 0.1622 8.10e-02 7.93e-04
#> X400m        4.92e-01 0.084203 0.08039 0.185611 0.0108 2.43e-03 8.19e-02
#> X110m.hurdle 5.84e-01 0.000612 0.20150 0.000285 0.0503 6.93e-06 1.37e-01
#> Discus       5.52e-01 0.002466 0.03116 0.156032 0.1667 3.94e-02 2.04e-02
#> Pole.vault   4.72e-02 0.651977 0.00885 0.114911 0.0491 1.07e-01 1.08e-04
#> Javeline     1.83e-01 0.149080 0.36497 0.110048 0.0391 1.31e-01 1.78e-02
#> X1500m       1.83e-05 0.615409 0.04817 0.200713 0.0693 1.77e-03 1.24e-02
#>                 Dim.8    Dim.9   Dim.10
#> X100m        1.10e-02 0.091848 1.97e-03
#> Long.jump    1.58e-01 0.002661 8.83e-04
#> Shot.put     5.93e-04 0.002284 4.73e-02
#> High.jump    7.12e-03 0.012575 1.78e-02
#> X400m        5.45e-02 0.006750 1.17e-03
#> X110m.hurdle 6.96e-05 0.026166 2.44e-04
```

```
#> Discus       1.56e-03 0.000179 2.98e-02
#> Pole.vault   1.08e-03 0.000664 1.88e-02
#> Javeline     2.79e-03 0.001637 1.49e-05
#> X1500m       3.78e-02 0.010453 3.95e-03
# Results for individuals
res.ind <- get_pca_ind(res.pca)
res.ind$coord          # Coordinates
#>              Dim.1  Dim.2  Dim.3   Dim.4     Dim.5   Dim.6   Dim.7
#> SEBRLE       0.191 -1.554 -0.628  0.0821  1.142614 -0.4639 -0.2080
#> CLAY         0.790 -2.420  1.357  1.2698 -0.806848  1.3042 -0.2129
#> BERNARD     -1.329 -1.612 -0.196 -1.9209  0.082343 -0.4006 -0.4064
#> YURKOV      -0.869  0.433 -2.474  0.6972  0.398858  0.1029 -0.3249
#> ZSIVOCZKY   -0.106  2.023  1.305 -0.0993 -0.197024  0.8955  0.0883
#> McMULLEN     0.119  0.992  0.844  1.3122  1.585871  0.1866  0.4783
#> MARTINEAU   -2.392  1.285 -0.898  0.3731 -2.243352 -0.4567 -0.2998
#> HERNU       -1.891 -1.178 -0.156  0.8913 -0.126741  0.4362 -0.5661
#> BARRAS      -1.774  0.413  0.658  0.2287 -0.233837  0.0903  0.2159
#> NOOL        -2.777  1.573  0.607 -1.5555  1.424184  0.4972 -0.5321
#> BOURGUIGNON -4.414 -1.264 -0.010  0.6668  0.419152 -0.0820 -0.5983
#> Sebrle       3.451 -1.217 -1.678 -0.8087 -0.025053 -0.0828  0.0102
#> Clay         3.316 -1.623 -0.618 -0.3168  0.569165  0.7772  0.2575
#> Karpov       4.070  0.798  1.015  0.3134 -0.797426 -0.3296 -1.3637
#> Macey        1.848  2.064 -0.979  0.5847 -0.000216 -0.1973 -0.2693
#> Warners      1.387 -0.282  2.000 -1.0196 -0.040540 -0.5567 -0.2674
#> Zsivoczky    0.472  0.927 -1.728 -0.1848  0.407303 -0.1138  0.0399
#> Hernu        0.276  1.166  0.171 -0.8487 -0.689480 -0.3317  0.4431
#> Bernard      1.367  1.478  0.831  0.7453  0.859802 -0.3281  0.3636
#> Schwarzl    -0.710 -0.658  1.041 -0.9272 -0.288757 -0.6889  0.5657
#> Pogorelov   -0.214 -0.861  0.298  1.3556 -0.015053 -1.5938  0.7837
#> Schoenbeck  -0.495 -1.300  0.103 -0.2493 -0.645226  0.1617  0.8575
#> Barras      -0.316  0.819 -0.862 -0.5894 -0.779739  1.1742  0.9451
#>                Dim.8    Dim.9  Dim.10
#> SEBRLE       0.04346 -0.65934  0.0327
#> CLAY         0.61724 -0.06013 -0.3172
#> BERNARD      0.70386  0.17008 -0.0991
#> YURKOV       0.11500 -0.10952 -0.1197
#> ZSIVOCZKY   -0.20234 -0.52310 -0.3484
#> McMULLEN     0.29309 -0.10562 -0.3932
#> MARTINEAU   -0.29163 -0.22342 -0.6164
#> HERNU       -1.52940  0.00618  0.5537
#> BARRAS       0.68258 -0.66928  0.5309
#> NOOL        -0.43339 -0.11578 -0.0962
#> BOURGUIGNON  0.56362  0.52581  0.0586
#> Sebrle      -0.03059 -0.84721  0.2197
#> Clay        -0.58064  0.40978 -0.6160
#> Karpov       0.34531  0.19306  0.2172
#> Macey       -0.36322  0.36826  0.2125
#> Warners     -0.10947  0.18028  0.2421
#> Zsivoczky    0.53804  0.58597 -0.1427
#> Hernu        0.24729  0.06691 -0.2087
#> Bernard      0.00617  0.27949  0.3207
#> Schwarzl    -0.68705 -0.00836 -0.3021
#> Pogorelov   -0.03762 -0.13053 -0.0370
```

```
#> Schoenbeck  -0.25585  0.56422  0.2968
#> Barras       0.36555  0.10226  0.6119
res.ind$contrib        # Contributions to the PCs
#>               Dim.1  Dim.2    Dim.3   Dim.4    Dim.5   Dim.6    Dim.7
#> SEBRLE       0.0385  5.712 1.39e+00  0.0357 8.09e+00  2.2126  0.62143
#> CLAY         0.6581 13.854 6.46e+00  8.5557 4.03e+00 17.4880  0.65141
#> BERNARD      1.8627  6.144 1.35e-01 19.5783 4.20e-02  1.6502  2.37365
#> YURKOV       0.7969  0.443 2.15e+01  2.5794 9.86e-01  0.1088  1.51656
#> ZSIVOCZKY    0.0118  9.682 5.97e+00  0.0523 2.41e-01  8.2456  0.11192
#> McMULLEN     0.0148  2.325 2.50e+00  9.1353 1.56e+01  0.3579  3.28702
#> MARTINEAU    6.0337  3.904 2.83e+00  0.7386 3.12e+01  2.1441  1.29111
#> HERNU        3.7700  3.284 8.58e-02  4.2151 9.96e-02  1.9566  4.60485
#> BARRAS       3.3194  0.402 1.52e+00  0.2776 3.39e-01  0.0838  0.67004
#> NOOL         8.1299  5.849 1.29e+00 12.8376 1.26e+01  2.5413  4.06767
#> BOURGUIGNON 20.5373  3.776 3.53e-04  2.3588 1.09e+00  0.0691  5.14425
#> Sebrle      12.5584  3.502 9.88e+00  3.4701 3.89e-03  0.0705  0.00148
#> Clay        11.5936  6.232 1.34e+00  0.5325 2.01e+00  6.2097  0.95282
#> Karpov      17.4661  1.507 3.61e+00  0.5210 3.94e+00  1.1168 26.72016
#> Macey        3.6021 10.073 3.36e+00  1.8139 2.89e-07  0.4001  1.04191
#> Warners      2.0291  0.188 1.40e+01  5.5159 1.02e-02  3.1867  1.02738
#> Zsivoczky    0.2344  2.031 1.05e+01  0.1813 1.03e+00  0.1332  0.02289
#> Hernu        0.0805  3.214 1.02e-01  3.8217 2.95e+00  1.1311  2.82103
#> Bernard      1.9708  5.166 2.43e+00  2.9474 4.58e+00  1.1066  1.89945
#> Schwarzl     0.5318  1.025 3.80e+00  4.5612 5.17e-01  4.8796  4.59812
#> Pogorelov    0.0484  1.753 3.11e-01  9.7503 1.40e-03 26.1167  8.82532
#> Schoenbeck   0.2586  3.997 3.72e-02  0.3297 2.58e+00  0.2689 10.56627
#> Barras       0.1052  1.588 2.61e+00  1.8430 3.77e+00 14.1743 12.83542
#>                Dim.8    Dim.9  Dim.10
#> SEBRLE      2.99e-02 12.17748  0.0382
#> CLAY        6.04e+00  0.10126  3.5857
#> BERNARD     7.85e+00  0.81032  0.3499
#> YURKOV      2.09e-01  0.33601  0.5107
#> ZSIVOCZKY   6.49e-01  7.66492  4.3274
#> McMULLEN    1.36e+00  0.31250  5.5105
#> MARTINEAU   1.35e+00  1.39820 13.5440
#> HERNU       3.71e+01  0.00107 10.9278
#> BARRAS      7.38e+00 12.54733 10.0454
#> NOOL        2.98e+00  0.37548  0.3300
#> BOURGUIGNON 5.03e+00  7.74457  0.1222
#> Sebrle      1.48e-02 20.10555  1.7206
#> Clay        5.34e+00  4.70357 13.5271
#> Karpov      1.89e+00  1.04399  1.6819
#> Macey       2.09e+00  3.79877  1.6096
#> Warners     1.90e-01  0.91042  2.0890
#> Zsivoczky   4.59e+00  9.61785  0.7261
#> Hernu       9.69e-01  0.12540  1.5523
#> Bernard     6.02e-04  2.18807  3.6657
#> Schwarzl    7.48e+00  0.00196  3.2536
#> Pogorelov   2.24e-02  0.47727  0.0487
#> Schoenbeck  1.04e+00  8.91730  3.1402
#> Barras      2.12e+00  0.29289 13.3453
res.ind$cos2           # Quality of representation
#>               Dim.1  Dim.2    Dim.3   Dim.4    Dim.5    Dim.6    Dim.7
```

```
#> SEBRLE      0.00753 0.4975 8.13e-02 0.00139 2.69e-01 0.044324 8.91e-03
#> CLAY        0.04870 0.4570 1.44e-01 0.12579 5.08e-02 0.132691 3.54e-03
#> BERNARD     0.19720 0.2900 4.29e-03 0.41182 7.57e-04 0.017913 1.84e-02
#> YURKOV      0.09611 0.0238 7.78e-01 0.06181 2.02e-02 0.001345 1.34e-02
#> ZSIVOCZKY   0.00157 0.5764 2.40e-01 0.00139 5.47e-03 0.112918 1.10e-03
#> McMULLEN    0.00218 0.1522 1.10e-01 0.26649 3.89e-01 0.005388 3.54e-02
#> MARTINEAU   0.40401 0.1165 5.69e-02 0.00983 3.55e-01 0.014721 6.34e-03
#> HERNU       0.39928 0.1551 2.73e-03 0.08870 1.79e-03 0.021248 3.58e-02
#> BARRAS      0.61624 0.0333 8.48e-02 0.01024 1.07e-02 0.001594 9.13e-03
#> NOOL        0.48987 0.1571 2.34e-02 0.15369 1.29e-01 0.015701 1.80e-02
#> BOURGUIGNON 0.85970 0.0705 4.45e-06 0.01962 7.75e-03 0.000297 1.58e-02
#> Sebrle      0.67538 0.0840 1.60e-01 0.03708 3.56e-05 0.000389 5.85e-06
#> Clay        0.68759 0.1648 2.39e-02 0.00627 2.03e-02 0.037763 4.15e-03
#> Karpov      0.78367 0.0301 4.87e-02 0.00464 3.01e-02 0.005138 8.80e-02
#> Macey       0.36344 0.4531 1.02e-01 0.03636 4.95e-09 0.004140 7.71e-03
#> Warners     0.25565 0.0106 5.31e-01 0.13808 2.18e-04 0.041169 9.50e-03
#> Zsivoczky   0.04505 0.1740 6.05e-01 0.00692 3.36e-02 0.002625 3.23e-04
#> Hernu       0.02482 0.4418 9.46e-03 0.23420 1.55e-01 0.035771 6.38e-02
#> Bernard     0.28935 0.3381 1.07e-01 0.08598 1.14e-01 0.016659 2.05e-02
#> Schwarzl    0.11672 0.1003 2.51e-01 0.19889 1.93e-02 0.109806 7.40e-02
#> Pogorelov   0.00780 0.1259 1.50e-02 0.31210 3.85e-05 0.431416 1.04e-01
#> Schoenbeck  0.06707 0.4620 2.90e-03 0.01699 1.14e-01 0.007150 2.01e-01
#> Barras      0.01897 0.1277 1.41e-01 0.06604 1.16e-01 0.262130 1.70e-01
#>                Dim.8    Dim.9   Dim.10
#> SEBRLE      3.89e-04 8.95e-02 0.000221
#> CLAY        2.97e-02 2.82e-04 0.007847
#> BERNARD     5.53e-02 3.23e-03 0.001096
#> YURKOV      1.68e-03 1.53e-03 0.001822
#> ZSIVOCZKY   5.76e-03 3.85e-02 0.017092
#> McMULLEN    1.33e-02 1.73e-03 0.023927
#> MARTINEAU   6.00e-03 3.52e-03 0.026821
#> HERNU       2.61e-01 4.27e-06 0.034229
#> BARRAS      9.12e-02 8.77e-02 0.055153
#> NOOL        1.19e-02 8.51e-04 0.000588
#> BOURGUIGNON 1.40e-02 1.22e-02 0.000151
#> Sebrle      5.30e-05 4.07e-02 0.002737
#> Clay        2.11e-02 1.05e-02 0.023726
#> Karpov      5.64e-03 1.76e-03 0.002232
#> Macey       1.40e-02 1.44e-02 0.004803
#> Warners     1.59e-03 4.32e-03 0.007784
#> Zsivoczky   5.87e-02 6.96e-02 0.004127
#> Hernu       1.99e-02 1.46e-03 0.014160
#> Bernard     5.88e-06 1.21e-02 0.015917
#> Schwarzl    1.09e-01 1.62e-05 0.021117
#> Pogorelov   2.40e-04 2.89e-03 0.000232
#> Schoenbeck  1.79e-02 8.70e-02 0.024083
#> Barras      2.54e-02 1.99e-03 0.071184
```

1.8 Predict using PCA

In this section, we'll show how to predict the coordinates of supplementary individuals and variables using only the information provided by the previously performed PCA.

1. Data: rows 24 to 27 and columns 1 to to 10 [in decathlon2 data sets]. The new data must contain columns (variables) with the same names and in the same order as the active data used to compute PCA.

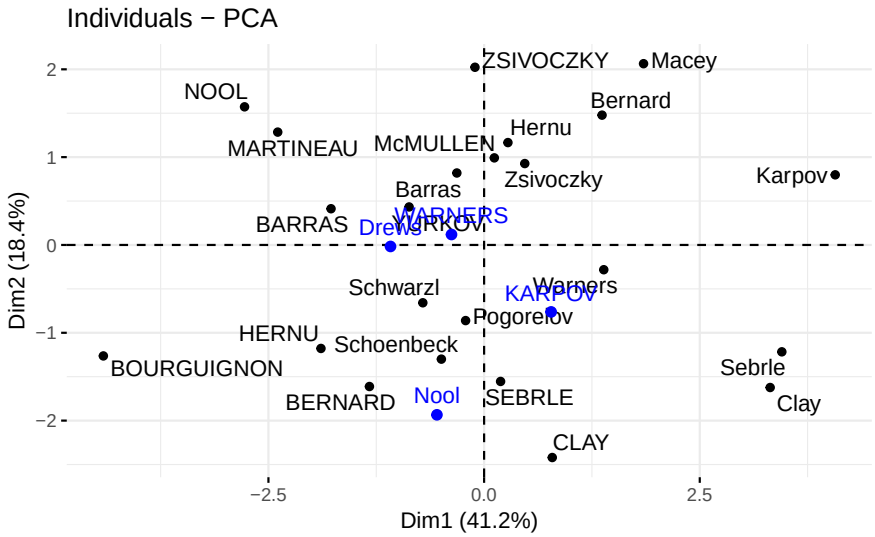
```
# Data for the supplementary individuals
ind.sup <- decathlon2[24:27, 1:10]
ind.sup[, 1:6]
#>          X100m Long.jump Shot.put High.jump X400m X110m.hurdle
#> KARPOV    11.0      7.30     14.8      2.04  48.4         14.1
#> WARNERS   11.1      7.60     14.3      1.98  48.7         14.2
#> Nool      10.8      7.53     14.3      1.88  48.8         14.8
#> Drews     10.9      7.38     13.1      1.88  48.5         14.0
```

2. Predict the coordinates of new individuals data. Use the R base function predict():

```
ind.sup.coord <- predict(res.pca, newdata = ind.sup)
ind.sup.coord[, 1:4]
#>             PC1    PC2   PC3    PC4
#> KARPOV    0.777 -0.762 1.597  1.686
#> WARNERS -0.378  0.119 1.701 -0.691
#> Nool     -0.547 -1.934 0.472 -2.228
#> Drews    -1.085 -0.017 2.982 -1.501
```

3. Graph of individuals including the supplementary individuals:

```
# Plot of active individuals
p <- fviz_pca_ind(res.pca, repel = TRUE)
# Add supplementary individuals
fviz_add(p, ind.sup.coord, color ="blue")
```



The predicted coordinates of individuals can be manually calculated as follow:

1. Center and scale the new individuals data using the center and the scale of the PCA
2. Calculate the predicted coordinates by multiplying the scaled values with the eigenvectors (loadings) of the principal components. The R code below can be used :

```
# Centering and scaling the supplementary individuals
ind.scaled <- scale(ind.sup,
                    center = res.pca$center,
                    scale = res.pca$scale)
```

```
# Coordinates of the individividuals
coord_func <- function(ind, loadings){
  r <- loadings*ind
  apply(r, 2, sum)
}
pca.loadings <- res.pca$rotation
ind.sup.coord <- t(apply(ind.scaled, 1, coord_func, pca.loadings ))
ind.sup.coord[, 1:4]
#>             PC1    PC2   PC3    PC4
#> KARPOV   0.777 -0.762 1.597  1.686
#> WARNERS -0.378  0.119 1.701 -0.691
#> Nool    -0.547 -1.934 0.472 -2.228
#> Drews   -1.085 -0.017 2.982 -1.501
```

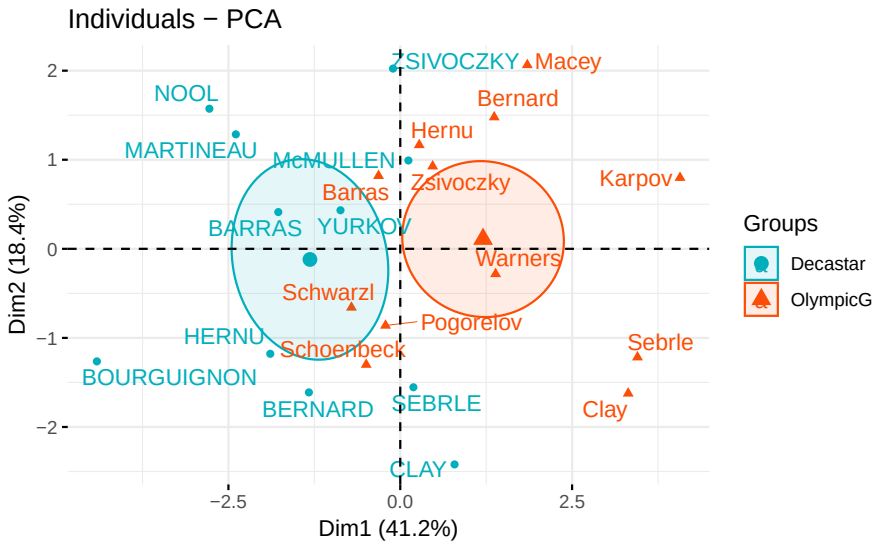
1.9 Supplementary variables

1.9.1 Qualitative / categorical variables

The data sets decathlon2 contain a supplementary qualitative variable at columns 13 corresponding to the type of competitions.

Qualitative / categorical variables can be used to color individuals by groups. The grouping variable should be of same length as the number of active individuals (here 23).

```
groups <- as.factor(decathlon2$Competition[1:23])
fviz_pca_ind(res.pca,
             col.ind = groups, # color by groups
             palette = c("#00AFBB",  "#FC4E07"),
             addEllipses = TRUE, # Concentration ellipses
             ellipse.type = "confidence",
             legend.title = "Groups",
             repel = TRUE
             )
```



Calculate the coordinates for the levels of grouping variables. The coordinates for a given group is calculated as the mean coordinates of the individuals in the group.

```
library(magrittr) # for pipe %>%
library(dplyr)    # everything else
#>
#> 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

# 1. Individual coordinates
res.ind <- get_pca_ind(res.pca)
# 2. Coordinate of groups
coord.groups <- res.ind$coord %>%
  as_data_frame() %>%
  select(Dim.1, Dim.2) %>%
  mutate(competition = groups) %>%
  group_by(competition) %>%
  summarise(
    Dim.1 = mean(Dim.1),
    Dim.2 = mean(Dim.2)
    )
#> Warning: `as_data_frame()` is deprecated, use `as_tibble()` (but mind the new semantics).
#> This warning is displayed once per session.
coord.groups
#> # A tibble: 2 x 3
#>   competition Dim.1  Dim.2
#>   <fct>       <dbl>  <dbl>
#> 1 Decastar    -1.31 -0.119
#> 2 OlympicG     1.20  0.109
```

1.9.2 Quantitative variables

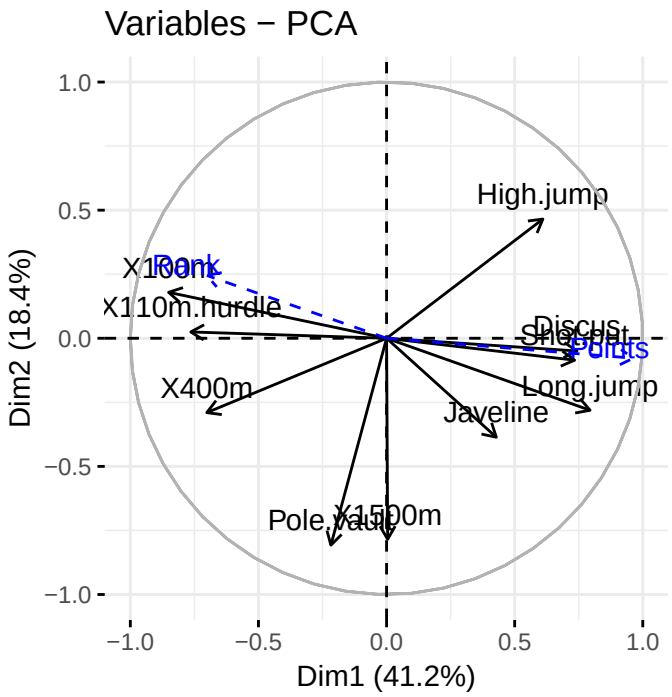
Data: columns 11:12. Should be of same length as the number of active individuals (here 23)

```
quanti.sup <- decathlon2[1:23, 11:12, drop = FALSE]
head(quanti.sup)
#>           Rank Points
#> SEBRLE       1   8217
#> CLAY         2   8122
#> BERNARD      4   8067
#> YURKOV       5   8036
#> ZSIVOCZKY    7   8004
#> McMULLEN     8   7995
```

The coordinates of a given quantitative variable are calculated as the correlation between the quantitative variables and the principal components.

```
# Predict coordinates and compute cos2
quanti.coord <- cor(quanti.sup, res.pca$x)
quanti.cos2 <- quanti.coord^2
# Graph of variables including supplementary variables
p <- fviz_pca_var(res.pca)
```

```
fviz_add(p, quanti.coord, color ="blue", geom="arrow")
```



1.10 Theory behind PCA results

1.10.1 PCA results for variables

Here we'll show how to calculate the PCA results for variables: coordinates, cos2 and contributions:

`var.coord` = loadings * the component standard deviations `var.cos2` = var.coord^2 `var.contrib`. The contribution of a variable to a given principal component is (in percentage) : (var.cos2 * 100) / (total cos2 of the component)

```
# Helper function
#:::::::::::::::::::::::::::::::::::::
var_coord_func <- function(loadings, comp.sdev){
  loadings*comp.sdev
}
```

```
# Compute Coordinates
#:::::::::::::::::::::::::::::::::::::
loadings <- res.pca$rotation
sdev <- res.pca$sdev
var.coord <- t(apply(loadings, 1, var_coord_func, sdev))
head(var.coord[, 1:4])
#>                 PC1     PC2    PC3     PC4
#> X100m        -0.851  0.1794 -0.302  0.0336
#> Long.jump     0.794 -0.2809  0.191 -0.1154
#> Shot.put      0.734 -0.0854 -0.518  0.1285
#> High.jump     0.610  0.4652 -0.330  0.1446
#> X400m        -0.702 -0.2902 -0.284  0.4308
#> X110m.hurdle -0.764  0.0247 -0.449 -0.0169
```

```
# Compute Cos2
#:::::::::::::::::::::::::::::::::::::
var.cos2 <- var.coord^2
head(var.cos2[, 1:4])
#>                PC1      PC2    PC3      PC4
#> X100m        0.724 0.032184 0.0909 0.001127
#> Long.jump    0.631 0.078881 0.0363 0.013315
#> Shot.put     0.539 0.007294 0.2679 0.016504
#> High.jump    0.372 0.216424 0.1090 0.020895
#> X400m        0.492 0.084203 0.0804 0.185611
#> X110m.hurdle 0.584 0.000612 0.2015 0.000285
```

```
# Compute contributions
#::::::::::::::::::::::::::::::::::::::
comp.cos2 <- apply(var.cos2, 2, sum)
contrib <- function(var.cos2, comp.cos2){var.cos2*100/comp.cos2}
var.contrib <- t(apply(var.cos2,1, contrib, comp.cos2))
head(var.contrib[, 1:4])
#>                PC1     PC2   PC3     PC4
#> X100m        17.54  1.7505  7.34  0.1376
#> Long.jump    15.29  4.2904  2.93  1.6249
#> Shot.put     13.06  0.3967 21.62  2.0141
#> High.jump     9.02 11.7716  8.79  2.5499
#> X400m        11.94  4.5799  6.49 22.6509
#> X110m.hurdle 14.16  0.0333 16.26  0.0348
```

1.10.2 PCA results for individuals

- `ind.coord` = res.pca$x
- Cos2 of individuals. Two steps:
  - Calculate the square distance between each individual and the PCA center of gravity: d2 = [(var1_ind_i - mean_var1)/sd_var1]^2 + …+ [(var10_ind_i - mean_var10)/sd_var10]^2 + …+..
  - Calculate the cos2 as ind.coord^2/d2
- Contributions of individuals to the principal components: 100 * (1 / number_of_individuals)*(ind.coord^2 / comp_sdev^2). Note that the sum of all the contributions per column is 100

```
# Coordinates of individuals
#:::::::::::::::::::::::::::::::
ind.coord <- res.pca$x
head(ind.coord[, 1:4])
#>              PC1    PC2    PC3     PC4
#> SEBRLE     0.191 -1.554 -0.628  0.0821
#> CLAY       0.790 -2.420  1.357  1.2698
#> BERNARD   -1.329 -1.612 -0.196 -1.9209
#> YURKOV    -0.869  0.433 -2.474  0.6972
#> ZSIVOCZKY -0.106  2.023  1.305 -0.0993
#> McMULLEN   0.119  0.992  0.844  1.3122
```

```
# Cos2 of individuals
#:::::::::::::::::::::::::::::::
# 1. square of the distance between an individual and the
# PCA center of gravity
```

```
center <- res.pca$center
scale<- res.pca$scale

getdistance <- function(ind_row, center, scale){
  return(sum(((ind_row-center)/scale)^2))
}

d2 <- apply(decathlon2.active,1, getdistance, center, scale)
# 2. Compute the cos2. The sum of each row is 1
cos2 <- function(ind.coord, d2){return(ind.coord^2/d2)}
ind.cos2 <- apply(ind.coord, 2, cos2, d2)
head(ind.cos2[, 1:4])
#>                PC1    PC2     PC3     PC4
#> SEBRLE    0.00753 0.4975 0.08133 0.00139
#> CLAY      0.04870 0.4570 0.14363 0.12579
#> BERNARD   0.19720 0.2900 0.00429 0.41182
#> YURKOV    0.09611 0.0238 0.77823 0.06181
#> ZSIVOCZKY 0.00157 0.5764 0.23975 0.00139
#> McMULLEN  0.00218 0.1522 0.11014 0.26649
```

```
# Contributions of individuals
#::::::::::::::::::::::::::::::
contrib <- function(ind.coord, comp.sdev, n.ind){
  100*(1/n.ind)*ind.coord^2/comp.sdev^2
}
ind.contrib <- t(apply(ind.coord, 1, contrib,
                       res.pca$sdev, nrow(ind.coord)))
head(ind.contrib[, 1:4])
#>              PC1    PC2    PC3     PC4
#> SEBRLE    0.0385  5.712  1.385  0.0357
#> CLAY      0.6581 13.854  6.460  8.5557
#> BERNARD   1.8627  6.144  0.135 19.5783
#> YURKOV    0.7969  0.443 21.476  2.5794
#> ZSIVOCZKY 0.0118  9.682  5.975  0.0523
#> McMULLEN  0.0148  2.325  2.497  9.1353
```

Chapter 2

Principal Components Methods

http://www.sthda.com/english/articles/31-principal-component-methods-in-r-practical-guide/112-pca-principal-component

Principal component analysis (PCA) allows us to summarize and to visualize the information in a data set containing individuals/observations described by multiple inter-correlated quantitative variables. Each variable could be considered as a different dimension. If you have more than 3 variables in your data sets, it could be very difficult to visualize a multi-dimensional hyperspace.

Principal component analysis is used to extract the important information from a multivariate data table and to express this information as a set of few new variables called principal components. These new variables correspond to a linear combination of the originals. The number of principal components is less than or equal to the number of original variables.

The information in a given data set corresponds to the total variation it contains. The goal of PCA is to identify directions (or principal components) along which the variation in the data is maximal.

In other words, PCA reduces the dimensionality of a multivariate data to two or three principal components, that can be visualized graphically, with minimal loss of information.

```
# install.packages(c("FactoMineR", "factoextra"))
```

```
library(FactoMineR)
library(factoextra)
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
#> Welcome! Related Books: `Practical Guide To Cluster Analysis in R` at https://goo.gl/13EFCZ
```

```
data(decathlon2)
# head(decathlon2)
```

In PCA terminology, our data contains :

- Active individuals (in light blue, rows 1:23) : Individuals that are used during the principal component analysis.
- Supplementary individuals (in dark blue, rows 24:27) : The coordinates of these individuals will be predicted using the PCA information and parameters obtained with active individuals/variables
- Active variables (in pink, columns 1:10) : Variables that are used for the principal component analysis.

- Supplementary variables: As supplementary individuals, the coordinates of these variables will be predicted also. These can be:
  - Supplementary continuous variables (red): Columns 11 and 12 corresponding respectively to the rank and the points of athletes.
  - Supplementary qualitative variables (green): Column 13 corresponding to the two athlete-tic meetings (2004 Olympic Game or 2004 Decastar). This is a categorical (or factor) variable factor. It can be used to color individuals by groups.

We start by subsetting active individuals and active variables for the principal component analysis:

```
decathlon2.active <- decathlon2[1:23, 1:10]
head(decathlon2.active[, 1:6], 4)
#>          X100m Long.jump Shot.put High.jump X400m X110m.hurdle
#> SEBRLE    11.0      7.58     14.8      2.07  49.8         14.7
#> CLAY      10.8      7.40     14.3      1.86  49.4         14.1
#> BERNARD   11.0      7.23     14.2      1.92  48.9         15.0
#> YURKOV    11.3      7.09     15.2      2.10  50.4         15.3
```

2.1 Data standardization

In principal component analysis, variables are often scaled (i.e. standardized). This is particularly recommended when variables are measured in different scales (e.g: kilograms, kilometers, centimeters, …); otherwise, the PCA outputs obtained will be severely affected.

The goal is to make the variables comparable. Generally variables are scaled to have i) standard deviation one and ii) mean zero.

The function PCA() [FactoMineR package] can be used. A simplified format is:

```
library(FactoMineR)
res.pca <- PCA(decathlon2.active, graph = FALSE)
```

```
print(res.pca)
#> **Results for the Principal Component Analysis (PCA)**
#> The analysis was performed on 23 individuals, described by 10 variables
#> *The results are available in the following objects:
#>
#>    name               description
#> 1  "$eig"             "eigenvalues"
#> 2  "$var"             "results for the variables"
#> 3  "$var$coord"       "coord. for the variables"
#> 4  "$var$cor"         "correlations variables - dimensions"
#> 5  "$var$cos2"        "cos2 for the variables"
#> 6  "$var$contrib"     "contributions of the variables"
#> 7  "$ind"             "results for the individuals"
#> 8  "$ind$coord"       "coord. for the individuals"
#> 9  "$ind$cos2"        "cos2 for the individuals"
#> 10 "$ind$contrib"     "contributions of the individuals"
#> 11 "$call"            "summary statistics"
#> 12 "$call$centre"     "mean of the variables"
#> 13 "$call$ecart.type" "standard error of the variables"
#> 14 "$call$row.w"      "weights for the individuals"
#> 15 "$call$col.w"      "weights for the variables"
```

The object that is created using the function PCA() contains many information found in many different lists and matrices. These values are described in the next section.

2.2 Eigenvalues / Variances

As described in previous sections, the eigenvalues measure the amount of variation retained by each principal component. Eigenvalues are large for the first PCs and small for the subsequent PCs. That is, the first PCs corresponds to the directions with the maximum amount of variation in the data set.

We examine the eigenvalues to determine the number of principal components to be considered. The eigenvalues and the proportion of variances (i.e., information) retained by the principal components (PCs) can be extracted using the function get_eigenvalue() [factoextra package].

```
library(factoextra)
eig.val <- get_eigenvalue(res.pca)
eig.val
#>        eigenvalue variance.percent cumulative.variance.percent
#> Dim.1       4.124            41.24                        41.2
#> Dim.2       1.839            18.39                        59.6
#> Dim.3       1.239            12.39                        72.0
#> Dim.4       0.819             8.19                        80.2
#> Dim.5       0.702             7.02                        87.2
#> Dim.6       0.423             4.23                        91.5
#> Dim.7       0.303             3.03                        94.5
#> Dim.8       0.274             2.74                        97.2
#> Dim.9       0.155             1.55                        98.8
#> Dim.10      0.122             1.22                       100.0
```

The sum of all the eigenvalues give a total variance of 10.

The proportion of variation explained by each eigenvalue is given in the second column. For example, 4.124 divided by 10 equals 0.4124, or, about 41.24% of the variation is explained by this first eigenvalue. The cumulative percentage explained is obtained by adding the successive proportions of variation explained to obtain the running total. For instance, 41.242% plus 18.385% equals 59.627%, and so forth. Therefore, about 59.627% of the variation is explained by the first two eigenvalues together.

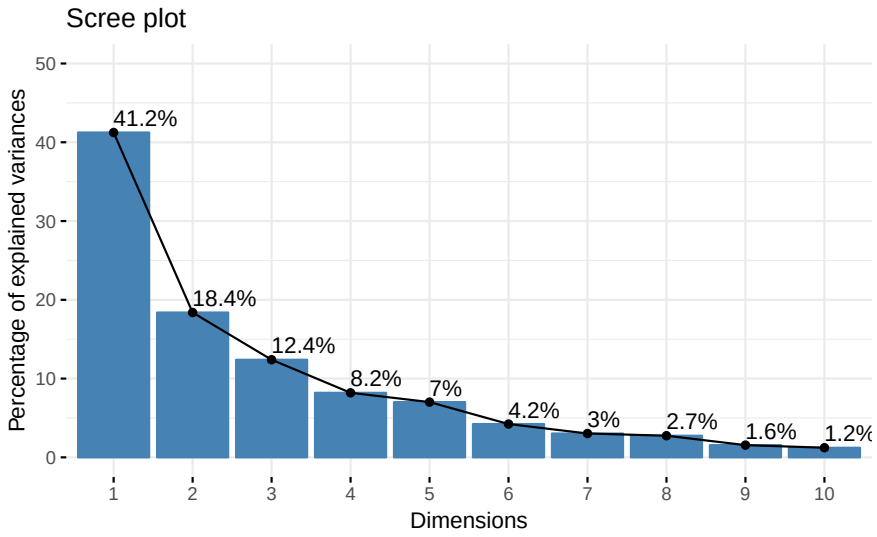
Unfortunately, there is no well-accepted objective way to decide how many principal components are enough. This will depend on the specific field of application and the specific data set. In practice, we tend to look at the first few principal components in order to find interesting patterns in the data.

In our analysis, the first three principal components explain 72% of the variation. This is an acceptably large percentage.

An alternative method to determine the number of principal components is to look at a Scree Plot, which is the plot of eigenvalues ordered from largest to the smallest. The number of component is determined at the point, beyond which the remaining eigenvalues are all relatively small and of comparable size (Jollife 2002, Peres-Neto, Jackson, and Somers (2005)).

The scree plot can be produced using the function fviz_eig() or fviz_screeplot() [factoextra package].

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



From the plot above, we might want to stop at the fifth principal component. 87% of the information (variances) contained in the data are retained by the first five principal components.

2.3 Graph of variables

Results A simple method to extract the results, for variables, from a PCA output is to use the function get_pca_var() [factoextra package]. This function provides a list of matrices containing all the results for the active variables (coordinates, correlation between variables and axes, squared cosine and contributions)

```
var <- get_pca_var(res.pca)
var
#> 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"
```

The components of the get_pca_var() can be used in the plot of variables as follow:

- var$coord: coordinates of variables to create a scatter plot
- var$cos2: represents the quality of representation for variables on the factor map. It's calculated as the squared coordinates: var.cos2 = var.coord * var.coord.
- var$contrib: contains the contributions (in percentage) of the variables to the principal components. The contribution of a variable (var) to a given principal component is (in percentage) : (var.cos2 * 100) / (total cos2 of the component).

  Note that, it's possible to plot variables and to color them according to either i) their quality on the factor map (cos2) or ii) their contribution values to the principal components (contrib).

The different components can be accessed as follow:

```
# Coordinates
head(var$coord)
#>              Dim.1   Dim.2  Dim.3   Dim.4  Dim.5
#> X100m       -0.851 -0.1794  0.302  0.0336 -0.194
#> Long.jump    0.794  0.2809 -0.191 -0.1154  0.233
```

```
#> Shot.put      0.734  0.0854  0.518  0.1285 -0.249
#> High.jump     0.610 -0.4652  0.330  0.1446  0.403
#> X400m        -0.702  0.2902  0.284  0.4308  0.104
#> X110m.hurdle -0.764 -0.0247  0.449 -0.0169  0.224
# Cos2: quality on the factore map
head(var$cos2)
#>              Dim.1    Dim.2  Dim.3    Dim.4  Dim.5
#> X100m        0.724 0.032184 0.0909 0.001127 0.0378
#> Long.jump    0.631 0.078881 0.0363 0.013315 0.0544
#> Shot.put     0.539 0.007294 0.2679 0.016504 0.0619
#> High.jump    0.372 0.216424 0.1090 0.020895 0.1622
#> X400m        0.492 0.084203 0.0804 0.185611 0.0108
#> X110m.hurdle 0.584 0.000612 0.2015 0.000285 0.0503
# Contributions to the principal components
head(var$contrib)
#>              Dim.1   Dim.2 Dim.3   Dim.4 Dim.5
#> X100m        17.54  1.7505  7.34  0.1376  5.39
#> Long.jump    15.29  4.2904  2.93  1.6249  7.75
#> Shot.put     13.06  0.3967 21.62  2.0141  8.82
#> High.jump     9.02 11.7716  8.79  2.5499 23.12
#> X400m        11.94  4.5799  6.49 22.6509  1.54
#> X110m.hurdle 14.16  0.0333 16.26  0.0348  7.17
```

In this section, we describe how to visualize variables and draw conclusions about their correlations. Next, we highlight variables according to either i) their quality of representation on the factor map or ii) their contributions to the principal components.

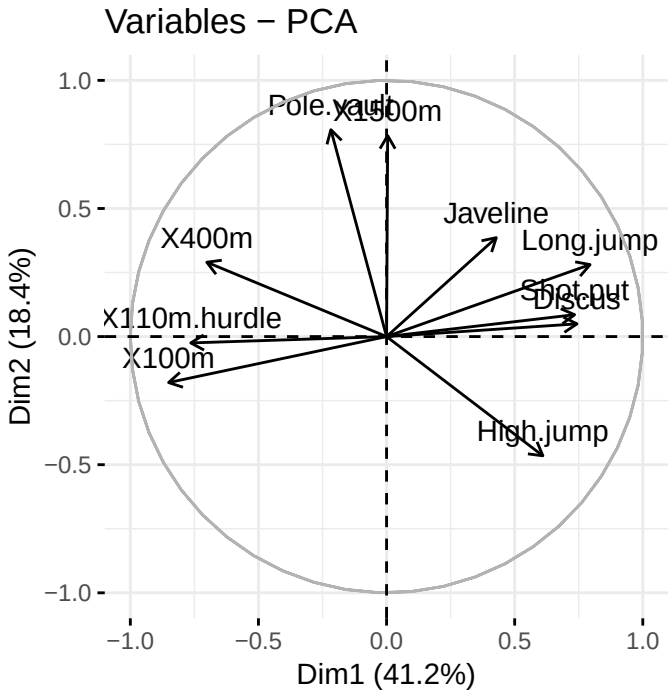
2.4 Correlation circle

The correlation between a variable and a principal component (PC) is used as the coordinates of the variable on the PC. The representation of variables differs from the plot of the observations: The observations are represented by their projections, but the variables are represented by their correlations (Abdi and Williams 2010).

```
# Coordinates of variables
head(var$coord, 4)
#>             Dim.1   Dim.2  Dim.3   Dim.4  Dim.5
#> X100m     -0.851 -0.1794  0.302  0.0336 -0.194
#> Long.jump  0.794  0.2809 -0.191 -0.1154  0.233
#> Shot.put   0.734  0.0854  0.518  0.1285 -0.249
#> High.jump  0.610 -0.4652  0.330  0.1446  0.403
```

To plot variables, type this:

```
fviz_pca_var(res.pca, col.var = "black")
```



The plot above is also known as variable correlation plots. It shows the relationships between all variables. It can be interpreted as follow:

- Positively correlated variables are grouped together.
- Negatively correlated variables are positioned on opposite sides of the plot origin (opposed quadrants).
- The distance between variables and the origin measures the quality of the variables on the factor map. Variables that are away from the origin are well represented on the factor map.

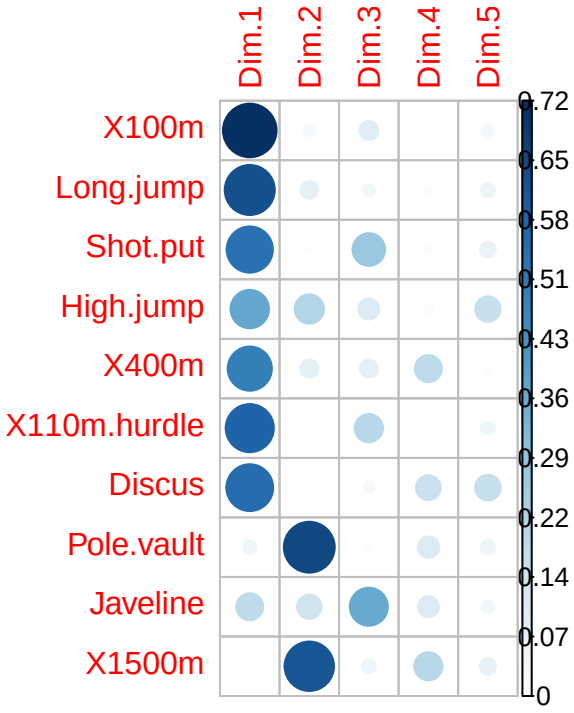
2.5 Quality of representation

The quality of representation of the variables on factor map is called cos2 (square cosine, squared coordinates) . You can access to the cos2 as follow:

```
head(var$cos2, 4)
#>           Dim.1   Dim.2  Dim.3   Dim.4  Dim.5
#> X100m     0.724 0.03218 0.0909 0.00113 0.0378
#> Long.jump 0.631 0.07888 0.0363 0.01331 0.0544
#> Shot.put  0.539 0.00729 0.2679 0.01650 0.0619
#> High.jump 0.372 0.21642 0.1090 0.02089 0.1622
```

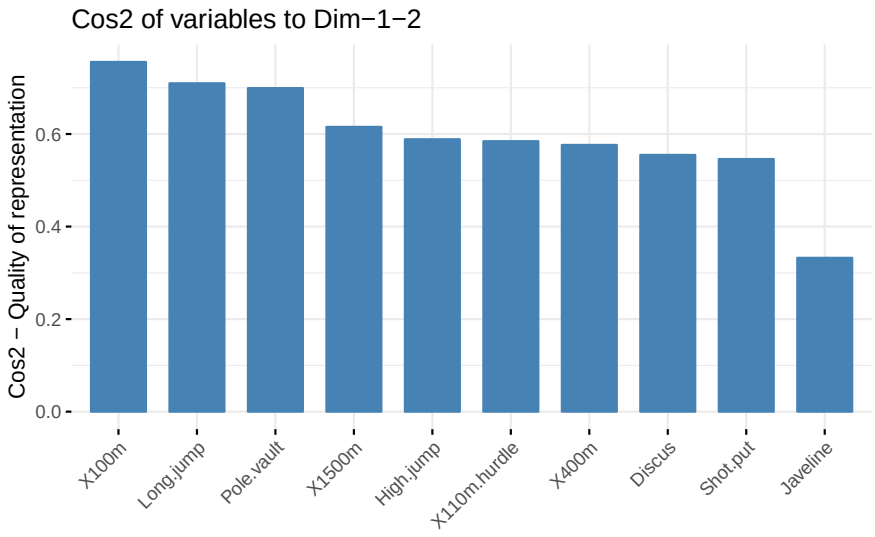
You can visualize the cos2 of variables on all the dimensions using the corrplot package:

```
library(corrplot)
#> corrplot 0.84 loaded
corrplot(var$cos2, is.corr=FALSE)
```



It's also possible to create a bar plot of variables cos2 using the function `fviz_cos2()` [in factoextra]:

```
# Total cos2 of variables on Dim.1 and Dim.2
fviz_cos2(res.pca, choice = "var", axes = 1:2)
```



Note that,

- 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.

If a variable is perfectly represented by only two principal components (Dim.1 & Dim.2), the sum of the cos2 on these two PCs is equal to one. In this case the variables will be positioned on the circle of correlations.

For some of the variables, more than 2 components might be required to perfectly represent the data. In this case the variables are positioned inside the circle of correlations.

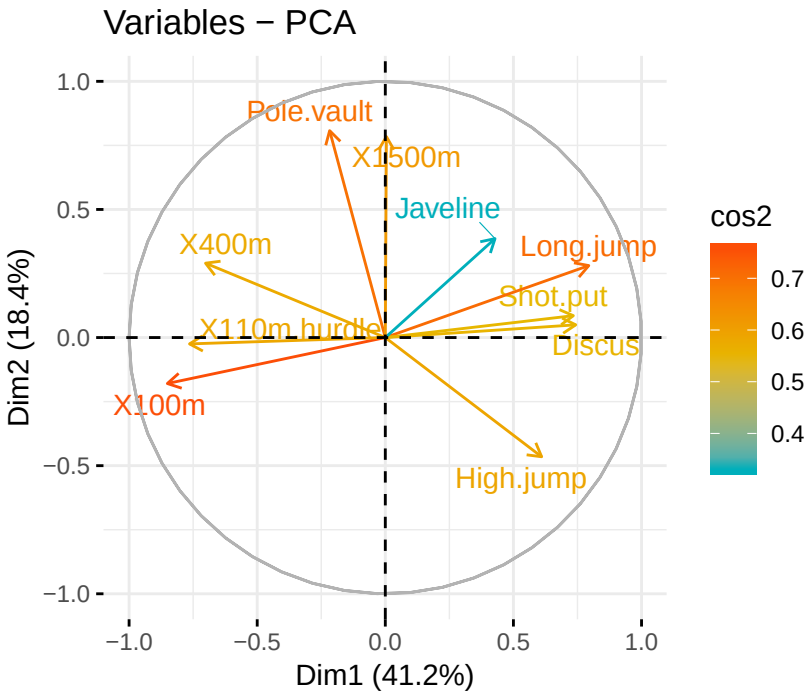
In summary:

- The cos2 values are used to estimate the quality of the representation
- The closer a variable is to the circle of correlations, the better its representation on the factor map (and the more important it is to interpret these components)
- Variables that are closed to the center of the plot are less important for the first components.

It's possible to color variables by their cos2 values using the argument col.var = "cos2". This produces a gradient colors. In this case, the argument gradient.cols can be used to provide a custom color. For instance, gradient.cols = c("white", "blue", "red") means that:

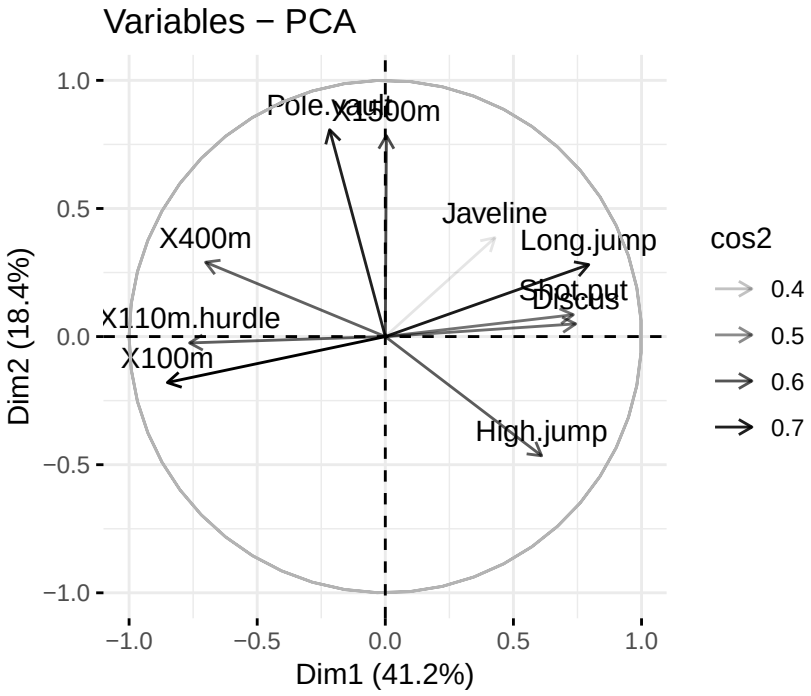
- variables with low cos2 values will be colored in "white"
- variables with mid cos2 values will be colored in "blue"
- variables with high cos2 values will be colored in red

```
# Color by cos2 values: quality on the factor map
fviz_pca_var(res.pca, col.var = "cos2",
             gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"),
             repel = TRUE # Avoid text overlapping
             )
```



Note that, it's also possible to change the transparency of the variables according to their cos2 values using the option alpha.var = "cos2". For example, type this:

```
# Change the transparency by cos2 values
fviz_pca_var(res.pca, alpha.var = "cos2")
```



2.6 Contributions of variables to PCs

The contributions of variables in accounting for the variability in a given principal component are expressed in percentage.

- Variables that are correlated with PC1 (i.e., Dim.1) and PC2 (i.e., Dim.2) are the most important in explaining the variability in the data set.
- Variables that do not correlated with any PC or correlated with the last dimensions are variables with low contribution and might be removed to simplify the overall analysis.

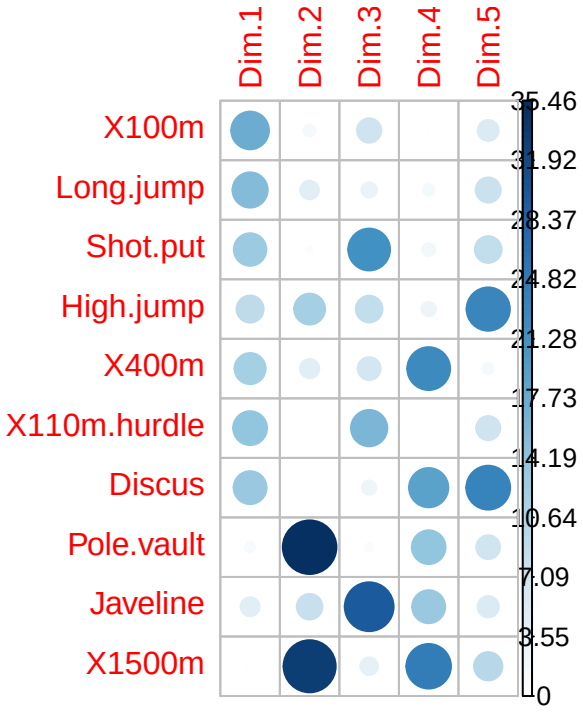
The contribution of variables can be extracted as follow :

```
head(var$contrib, 4)
#>           Dim.1  Dim.2 Dim.3 Dim.4 Dim.5
#> X100m     17.54  1.751  7.34 0.138  5.39
#> Long.jump 15.29  4.290  2.93 1.625  7.75
#> Shot.put  13.06  0.397 21.62 2.014  8.82
#> High.jump  9.02 11.772  8.79 2.550 23.12
```

The larger the value of the contribution, the more the variable contributes to the component.

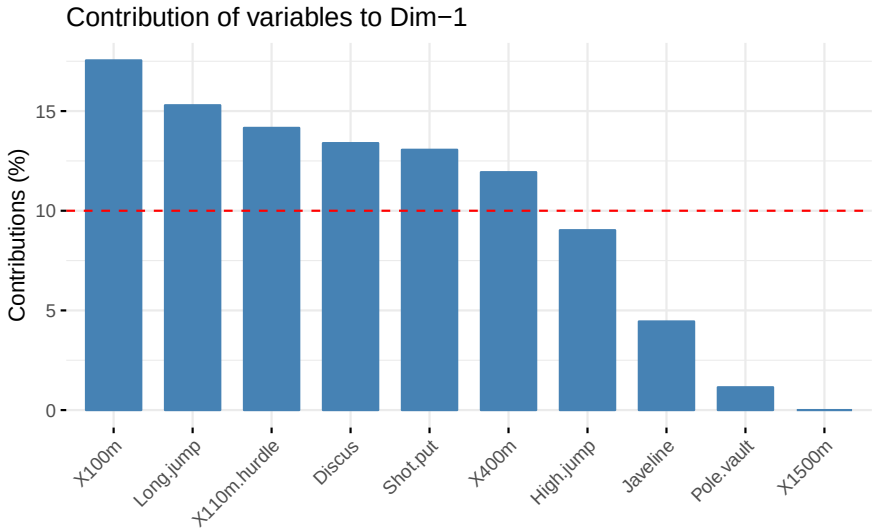
It's possible to use the function corrplot() [corrplot package] to highlight the most contributing variables for each dimension:

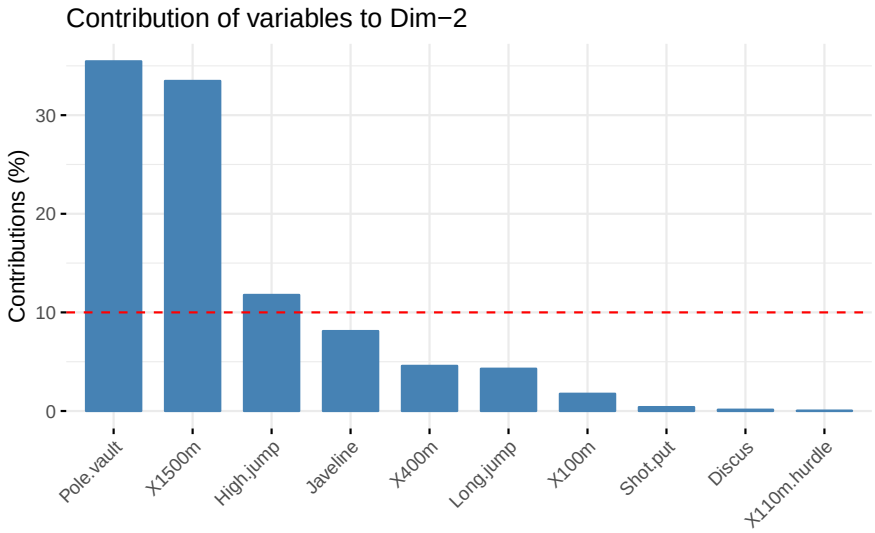
```
library("corrplot")

corrplot(var$contrib, is.corr=FALSE)
```



The function fviz_contrib() [factoextra package] can be used to draw a bar plot of variable contributions. If your data contains many variables, you can decide to show only the top contributing variables. The R code below shows the top 10 variables contributing to the principal components:

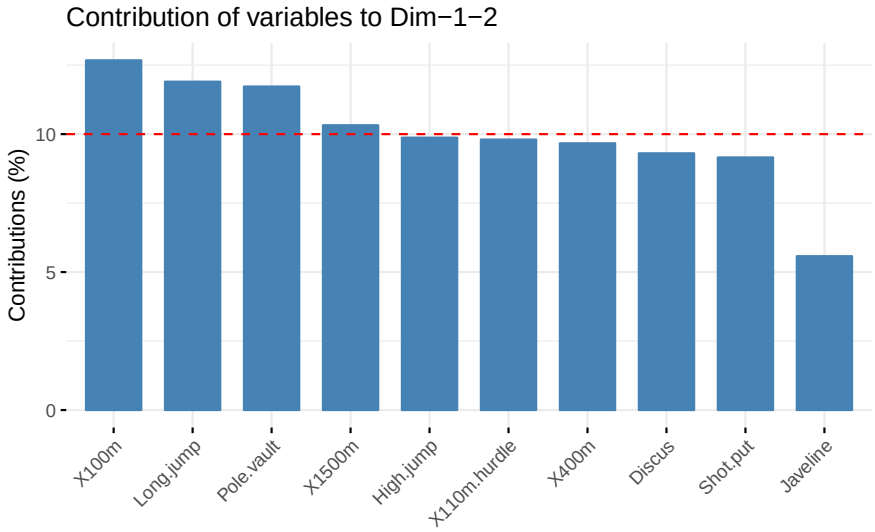
```
# Contributions of variables to PC1
fviz_contrib(res.pca, choice = "var", axes = 1, top = 10)
# Contributions of variables to PC2
fviz_contrib(res.pca, choice = "var", axes = 2, top = 10)
```





The total contribution to PC1 and PC2 is obtained with the following R code:

```
fviz_contrib(res.pca, choice = "var", axes = 1:2, top = 10)
```



The red dashed line on the graph above indicates the expected average contribution. If the contribution of the variables were uniform, the expected value would be 1/length(variables) = 1/10 = 10%. For a given component, a variable with a contribution larger than this cutoff could be considered as important in contributing to the component.

Note that, the total contribution of a given variable, on explaining the variations retained by two principal components, say PC1 and PC2, is calculated as contrib = [(C1 * Eig1) + (C2 * Eig2)]/(Eig1 + Eig2), where

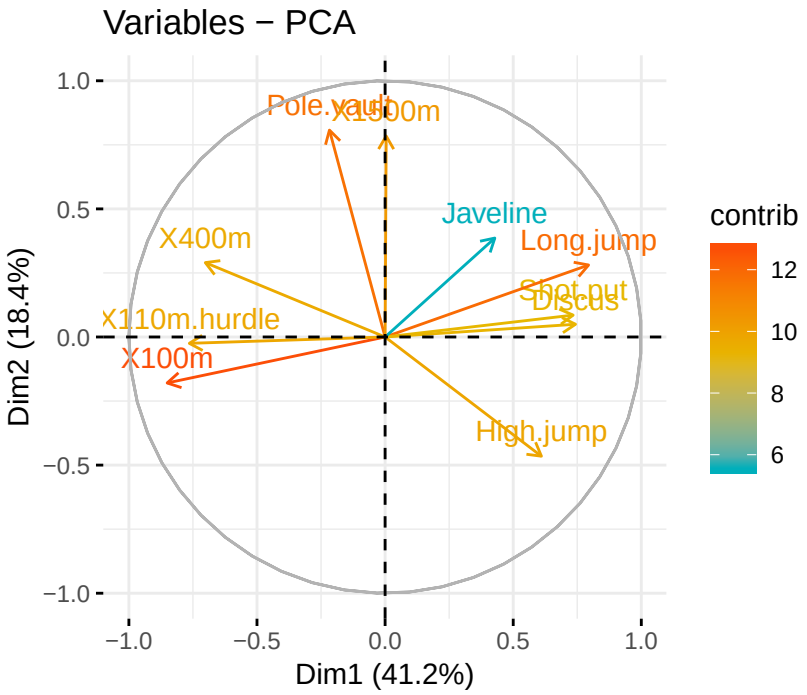
- C1 and C2 are the contributions of the variable on PC1 and PC2, respectively
- Eig1 and Eig2 are the eigenvalues of PC1 and PC2, respectively. Recall that eigenvalues measure the amount of variation retained by each PC.

In this case, the expected average contribution (cutoff) is calculated as follow: As mentioned above, if the contributions of the 10 variables were uniform, the expected average contribution on a given PC would be 1/10 = 10%. The expected average contribution of a variable for PC1 and PC2 is : [(10* Eig1) + (10 * Eig2)]/(Eig1 + Eig2)

It can be seen that the variables - X100m, Long.jump and Pole.vault - contribute the most to the dimensions 1 and 2.

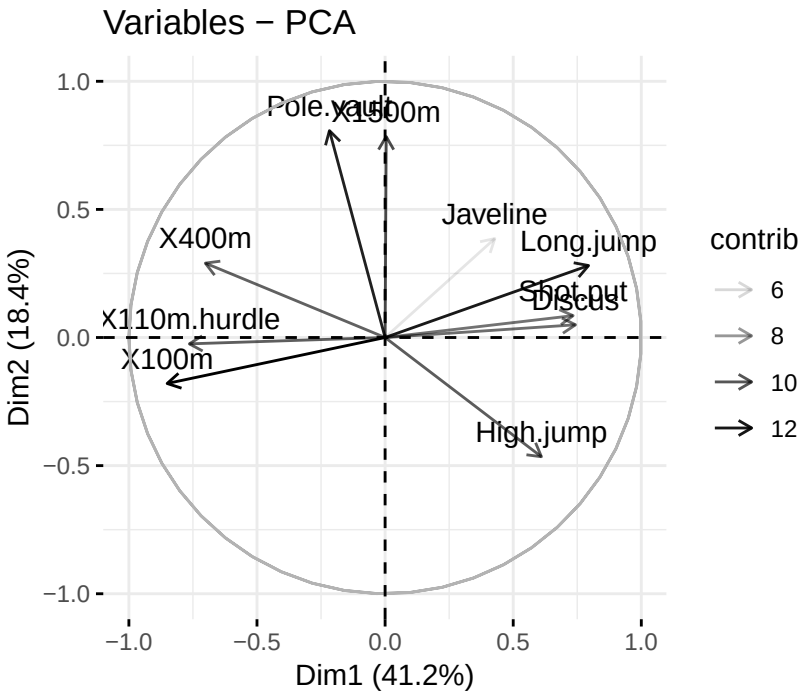
The most important (or, contributing) variables can be highlighted on the correlation plot as follow:

```
fviz_pca_var(res.pca, col.var = "contrib",
             gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07")
             )
```



Note that, it's also possible to change the transparency of variables according to their contrib values using the option alpha.var = "contrib". For example, type this:

```
# Change the transparency by contrib values
fviz_pca_var(res.pca, alpha.var = "contrib")
```

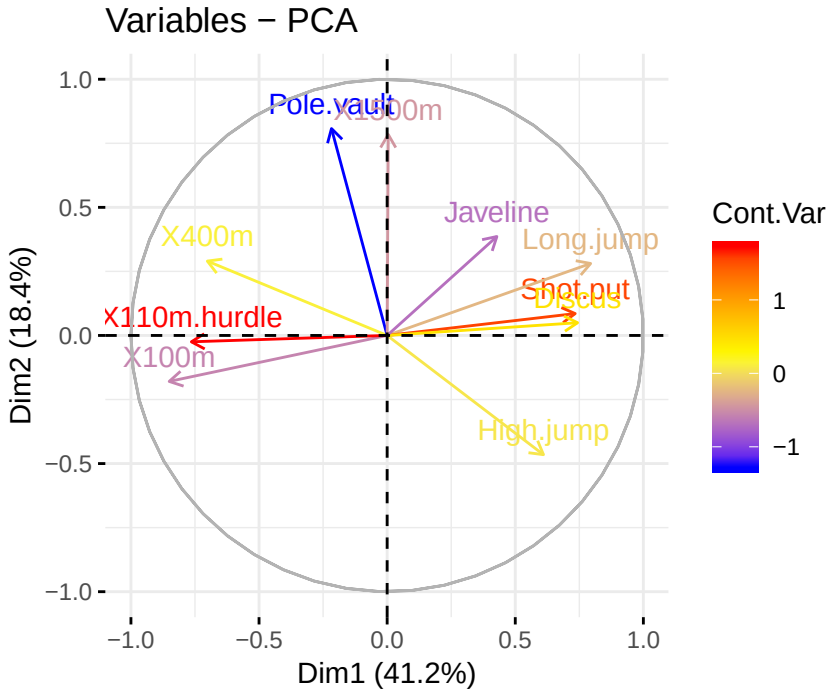


2.7 Color by a custom continuous variable

In the previous sections, we showed how to color variables by their contributions and their cos2. Note that, it's possible to color variables by any custom continuous variable. The coloring variable should have the same length as the number of active variables in the PCA (here n = 10).

For example, type this:

```
# Create a random continuous variable of length 10
set.seed(123)
my.cont.var <- rnorm(10)
# Color variables by the continuous variable
fviz_pca_var(res.pca, col.var = my.cont.var,
             gradient.cols = c("blue", "yellow", "red"),
             legend.title = "Cont.Var")
```



2.8 Color by groups

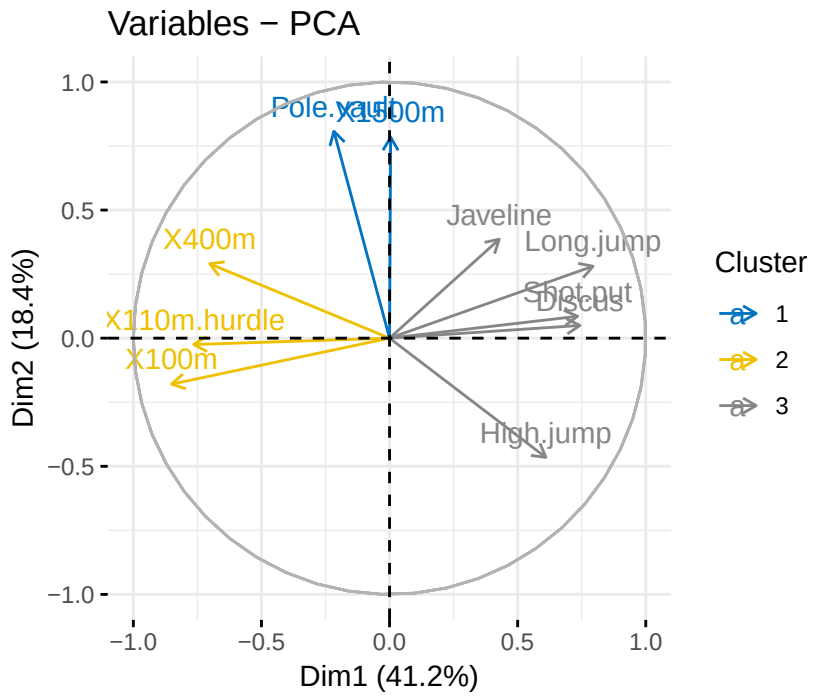
It's also possible to change the color of variables by groups defined by a qualitative/categorical variable, also called factor in R terminology.

As we don't have any grouping variable in our data sets for classifying variables, we'll create it.

In the following demo example, we start by classifying the variables into 3 groups using the kmeans clustering algorithm. Next, we use the clusters returned by the kmeans algorithm to color variables.

```
# Create a grouping variable using kmeans
# Create 3 groups of variables (centers = 3)
set.seed(123)
res.km <- kmeans(var$coord, centers = 3, nstart = 25)
grp <- as.factor(res.km$cluster)
# Color variables by groups
```

```
fviz_pca_var(res.pca, col.var = grp,
             palette = c("#0073C2FF", "#EFC000FF", "#868686FF"),
             legend.title = "Cluster")
```



2.9 Dimension description

In the section **?**(pca-variable-contributions), we described how to highlight variables according to their contributions to the principal components.

Note also that, the function dimdesc() [in FactoMineR], for dimension description, can be used to identify the most significantly associated variables with a given principal component . It can be used as follow:

```
res.desc <- dimdesc(res.pca, axes = c(1,2), proba = 0.05)
# Description of dimension 1
res.desc$Dim.1
#> $quanti
#>               correlation  p.value
#> Long.jump           0.794 6.06e-06
#> Discus              0.743 4.84e-05
#> Shot.put            0.734 6.72e-05
#> High.jump           0.610 1.99e-03
#> Javeline            0.428 4.15e-02
#> X400m              -0.702 1.91e-04
#> X110m.hurdle       -0.764 2.20e-05
#> X100m              -0.851 2.73e-07
```

```
# Description of dimension 2
res.desc$Dim.2
#> $quanti
#>             correlation  p.value
#> Pole.vault        0.807 3.21e-06
```

```
#> X1500m           0.784 9.38e-06
#> High.jump       -0.465 2.53e-02
```

2.10 Graph of individuals

Results The results, for individuals can be extracted using the function get_pca_ind() [factoextra package]. Similarly to the get_pca_var(), the function get_pca_ind() provides a list of matrices containing all the results for the individuals (coordinates, correlation between individuals and axes, squared cosine and contributions)

```
ind <- get_pca_ind(res.pca)
ind
#> Principal Component Analysis Results for individuals
#>  ===================================================
#>   Name       Description
#> 1 "$coord"   "Coordinates for the individuals"
#> 2 "$cos2"    "Cos2 for the individuals"
#> 3 "$contrib" "contributions of the individuals"
```

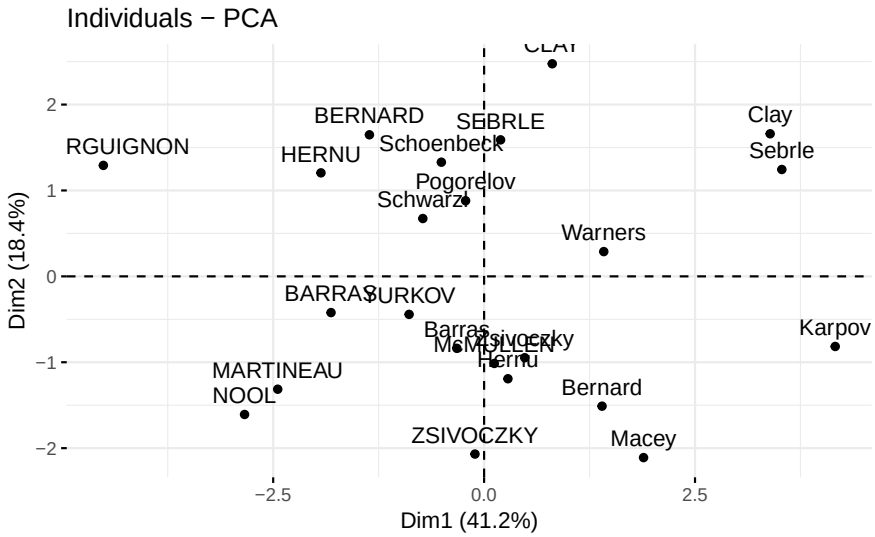
To get access to the different components, use this:

```
# Coordinates of individuals
head(ind$coord)
#>            Dim.1  Dim.2  Dim.3   Dim.4   Dim.5
#> SEBRLE     0.196  1.589  0.642  0.0839  1.1683
#> CLAY       0.808  2.475 -1.387  1.2984 -0.8250
#> BERNARD   -1.359  1.648  0.201 -1.9641  0.0842
#> YURKOV    -0.889 -0.443  2.530  0.7129  0.4078
#> ZSIVOCZKY -0.108 -2.069 -1.334 -0.1015 -0.2015
#> McMULLEN   0.121 -1.014 -0.863  1.3416  1.6215
# Quality of individuals
head(ind$cos2)
#>             Dim.1  Dim.2   Dim.3   Dim.4    Dim.5
#> SEBRLE    0.00753 0.4975 0.08133 0.00139 0.268903
#> CLAY      0.04870 0.4570 0.14363 0.12579 0.050785
#> BERNARD   0.19720 0.2900 0.00429 0.41182 0.000757
#> YURKOV    0.09611 0.0238 0.77823 0.06181 0.020228
#> ZSIVOCZKY 0.00157 0.5764 0.23975 0.00139 0.005465
#> McMULLEN  0.00218 0.1522 0.11014 0.26649 0.389262
# Contributions of individuals
head(ind$contrib)
#>            Dim.1  Dim.2  Dim.3   Dim.4   Dim.5
#> SEBRLE    0.0403  5.971  1.448  0.0373  8.4589
#> CLAY      0.6881 14.484  6.754  8.9446  4.2179
#> BERNARD   1.9474  6.423  0.141 20.4682  0.0439
#> YURKOV    0.8331  0.463 22.452  2.6966  1.0308
#> ZSIVOCZKY 0.0123 10.122  6.246  0.0547  0.2515
#> McMULLEN  0.0155  2.431  2.610  9.5506 16.2949
```

2.11 Plots: quality and contribution

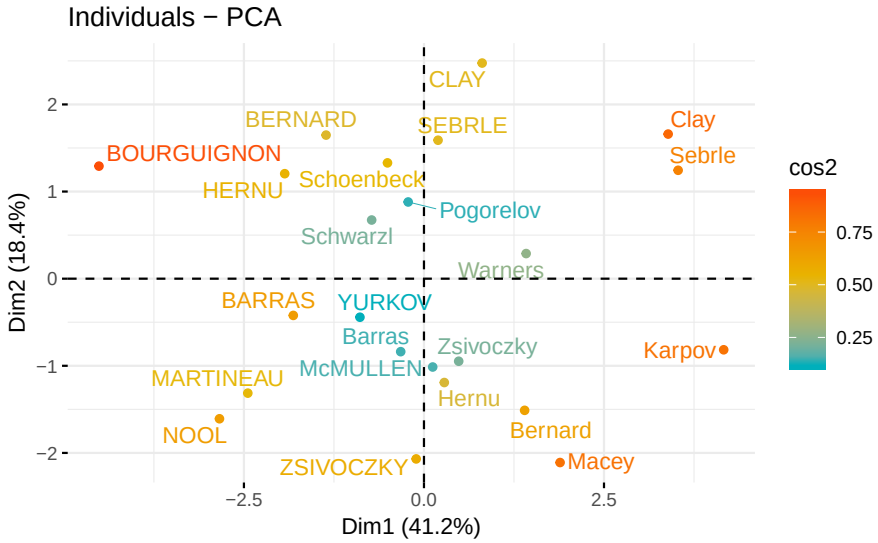
The fviz_pca_ind() is used to produce the graph of individuals. To create a simple plot, type this:

```
fviz_pca_ind(res.pca)
```



Like variables, it's also possible to color individuals by their cos2 values:

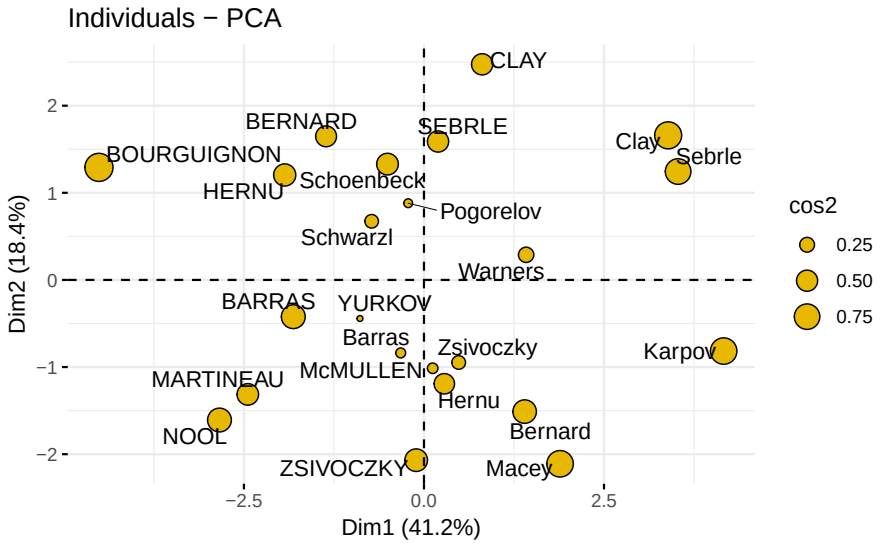
```
fviz_pca_ind(res.pca, col.ind = "cos2",
             gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"),
             repel = TRUE # Avoid text overlapping (slow if many points)
             )
```



Note that, individuals that are similar are grouped together on the plot.

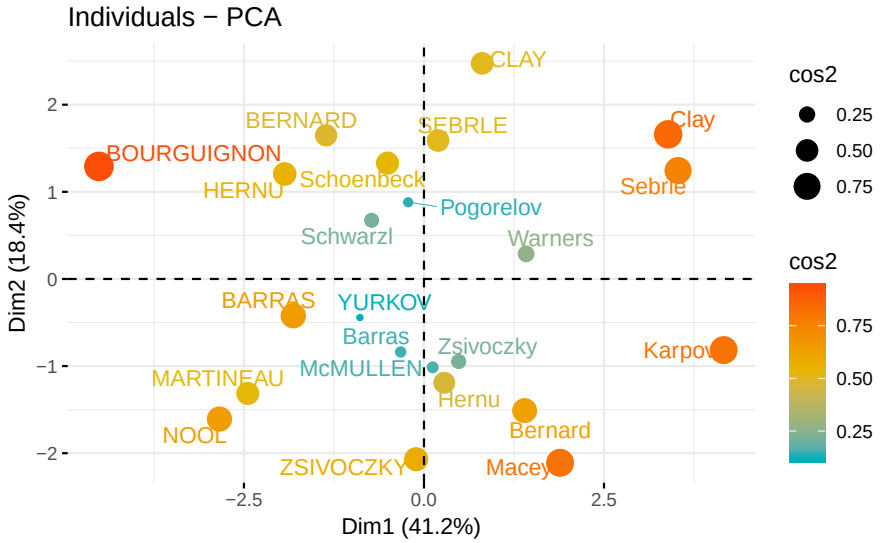
You can also change the point size according the cos2 of the corresponding individuals:

```
fviz_pca_ind(res.pca, pointsize = "cos2",
             pointshape = 21, fill = "#E7B800",
             repel = TRUE # Avoid text overlapping (slow if many points)
             )
```



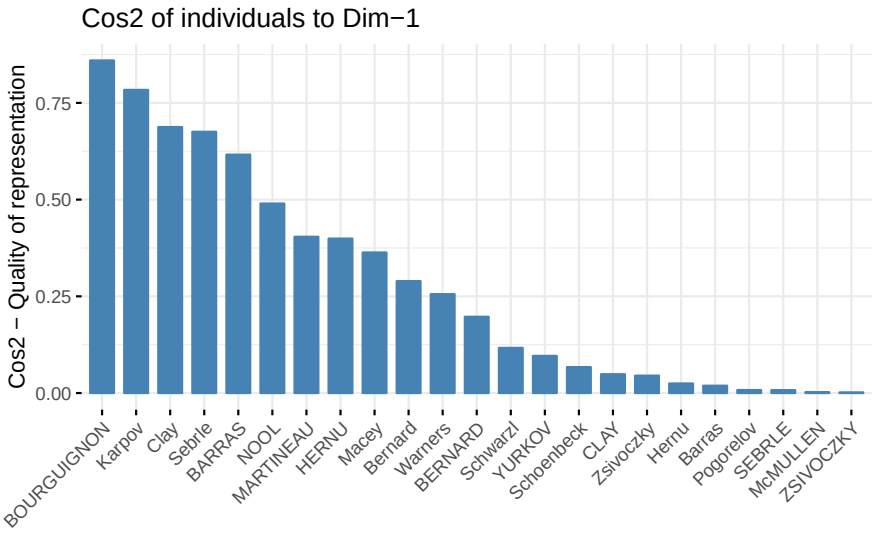
To change both point size and color by cos2, try this:

```
fviz_pca_ind(res.pca, col.ind = "cos2", pointsize = "cos2",
             gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"),
             repel = TRUE # Avoid text overlapping (slow if many points)
             )
```



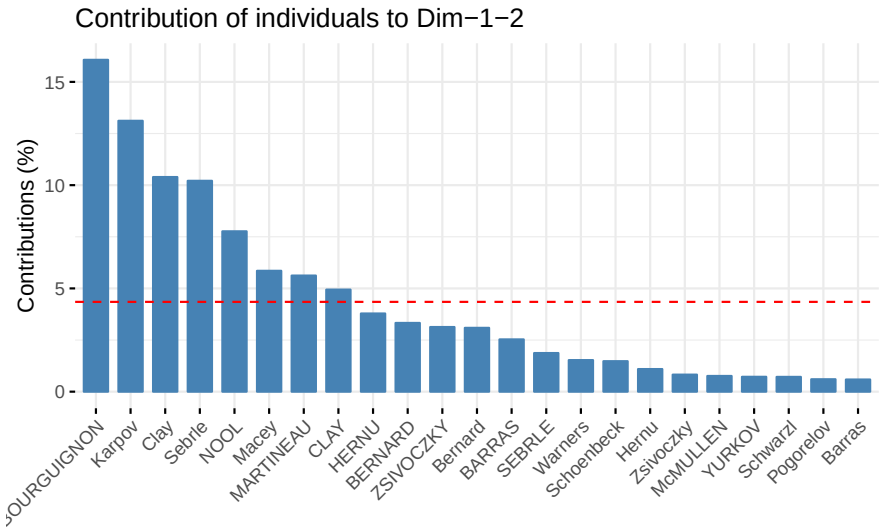
To create a bar plot of the quality of representation (cos2) of individuals on the factor map, you can use the function fviz_cos2() as previously described for variables:

```
fviz_cos2(res.pca, choice = "ind")
```



To visualize the contribution of individuals to the first two principal components, type this:

```
# Total contribution on PC1 and PC2
fviz_contrib(res.pca, choice = "ind", axes = 1:2)
```

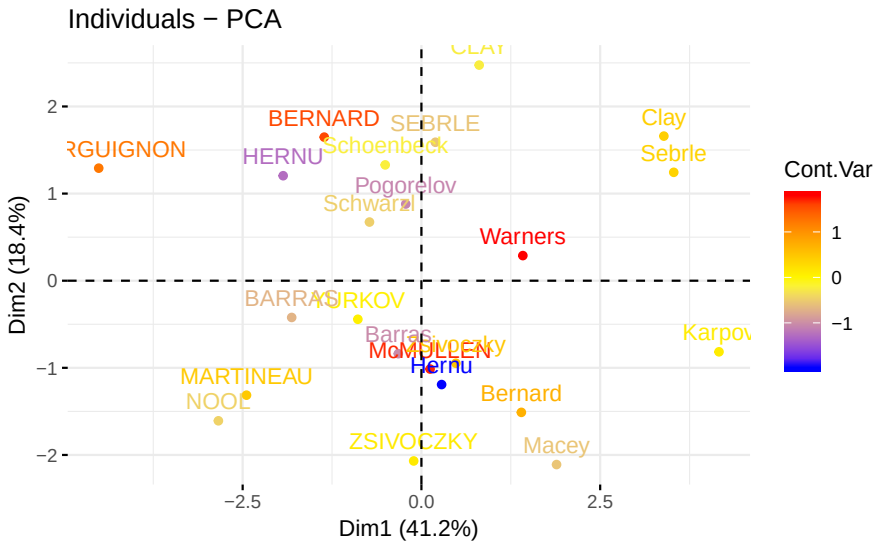


2.12 Color by a custom continuous variable

As for variables, individuals can be colored by any custom continuous variable by specifying the argument col.ind.

For example, type this:

```
# Create a random continuous variable of length 23,
# Same length as the number of active individuals in the PCA
set.seed(123)
my.cont.var <- rnorm(23)
# Color individuals by the continuous variable
fviz_pca_ind(res.pca, col.ind = my.cont.var,
             gradient.cols = c("blue", "yellow", "red"),
             legend.title = "Cont.Var")
```



2.13 Color by groups

Here, we describe how to color individuals by group. Additionally, we show how to add concentration ellipses and confidence ellipses by groups. For this, we'll use the iris data as demo data sets.

Iris data sets look like this:

```
head(iris, 3)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1          5.1         3.5          1.4         0.2  setosa
#> 2          4.9         3.0          1.4         0.2  setosa
#> 3          4.7         3.2          1.3         0.2  setosa
```

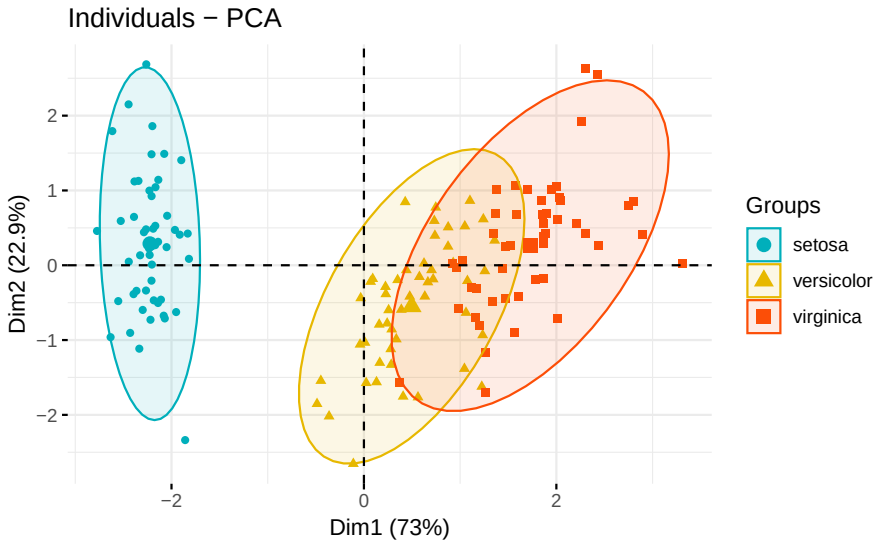
The column "Species" will be used as grouping variable. We start by computing principal component analysis as follow:

```
# The variable Species (index = 5) is removed
# before PCA analysis
iris.pca <- PCA(iris[,-5], graph = FALSE)
```

In the R code below: the argument habillage or col.ind can be used to specify the factor variable for coloring the individuals by groups.

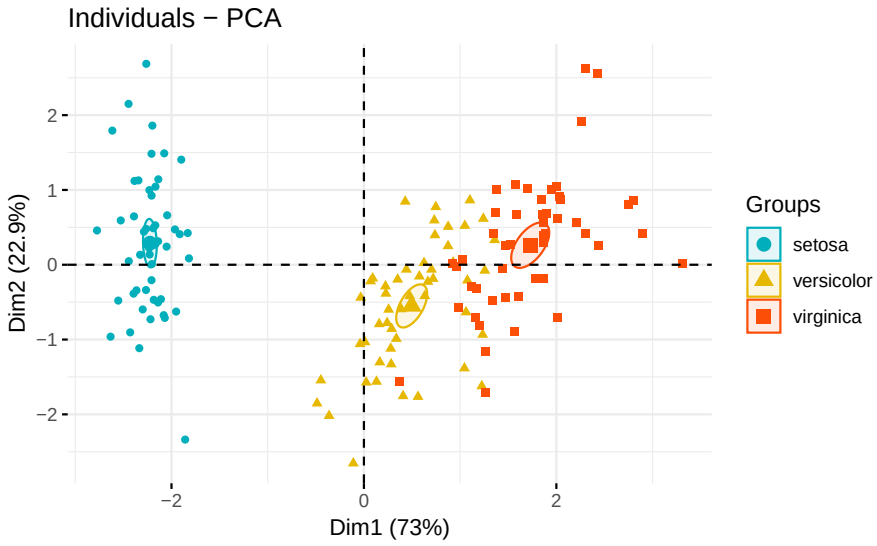
To add a concentration ellipse around each group, specify the argument addEllipses = TRUE. The argument palette can be used to change group colors.

```
fviz_pca_ind(iris.pca,
             geom.ind = "point", # show points only (nbut not "text")
             col.ind = iris$Species, # color by groups
             palette = c("#00AFBB", "#E7B800", "#FC4E07"),
             addEllipses = TRUE, # Concentration ellipses
             legend.title = "Groups"
             )
```



To remove the group mean point, specify the argument mean.point = FALSE. If you want confidence ellipses instead of concentration ellipses, use ellipse.type = "confidence".

```
# Add confidence ellipses
fviz_pca_ind(iris.pca, geom.ind = "point", col.ind = iris$Species,
             palette = c("#00AFBB", "#E7B800", "#FC4E07"),
             addEllipses = TRUE, ellipse.type = "confidence",
             legend.title = "Groups"
             )
```

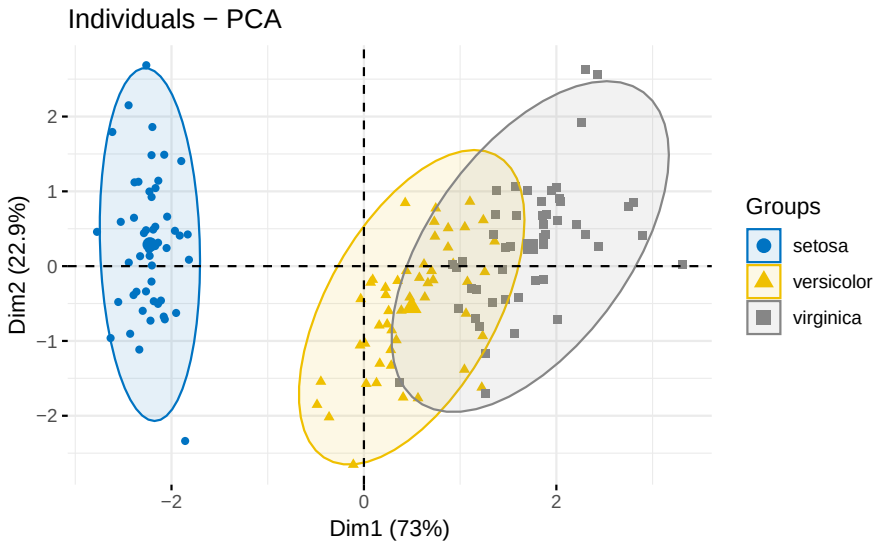


Note that, allowed values for palette include:

- "grey" for grey color palettes;
- brewer palettes e.g. "RdBu", "Blues", ...; To view all, type this in R: RColorBrewer::display.brewer.all().
- custom color palette e.g. c("blue", "red"); and scientific journal palettes from ggsci R package, e.g.: "npg", "aaas", * "lancet", "jco", "ucscgb", "uchicago", "simpsons" and "rickandmorty". For example, to use the jco (journal of clinical oncology) color palette, type this:

```
fviz_pca_ind(iris.pca,
             label = "none", # hide individual labels
             habillage = iris$Species, # color by groups
```

```
             addEllipses = TRUE, # Concentration ellipses
             palette = "jco"
             )
```



2.14 Graph customization

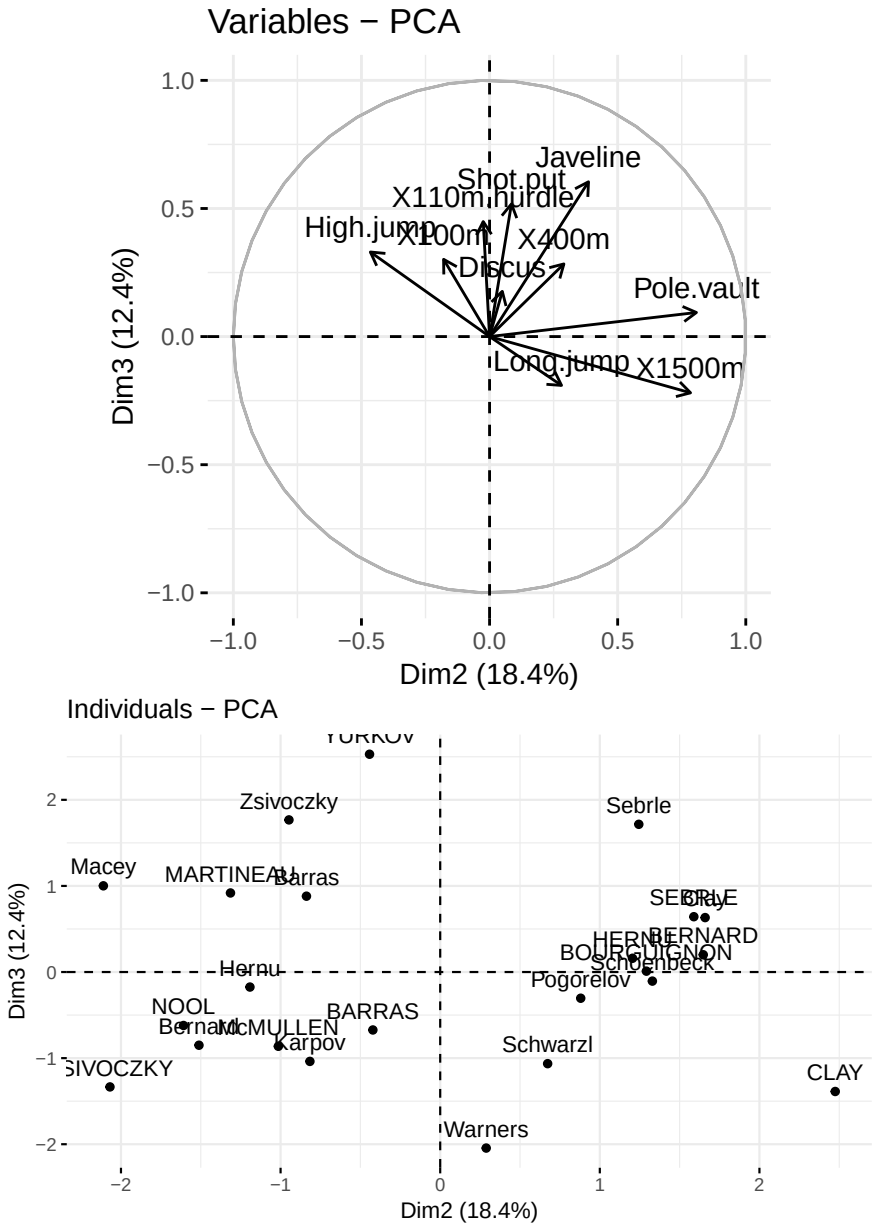
Note that, fviz_pca_ind() and fviz_pca_var() and related functions are wrapper around the core function fviz() [in factoextra]. fviz() is a wrapper around the function ggscatter() [in ggpubr]. Therefore, further arguments, to be passed to the function fviz() and ggscatter(), can be specified in fviz_pca_ind() and fviz_pca_var().

Here, we present some of these additional arguments to customize the PCA graph of variables and individuals.

2.14.1 Dimensions

By default, variables/individuals are represented on dimensions 1 and 2. If you want to visualize them on dimensions 2 and 3, for example, you should specify the argument axes = c(2, 3).

```
# Variables on dimensions 2 and 3
fviz_pca_var(res.pca, axes = c(2, 3))
# Individuals on dimensions 2 and 3
fviz_pca_ind(res.pca, axes = c(2, 3))
```



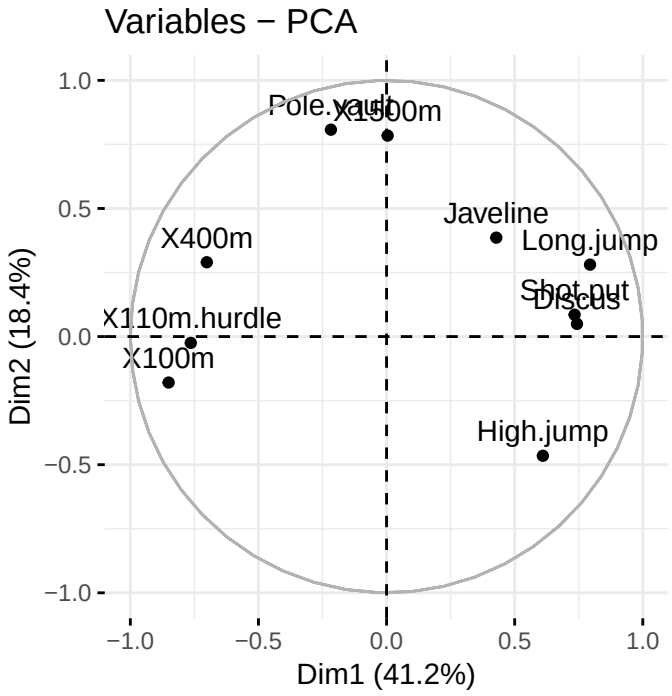
Plot elements: point, text, arrow The argument geom (for geometry) and derivatives are used to specify the geometry elements or graphical elements to be used for plotting.

1. geom.var: a text specifying the geometry to be used for plotting variables. Allowed values are the combination of c("point", "arrow", "text").

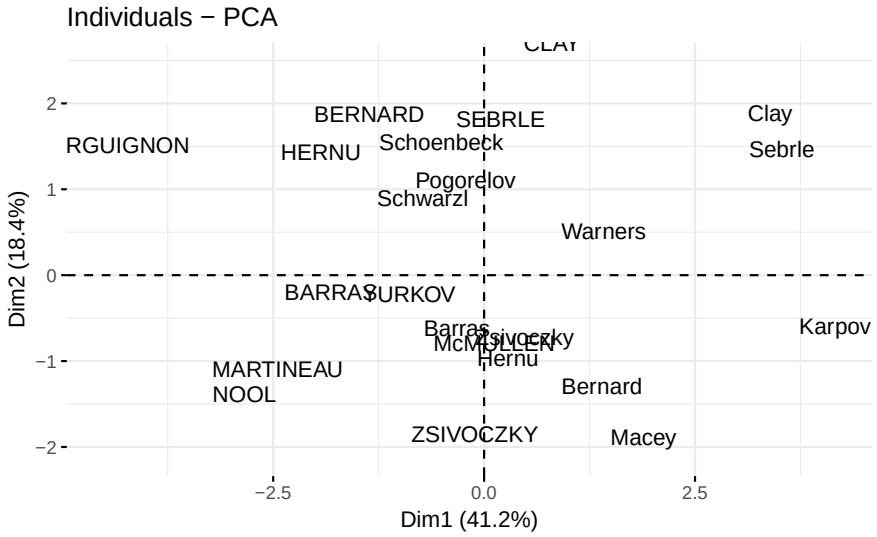
- Use geom.var = "point", to show only points;
- Use geom.var = "text" to show only text labels;
- Use geom.var = c("point", "text") to show both points and text labels
- Use geom.var = c("arrow", "text") to show arrows and labels (default).

For example, type this:

```
# Show variable points and text labels
fviz_pca_var(res.pca, geom.var = c("point", "text"))
```



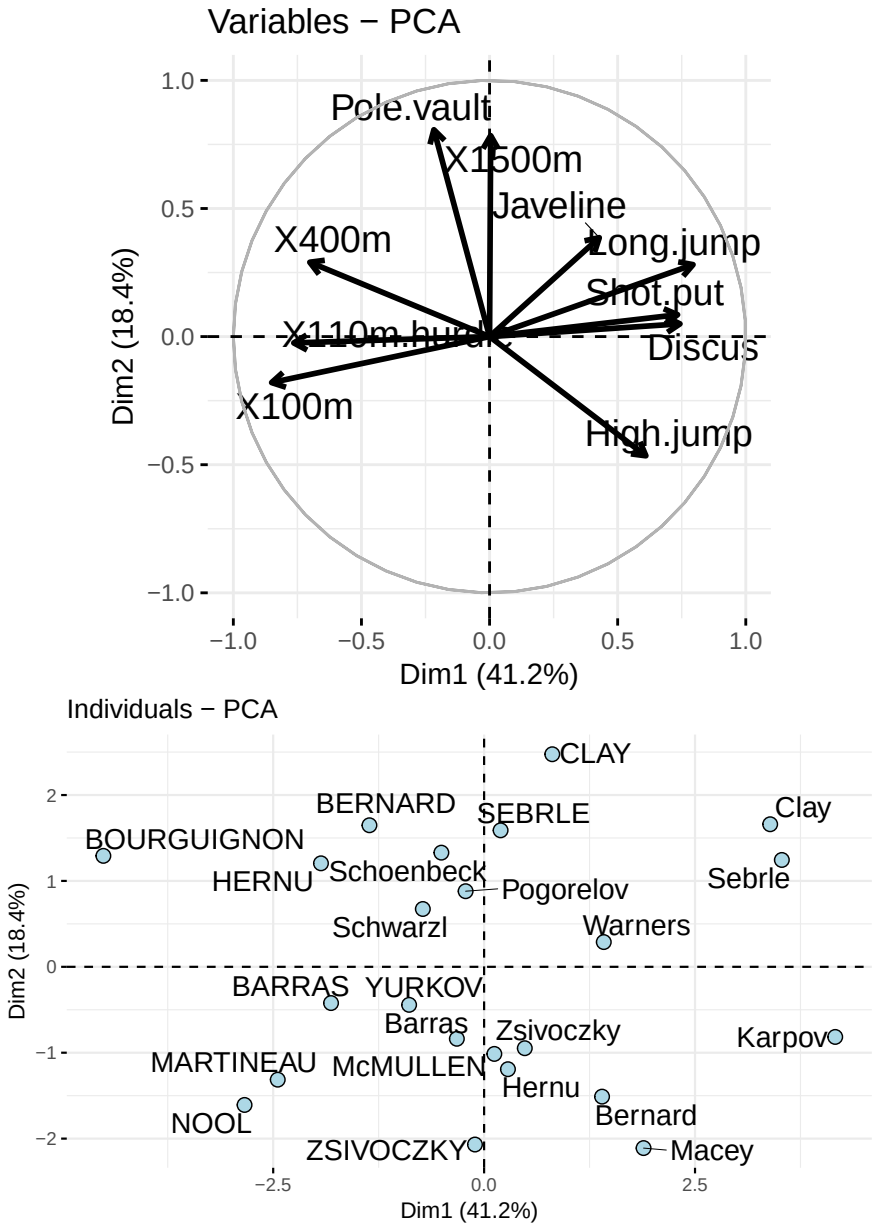

```
# Show individuals text labels only
fviz_pca_ind(res.pca, geom.ind =  "text")
```



2.15 Size and shape of plot elements

```
# Change the size of arrows an labels
fviz_pca_var(res.pca, arrowsize = 1, labelsize = 5,
             repel = TRUE)
# Change points size, shape and fill color
# Change labelsize
fviz_pca_ind(res.pca,
             pointsize = 3, pointshape = 21, fill = "lightblue",
```

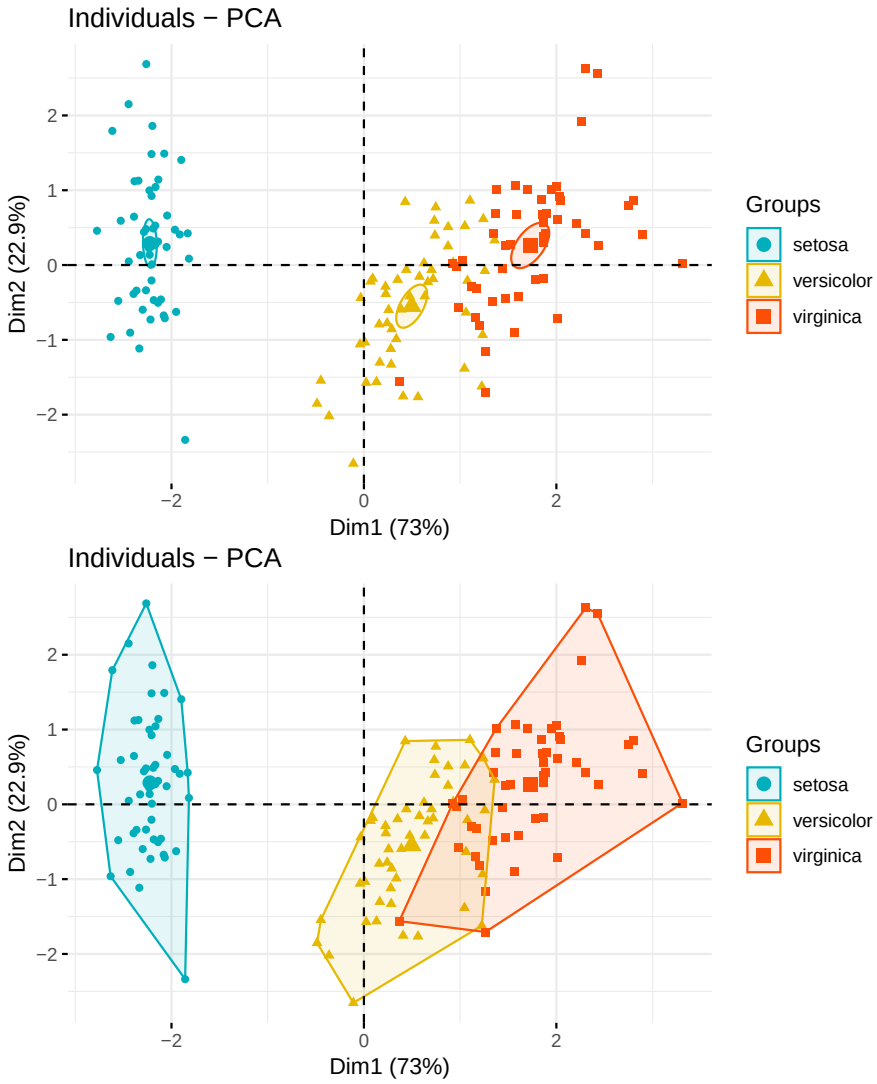
```
            labelsize = 5, repel = TRUE)
```



2.16 Ellipses

```
# Add confidence ellipses
fviz_pca_ind(iris.pca, geom.ind = "point",
             col.ind = iris$Species, # color by groups
             palette = c("#00AFBB", "#E7B800", "#FC4E07"),
             addEllipses = TRUE, ellipse.type = "confidence",
             legend.title = "Groups"
             )
# Convex hull
```

```
fviz_pca_ind(iris.pca, geom.ind = "point",
             col.ind = iris$Species, # color by groups
             palette = c("#00AFBB", "#E7B800", "#FC4E07"),
             addEllipses = TRUE, ellipse.type = "convex",
             legend.title = "Groups"
             )
```



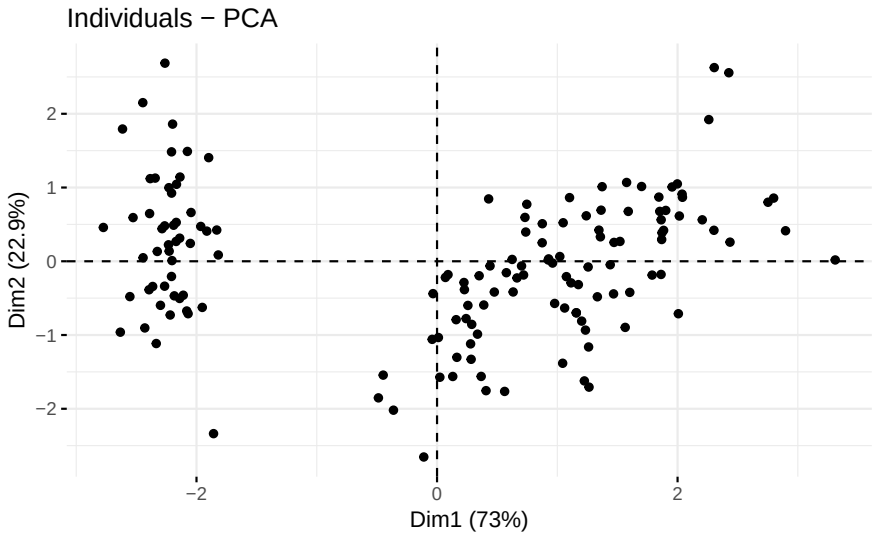
2.17 Group mean points

When coloring individuals by groups (section **?**(color-ind-by-groups)), the mean points of groups (barycenters) are also displayed by default.

To remove the mean points, use the argument mean.point = FALSE.

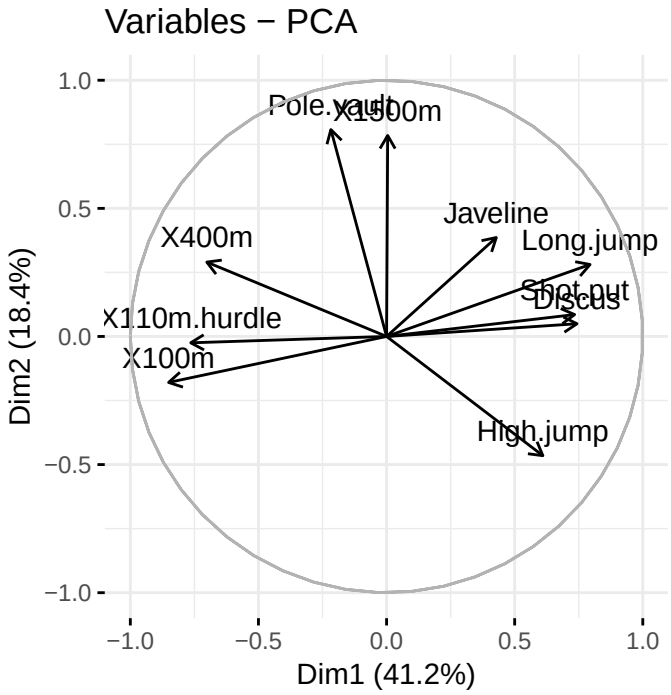
```
fviz_pca_ind(iris.pca,
             geom.ind = "point", # show points only (but not "text")
             group.ind = iris$Species, # color by groups
             legend.title = "Groups",
```

```
             mean.point = FALSE)
```



2.18 Axis lines

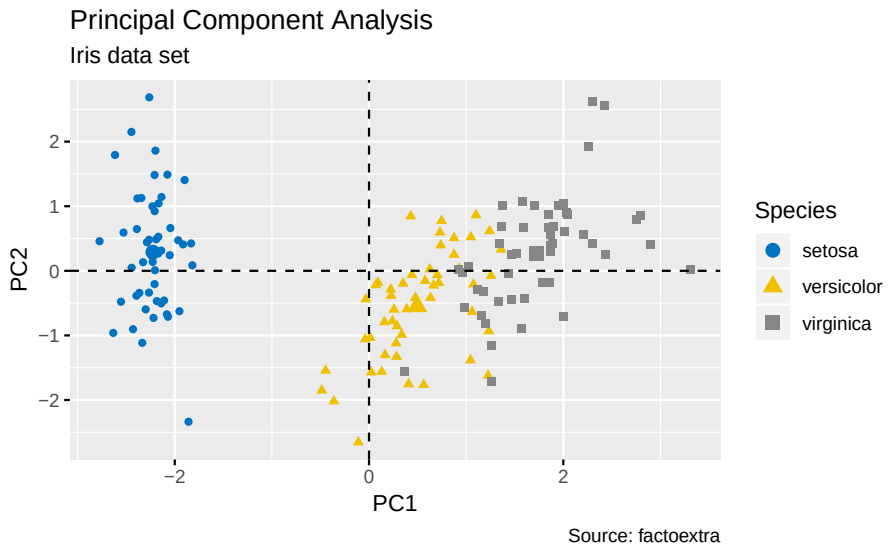
```
fviz_pca_var(res.pca, axes.linetype = "blank")
```



2.19 Graphical parameters

To change easily the graphical of any ggplots, you can use the function ggpar() [ggpubr package]

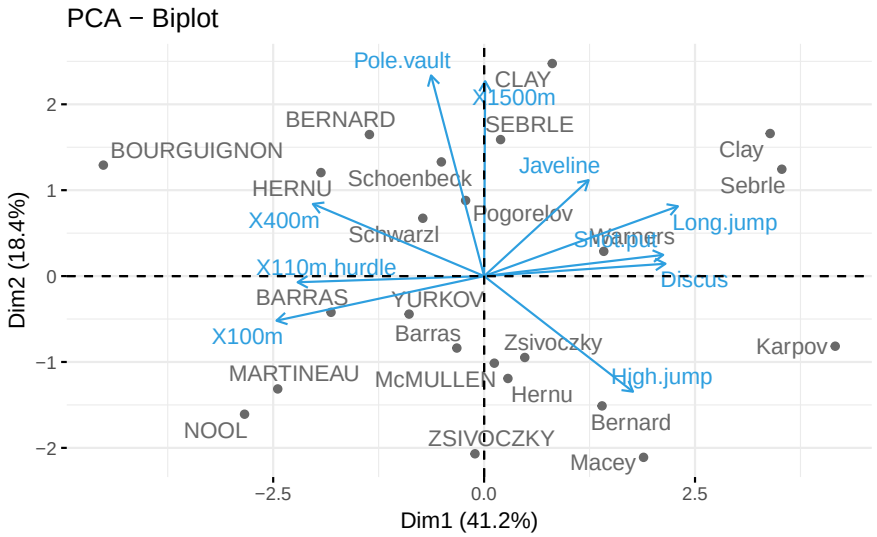
```
ind.p <- fviz_pca_ind(iris.pca, geom = "point", col.ind = iris$Species)
ggpubr::ggpar(ind.p,
              title = "Principal Component Analysis",
              subtitle = "Iris data set",
              caption = "Source: factoextra",
              xlab = "PC1", ylab = "PC2",
              legend.title = "Species", legend.position = "top",
              ggtheme = theme_gray(), palette = "jco"
              )
```



2.20 Biplot

To make a simple biplot of individuals and variables, type this:

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

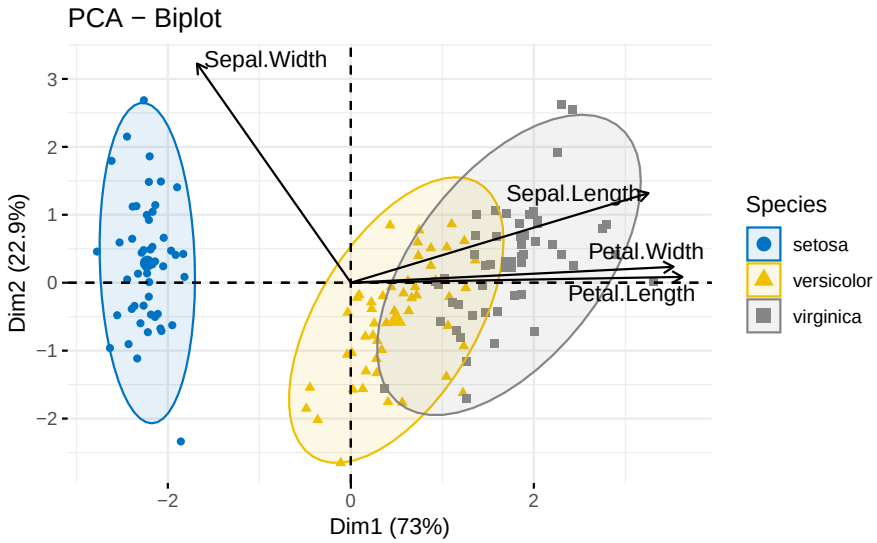


Note that, the biplot might be only useful when there is a low number of variables and individuals in the data set; otherwise the final plot would be unreadable.

Note also that, the coordinate of individuals and variables are not constructed on the same space. Therefore, in the biplot, you should mainly focus on the direction of variables but not on their absolute positions on the plot.

Roughly speaking a biplot can be interpreted as follow: * an individual that is on the same side of a given variable has a high value for this variable; * an individual that is on the opposite side of a given variable has a low value for this variable.

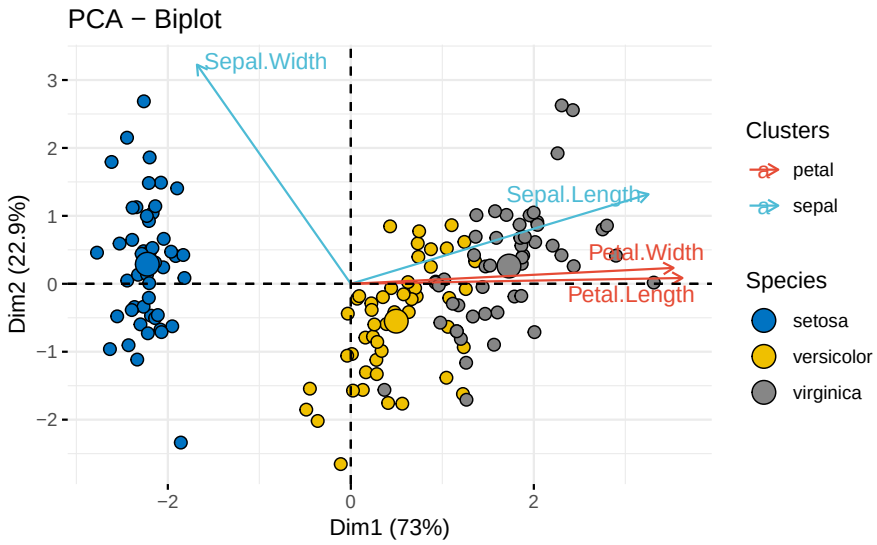
```
fviz_pca_biplot(iris.pca,
                col.ind = iris$Species, palette = "jco",
                addEllipses = TRUE, label = "var",
                col.var = "black", repel = TRUE,
                legend.title = "Species")
```



In the following example, we want to color both individuals and variables by groups. The trick is to use pointshape = 21 for individual points. This particular point shape can be filled by a color using the argument fill.ind. The border line color of individual points is set to "black" using col.ind. To color variable by groups, the argument col.var will be used.

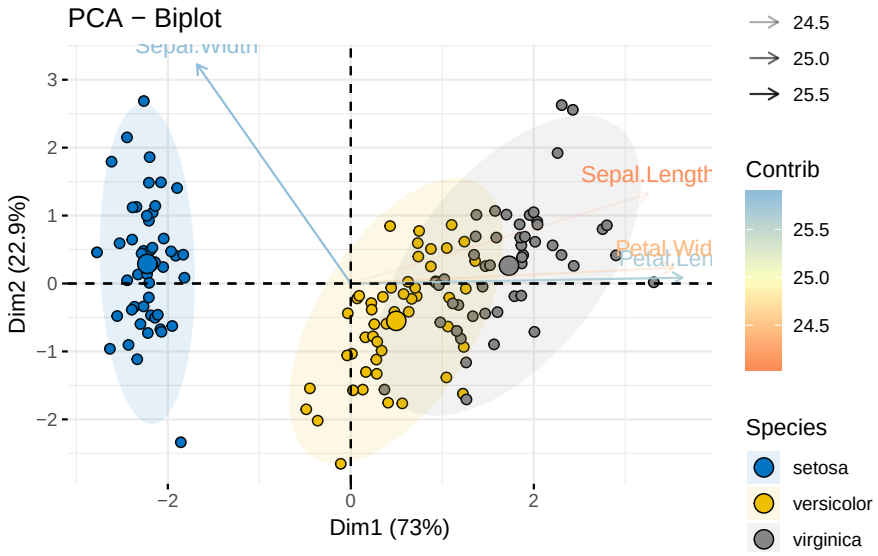
To customize individuals and variable colors, we use the helper functions fill_palette() and color_palette() [in ggpubr package].

```
fviz_pca_biplot(iris.pca,
                # Fill individuals by groups
                geom.ind = "point",
                pointshape = 21,
                pointsize = 2.5,
                fill.ind = iris$Species,
                col.ind = "black",
                # Color variable by groups
                col.var = factor(c("sepal", "sepal", "petal", "petal")),

                legend.title = list(fill = "Species", color = "Clusters"),
                repel = TRUE        # Avoid label overplotting
             )+
  ggpubr::fill_palette("jco")+      # Indiviual fill color
  ggpubr::color_palette("npg")      # Variable colors
```



Another complex example is to color individuals by groups (discrete color) and variables by their contributions to the principal components (gradient colors). Additionally, we'll change the transparency of variables by their contributions using the argument alpha.var.

```
fviz_pca_biplot(iris.pca,
                # Individuals
                geom.ind = "point",
                fill.ind = iris$Species, col.ind = "black",
                pointshape = 21, pointsize = 2,
                palette = "jco",
                addEllipses = TRUE,
                # Variables
                alpha.var ="contrib", col.var = "contrib",
                gradient.cols = "RdYlBu",

                legend.title = list(fill = "Species", color = "Contrib",
                                    alpha = "Contrib")
                )
```



2.21 Supplementary elements

Definition and types As described above (section ?(pca-data-format)), the decathlon2 data sets contain supplementary continuous variables (quanti.sup, columns 11:12), supplementary qualitative variables (quali.sup, column 13) and supplementary individuals (ind.sup, rows 24:27).

Supplementary variables and individuals are not used for the determination of the principal components. Their coordinates are predicted using only the information provided by the performed principal component analysis on active variables/individuals.

Specification in PCA To specify supplementary individuals and variables, the function PCA() can be used as follow:

```
res.pca <- PCA(decathlon2, ind.sup = 24:27,
               quanti.sup = 11:12, quali.sup = 13, graph=FALSE)
```

2.22 Quantitative variables

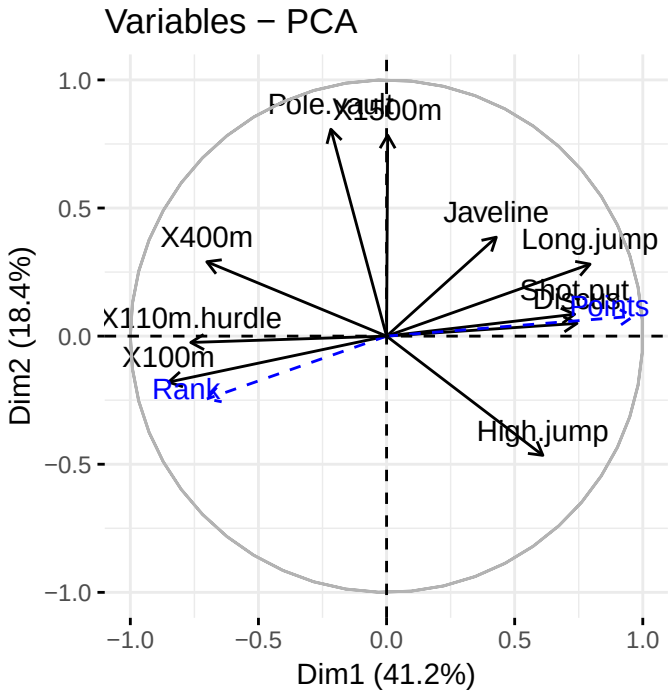
Predicted results (coordinates, correlation and cos2) for the supplementary quantitative variables:

```
res.pca$quanti.sup
#> $coord
#>          Dim.1   Dim.2  Dim.3   Dim.4   Dim.5
#> Rank   -0.701 -0.2452 -0.183  0.0558 -0.0738
#> Points  0.964  0.0777  0.158 -0.1662 -0.0311
#>
#> $cor
#>          Dim.1   Dim.2  Dim.3   Dim.4   Dim.5
#> Rank   -0.701 -0.2452 -0.183  0.0558 -0.0738
#> Points  0.964  0.0777  0.158 -0.1662 -0.0311
#>
#> $cos2
#>         Dim.1   Dim.2  Dim.3   Dim.4   Dim.5
```

```
#> Rank   0.492 0.06012 0.0336 0.00311 0.00545
#> Points 0.929 0.00603 0.0250 0.02763 0.00097
```

Visualize all variables (active and supplementary ones):

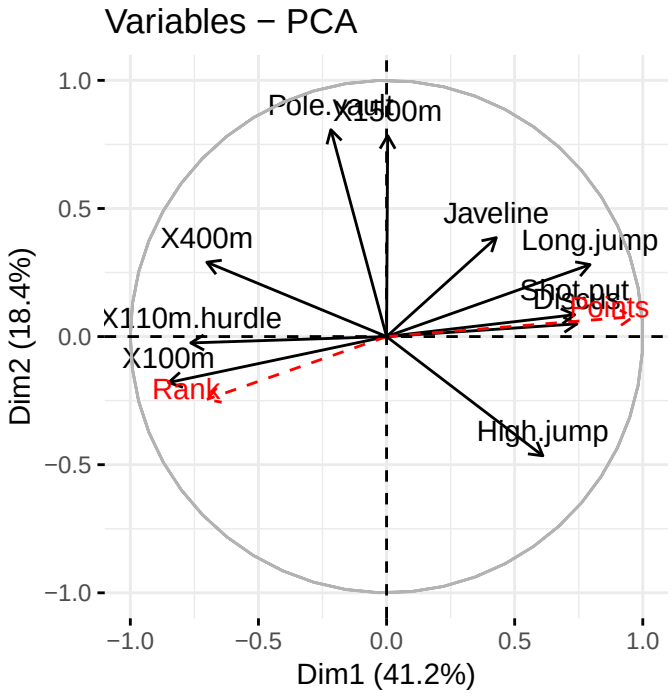
```
fviz_pca_var(res.pca)
```

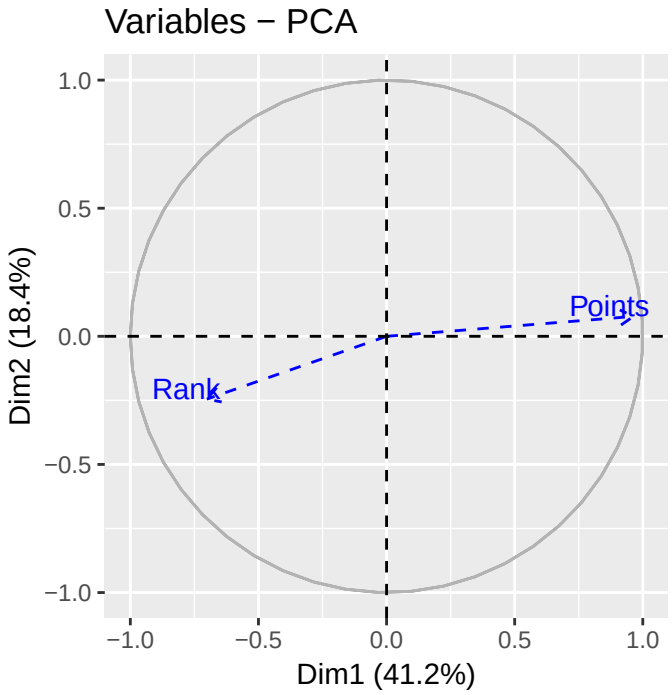


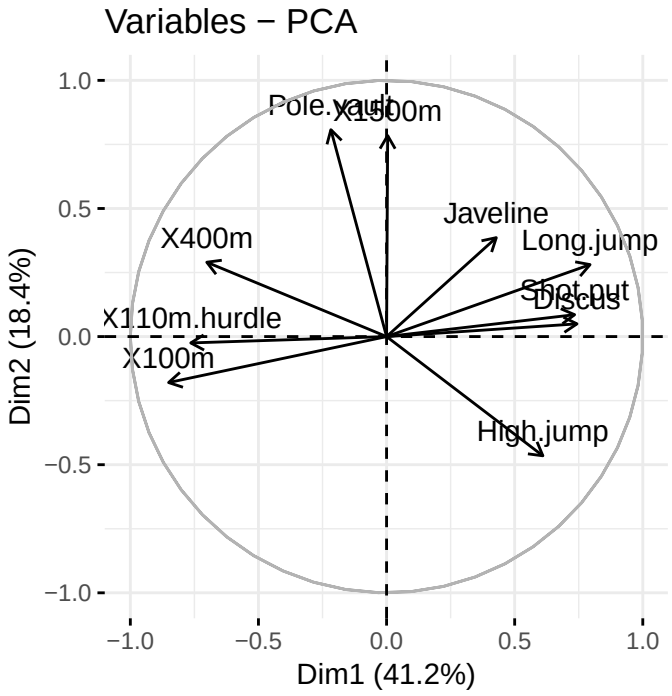
Note that, by default, supplementary quantitative variables are shown in blue color and dashed lines.

Further arguments to customize the plot:

```
# Change color of variables
fviz_pca_var(res.pca,
             col.var = "black",     # Active variables
             col.quanti.sup = "red" # Suppl. quantitative variables
             )
# Hide active variables on the plot,
# show only supplementary variables
fviz_pca_var(res.pca, invisible = "var")
# Hide supplementary variables
fviz_pca_var(res.pca, invisible = "quanti.sup")
```

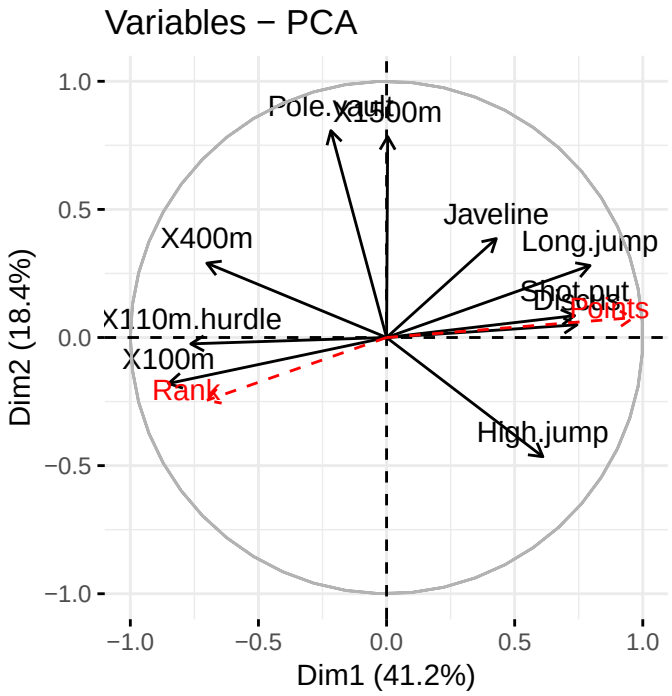

Variables − PCA
Pole.vault
X1500m
Javeline
Long.jump
X400m
Shot.put
Discus
Points
X110m.hurdle
X100m
Rank
High.jump
Dim2 (18.4%)
Dim1 (41.2%)
1.0
0.5
0.0
−0.5
−1.0


Variables − PCA
Points
Rank
Dim2 (18.4%)
Dim1 (41.2%)
1.0
0.5
0.0
−0.5
−1.0



Using the fviz_pca_var(), the quantitative supplementary variables are displayed automatically on the correlation circle plot. Note that, you can add the quanti.sup variables manually, using the fviz_add() function, for further customization. An example is shown below.

```
# Plot of active variables
p <- fviz_pca_var(res.pca, invisible = "quanti.sup")
# Add supplementary active variables
fviz_add(p, res.pca$quanti.sup$coord,
         geom = c("arrow", "text"),
         color = "red")
```



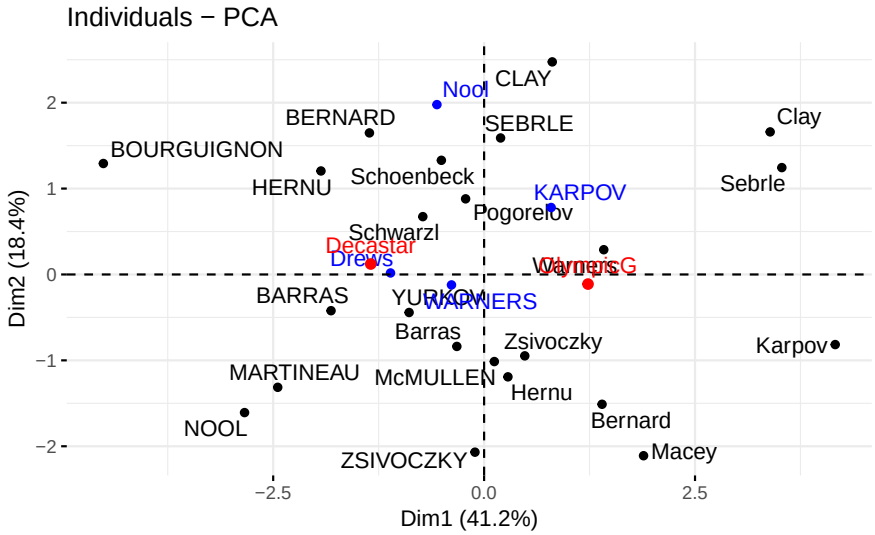
2.23 Individuals

Predicted results for the supplementary individuals (ind.sup):

```
res.pca$ind.sup
#> $coord
#>          Dim.1   Dim.2  Dim.3  Dim.4   Dim.5
#> KARPOV   0.795  0.7795 -1.633  1.724 -0.7507
#> WARNERS -0.386 -0.1216 -1.739 -0.706 -0.0323
#> Nool    -0.559  1.9775 -0.483 -2.278 -0.2546
#> Drews   -1.109  0.0174 -3.049 -1.534 -0.3264
#>
#> $cos2
#>          Dim.1    Dim.2  Dim.3  Dim.4    Dim.5
#> KARPOV  0.0510 4.91e-02 0.2155 0.2403 0.045549
#> WARNERS 0.0242 2.40e-03 0.4904 0.0809 0.000169
#> Nool    0.0290 3.62e-01 0.0216 0.4811 0.006008
#> Drews   0.0921 2.27e-05 0.6956 0.1762 0.007974
#>
#> $dist
#>  KARPOV WARNERS    Nool   Drews
#>    3.52    2.48    3.28    3.66
```

Visualize all individuals (active and supplementary ones). On the graph, you can add also the supplementary qualitative variables (quali.sup), which coordinates is accessible using res.pca*quali.supp*coord.

```
p <- fviz_pca_ind(res.pca, col.ind.sup = "blue", repel = TRUE)
p <- fviz_add(p, res.pca$quali.sup$coord, color = "red")
p
```



Supplementary individuals are shown in blue. The levels of the supplementary qualitative variable are shown in red color.

2.24 Qualitative variables

In the previous section, we showed that you can add the supplementary qualitative variables on individuals plot using fviz_add().

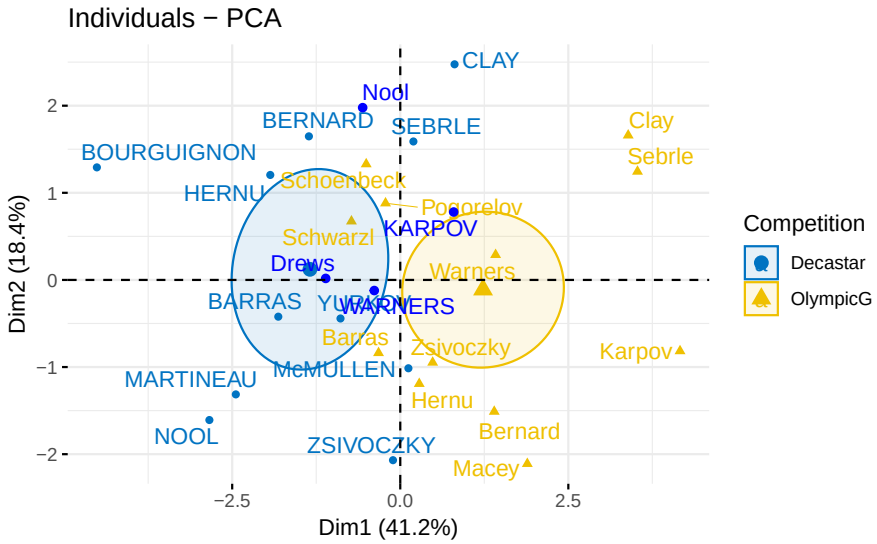
Note that, the supplementary qualitative variables can be also used for coloring individuals by groups. This can help to interpret the data. The data sets decathlon2 contain a supplementary qualitative variable at columns 13 corresponding to the type of competitions.

The results concerning the supplementary qualitative variable are:

```
res.pca$quali
#> $coord
#>          Dim.1  Dim.2   Dim.3  Dim.4  Dim.5
#> Decastar -1.34  0.122 -0.0379  0.181  0.134
#> OlympicG  1.23 -0.112  0.0347 -0.166 -0.123
#>
#> $cos2
#>          Dim.1   Dim.2   Dim.3  Dim.4   Dim.5
#> Decastar 0.905 0.00744 0.00072 0.0164 0.00905
#> OlympicG 0.905 0.00744 0.00072 0.0164 0.00905
#>
#> $v.test
#>          Dim.1  Dim.2  Dim.3  Dim.4 Dim.5
#> Decastar -2.97  0.403 -0.153  0.897  0.72
#> OlympicG  2.97 -0.403  0.153 -0.897 -0.72
#>
#> $dist
#> Decastar OlympicG
#>     1.41     1.29
#>
#> $eta2
#>             Dim.1  Dim.2   Dim.3  Dim.4  Dim.5
#> Competition 0.401 0.0074 0.00106 0.0366 0.0236
```

To color individuals by a supplementary qualitative variable, the argument habillage is used to specify the index of the supplementary qualitative variable. Historically, this argument name comes from the FactoMineR package. It's a french word meaning "dressing" in english. To keep consistency between FactoMineR and factoextra, we decided to keep the same argument name

```
fviz_pca_ind(res.pca, habillage = 13,
             addEllipses =TRUE, ellipse.type = "confidence",
             palette = "jco", repel = TRUE)
```

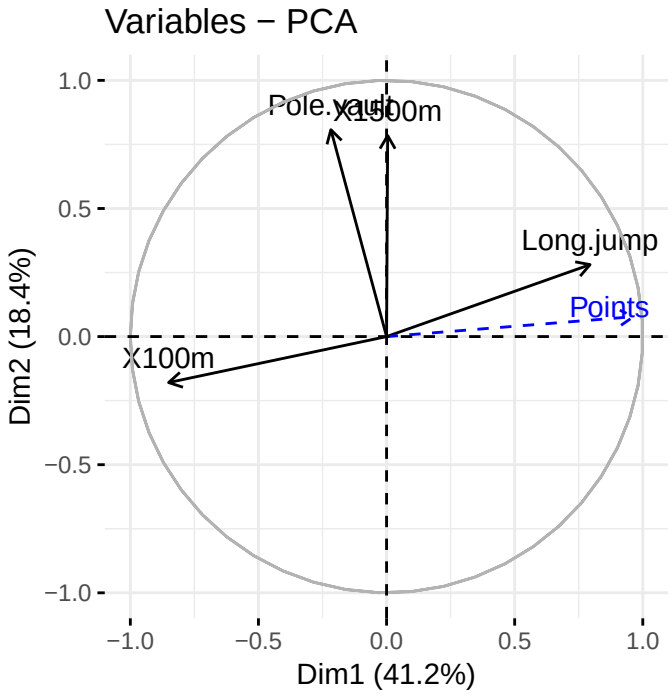


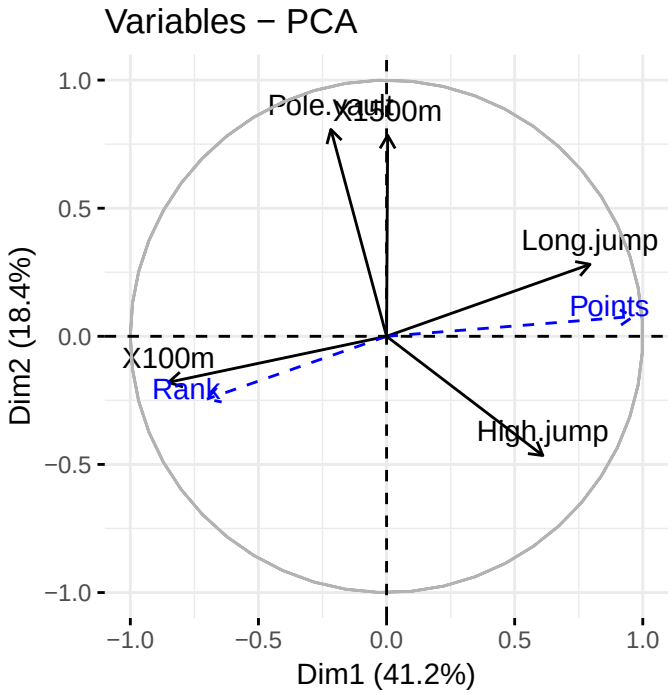
Recall that, to remove the mean points of groups, specify the argument mean.point = FALSE.

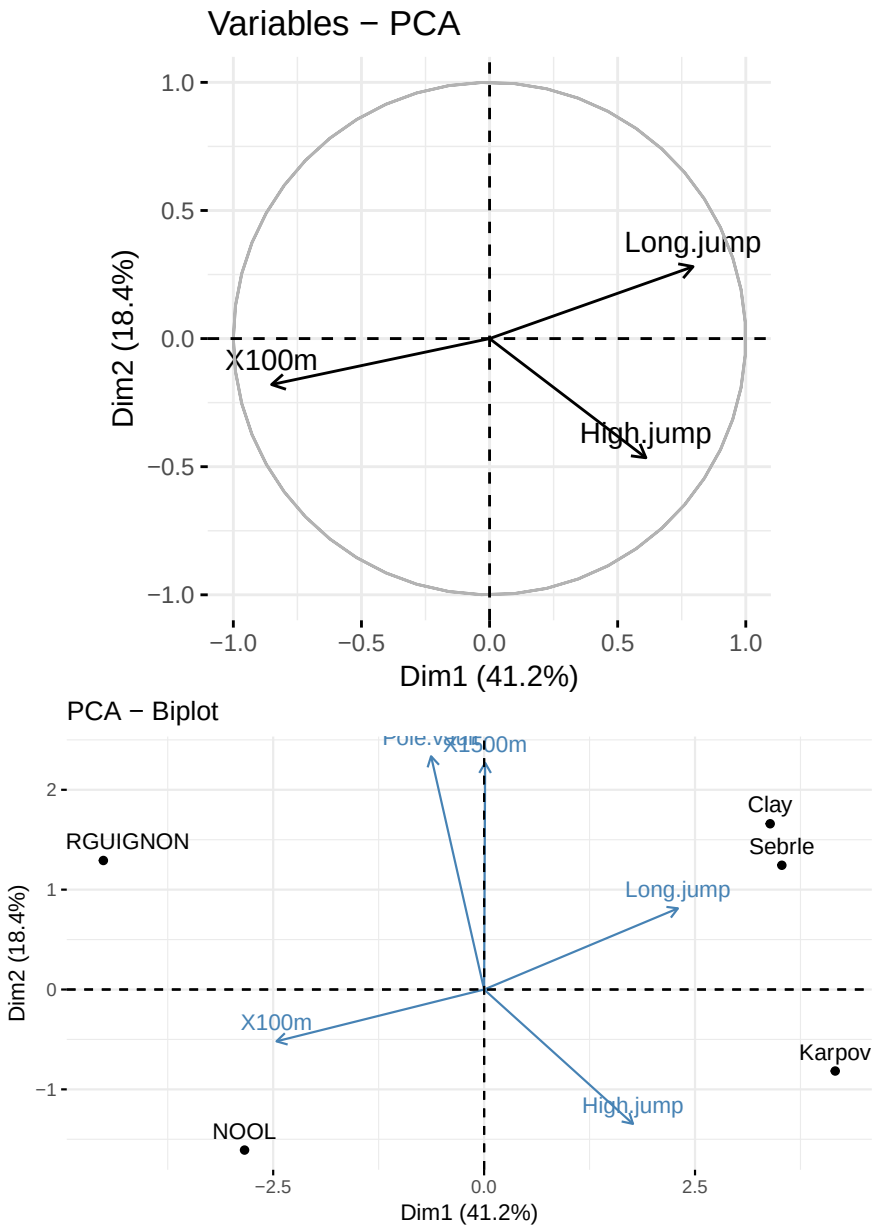
2.25 Filtering results

If you have many individuals/variable, it's possible to visualize only some of them using the arguments select.ind and select.var.

```
# Visualize variable with cos2 >= 0.6
fviz_pca_var(res.pca, select.var = list(cos2 = 0.6))
# Top 5 active variables with the highest cos2
fviz_pca_var(res.pca, select.var= list(cos2 = 5))
# Select by names
name <- list(name = c("Long.jump", "High.jump", "X100m"))
fviz_pca_var(res.pca, select.var = name)
# top 5 contributing individuals and variable
fviz_pca_biplot(res.pca, select.ind = list(contrib = 5),
               select.var = list(contrib = 5),
               ggtheme = theme_minimal())
```


Variables − PCA
Pole.vault
X1500m
Long.jump
Points
X100m
Dim2 (18.4%)
Dim1 (41.2%)
1.0
0.5
0.0
−0.5
−1.0


Variables − PCA
Pole.vault
X1500m
Long.jump
Points
X100m
Rank
High.jump
Dim2 (18.4%)
Dim1 (41.2%)
1.0
0.5
0.0
−0.5
−1.0



When the selection is done according to the contribution values, supplementary individuals/variables are not shown because they don't contribute to the construction of the axes.

2.26 Exporting results

Export plots to PDF/PNG files The factoextra package produces a ggplot2-based graphs. To save any ggplots, the standard R code is as follow:

```
# Print the plot to a pdf file
pdf("myplot.pdf")
print(myplot)
dev.off()
```

In the following examples, we'll show you how to save the different graphs into pdf or png files.

The first step is to create the plots you want as an R object:

```
# Scree plot
scree.plot <- fviz_eig(res.pca)
# Plot of individuals
ind.plot <- fviz_pca_ind(res.pca)
# Plot of variables
var.plot <- fviz_pca_var(res.pca)
```

```
pdf(file.path(data_out_dir, "PCA.pdf"))   # Create a new pdf device
print(scree.plot)
print(ind.plot)
print(var.plot)
dev.off() # Close the pdf device
#> pdf
#>   2
```

> Note that, using the above R code will create the PDF file into your current working directory. To see the path of your current working directory, type getwd() in the R console.

To print each plot to specific png file, the R code looks like this:

```
# Print scree plot to a png file
png(file.path(data_out_dir, "pca-scree-plot.png"))
print(scree.plot)
dev.off()
#> pdf
#>   2
# Print individuals plot to a png file
png(file.path(data_out_dir, "pca-variables.png"))
print(var.plot)
dev.off()
#> pdf
#>   2
# Print variables plot to a png file
png(file.path(data_out_dir, "pca-individuals.png"))
print(ind.plot)
dev.off()
#> pdf
#>   2
```

Another alternative, to export ggplots, is to use the function ggexport() [in ggpubr package]. We like ggexport(), because it's very simple. With one line R code, it allows us to export individual plots to a file (pdf, eps or png) (one plot per page). It can also arrange the plots (2 plot per page, for example) before exporting them. The examples below demonstrates how to export ggplots using ggexport().

Export individual plots to a pdf file (one plot per page):

```
library(ggpubr)
#> Loading required package: magrittr
ggexport(plotlist = list(scree.plot, ind.plot, var.plot),
         filename = file.path(data_out_dir, "PCA.pdf"))
#> file saved to /home/datascience/repos/machine-learning-rsuite/export/PCA.pdf
```

Arrange and export. Specify nrow and ncol to display multiple plots on the same page:

```
ggexport(plotlist = list(scree.plot, ind.plot, var.plot),
         nrow = 2, ncol = 2,
```

```
        filename = file.path(data_out_dir, "PCA.pdf"))
#> file saved to /home/datascience/repos/machine-learning-rsuite/export/PCA.pdf
```

Export plots to png files. If you specify a list of plots, then multiple png files will be automatically created to hold each plot.

```
ggexport(plotlist = list(scree.plot, ind.plot, var.plot),
        filename = file.path(data_out_dir, "PCA.png"))
#> [1] "/home/datascience/repos/machine-learning-rsuite/export/PCA%03d.png"
#> file saved to /home/datascience/repos/machine-learning-rsuite/export/PCA%03d.png
```

2.27 Export results to txt/csv files

All the outputs of the PCA (individuals/variables coordinates, contributions, etc) can be exported at once, into a TXT/CSV file, using the function write.infile() [in FactoMineR] package:

```
# Export into a TXT file
write.infile(res.pca, file.path(data_out_dir, "pca.txt"), sep = "\t")
# Export into a CSV file
write.infile(res.pca, file.path(data_out_dir, "pca.csv"), sep = ";")
```

2.28 Summary

In conclusion, we described how to perform and interpret principal component analysis (PCA). We computed PCA using the PCA() function [FactoMineR]. Next, we used the factoextra R package to produce ggplot2-based visualization of the PCA results.

There are other functions [packages] to compute PCA in R:

1. Using prcomp() [stats]

```
res.pca <- prcomp(iris[, -5], scale. = TRUE)
```

```
res.pca <- princomp(iris[, -5], cor = TRUE)
```

3. Using dudi.pca() [ade4]

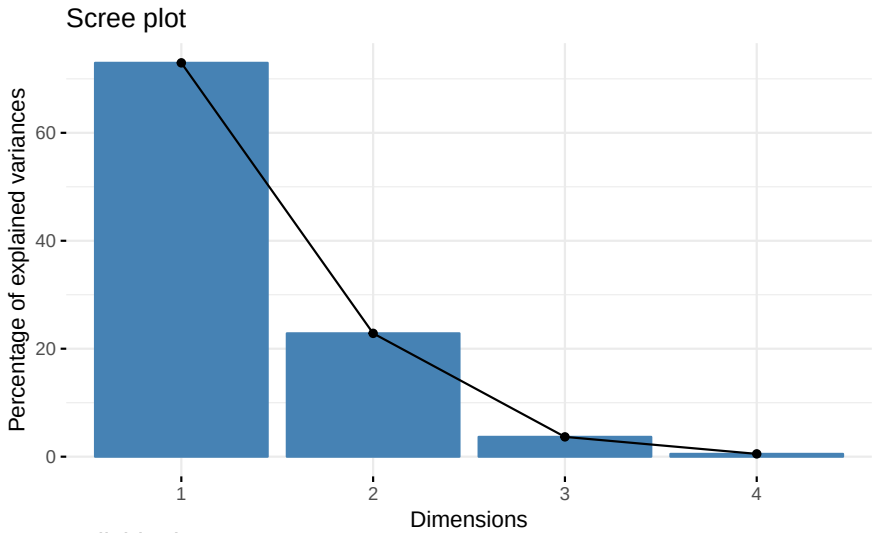
```
library(ade4)
#>
#> Attaching package: 'ade4'
#> The following object is masked from 'package:FactoMineR':
#>
#>     reconst
res.pca <- dudi.pca(iris[, -5], scannf = FALSE, nf = 5)
```

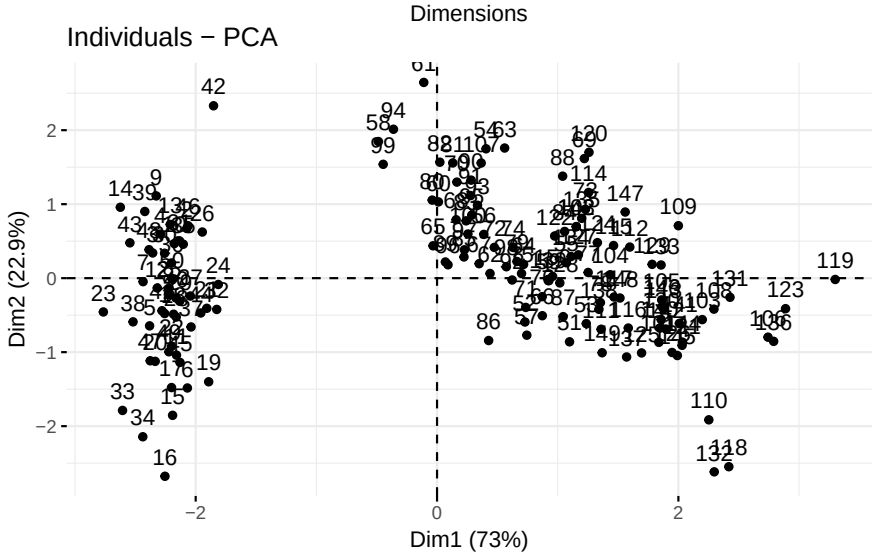
4. Using epPCA() [ExPosition]

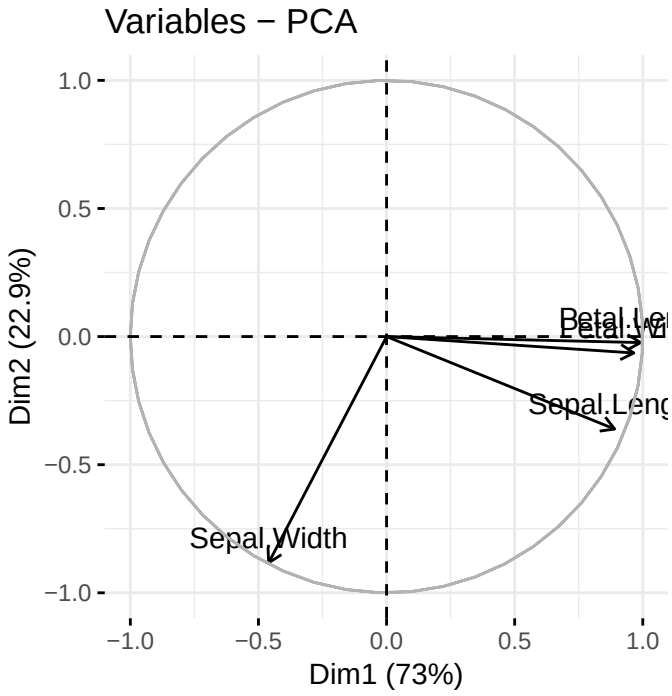
```
library(ExPosition)
#> Loading required package: prettyGraphs
res.pca <- epPCA(iris[, -5], graph = FALSE)
```

No matter what functions you decide to use, in the list above, the factoextra package can handle the output for creating beautiful plots similar to what we described in the previous sections for FactoMineR:

```
fviz_eig(res.pca)     # Scree plot
fviz_pca_ind(res.pca) # Graph of individuals
fviz_pca_var(res.pca) # Graph of variables
```






Variables − PCA
1.0
0.5
0.0
−0.5
−1.0
Dim2 (22.9%)
Petal.Le
Petal.W
Sepal.Leng
Sepal.Width
−1.0
−0.5
0.0
0.5
1.0
Dim1 (73%)

Chapter 3

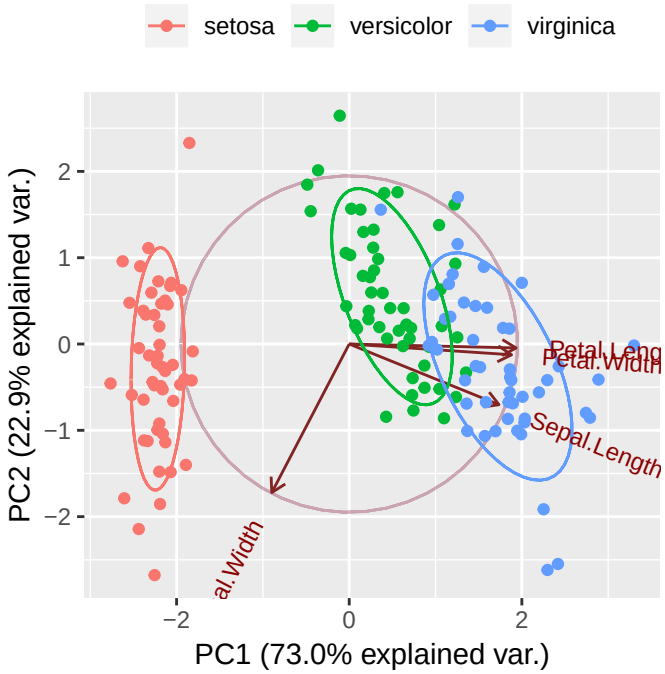
Biplot of the Iris data set

```
# devtools::install_github("vqv/ggbiplot")
library(ggbiplot)
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
#> Loading required package: plyr
#> Loading required package: scales
#> Loading required package: grid

iris.pca <- prcomp(iris[, 1:4], center = TRUE, scale = TRUE)
print(iris.pca)
#> Standard deviations (1, .., p=4):
#> [1] 1.708 0.956 0.383 0.144
#>
#> Rotation (n x k) = (4 x 4):
#>                 PC1     PC2    PC3    PC4
#> Sepal.Length  0.521 -0.3774  0.720  0.261
#> Sepal.Width  -0.269 -0.9233 -0.244 -0.124
#> Petal.Length  0.580 -0.0245 -0.142 -0.801
#> Petal.Width   0.565 -0.0669 -0.634  0.524
```

```
summary(iris.pca)
#> Importance of components:
#>                          PC1   PC2    PC3     PC4
#> Standard deviation     1.71 0.956 0.3831 0.14393
#> Proportion of Variance 0.73 0.229 0.0367 0.00518
#> Cumulative Proportion  0.73 0.958 0.9948 1.00000
```

```
g <- ggbiplot(iris.pca,
              obs.scale = 1,
              var.scale = 1,
              groups = iris$Species,
              ellipse = TRUE,
              circle = TRUE) +
    scale_color_discrete(name = "") +
```

```
    theme(legend.direction = "horizontal", legend.position = "top")

print(g)
```



The PC1 axis explains 0.730 of the variance, while the PC2 axis explains 0.229 of the variance.

3.1 Iris: underlying principal components

```
# Run PCA here with prcomp ()
iris.pca <- prcomp(iris[, 1:4], center = TRUE, scale = TRUE)

print(iris.pca)
#> Standard deviations (1, .., p=4):
#> [1] 1.708 0.956 0.383 0.144
#>
#> Rotation (n x k) = (4 x 4):
#>                 PC1     PC2    PC3    PC4
#> Sepal.Length  0.521 -0.3774  0.720  0.261
#> Sepal.Width  -0.269 -0.9233 -0.244 -0.124
#> Petal.Length  0.580 -0.0245 -0.142 -0.801
#> Petal.Width   0.565 -0.0669 -0.634  0.524
```

```
# Now, compute the new dataset aligned to the PCs by
# using the predict() function .
df.new <- predict(iris.pca, iris[, 1:4])
head(df.new)
#>        PC1    PC2     PC3      PC4
#> [1,] -2.26 -0.478  0.1273  0.02409
#> [2,] -2.07  0.672  0.2338  0.10266
#> [3,] -2.36  0.341 -0.0441  0.02828
```

```
#> [4,] -2.29  0.595 -0.0910 -0.06574
#> [5,] -2.38 -0.645 -0.0157 -0.03580
#> [6,] -2.07 -1.484 -0.0269  0.00659
```

```
# Show the PCA model's sdev values are the square root
# of the projected variances, which are along the diagonal
# of the covariance matrix of the projected data.
iris.pca$sdev^2
#> [1] 2.9185 0.9140 0.1468 0.0207
```

```
# # Compute covariance matrix for new dataset.
# Recall that the standard deviation is the square root of the variance.
round(cov(df.new), 5)
#>      PC1   PC2   PC3    PC4
#> PC1 2.92 0.000 0.000 0.0000
#> PC2 0.00 0.914 0.000 0.0000
#> PC3 0.00 0.000 0.147 0.0000
#> PC4 0.00 0.000 0.000 0.0207
```

3.2 Iris. Compute the eigenvectors and eigenvalues

```
# Scale and center the data.
df.scaled <- scale(iris[, 1:4], center = TRUE, scale = TRUE)

# Compute the covariance matrix.
cov.df.scaled <- cov(df.scaled)

# Compute the eigenvectors and eigen values.
# Each eigenvector (column) is a principal component.
# Each eigenvalue is the variance explained by the
# associated eigenvector.
eigenInformation <- eigen(cov.df.scaled)

print(eigenInformation)
#> eigen() decomposition
#> $values
#> [1] 2.9185 0.9140 0.1468 0.0207
#>
#> $vectors
#>        [,1]    [,2]   [,3]   [,4]
#> [1,]  0.521 -0.3774  0.720  0.261
#> [2,] -0.269 -0.9233 -0.244 -0.124
#> [3,]  0.580 -0.0245 -0.142 -0.801
#> [4,]  0.565 -0.0669 -0.634  0.524
```

```
# Now, compute the new dataset aligned to the PCs by
# multiplying the eigenvector and data matrices.


# Create transposes in preparation for matrix multiplication
eigenvectors.t <- t(eigenInformation$vectors)     # 4x4
df.scaled.t <- t(df.scaled)    # 4x150
```

```
# Perform matrix multiplication.
df.new <- eigenvectors.t %*% df.scaled.t   # 4x150

# Create new data frame. First take transpose and
# then add column names.
df.new.t <- t(df.new)    # 150x4
colnames(df.new.t) <- c("PC1", "PC2", "PC3", "PC4")

head(df.new.t)
#>         PC1    PC2     PC3      PC4
#> [1,] -2.26 -0.478  0.1273  0.02409
#> [2,] -2.07  0.672  0.2338  0.10266
#> [3,] -2.36  0.341 -0.0441  0.02828
#> [4,] -2.29  0.595 -0.0910 -0.06574
#> [5,] -2.38 -0.645 -0.0157 -0.03580
#> [6,] -2.07 -1.484 -0.0269  0.00659
```

```
# Compute covariance matrix for new dataset
round(cov(df.new.t), 5)
#>      PC1   PC2   PC3    PC4
#> PC1 2.92 0.000 0.000 0.0000
#> PC2 0.00 0.914 0.000 0.0000
#> PC3 0.00 0.000 0.147 0.0000
#> PC4 0.00 0.000 0.000 0.0207
```

Chapter 4

What is .hat in regression output

https://stats.stackexchange.com/a/256364/154908

Q. The augment() function in the broom package for R creates a dataframe of predicted values from a regression model. Columns created include the fitted values, the standard error of the fit and Cook's distance. They also include something with which I'm not familar and that is the column .hat.

```
library(broom)
data(mtcars)

m1 <- lm(mpg ~ wt, data = mtcars)

head(augment(m1))
#> # A tibble: 6 x 10
#>   .rownames   mpg    wt .fitted .se.fit .resid   .hat .sigma .cooksd
#>   <chr>     <dbl> <dbl>   <dbl>   <dbl>  <dbl>  <dbl>  <dbl>   <dbl>
#> 1 Mazda RX4  21    2.62    23.3   0.634 -2.28  0.0433   3.07 1.33e-2
#> 2 Mazda RX~  21    2.88    21.9   0.571 -0.920 0.0352   3.09 1.72e-3
#> 3 Datsun 7~  22.8  2.32    24.9   0.736 -2.09  0.0584   3.07 1.54e-2
#> 4 Hornet 4~  21.4  3.22    20.1   0.538  1.30  0.0313   3.09 3.02e-3
#> 5 Hornet S~  18.7  3.44    18.9   0.553 -0.200 0.0329   3.10 7.60e-5
#> 6 Valiant    18.1  3.46    18.8   0.555 -0.693 0.0332   3.10 9.21e-4
#> # ... with 1 more variable: .std.resid <dbl>
```

```
# .hat vector
augment(m1)$.hat
#>  [1] 0.0433 0.0352 0.0584 0.0313 0.0329 0.0332 0.0354 0.0313 0.0314 0.0329
#> [11] 0.0329 0.0558 0.0401 0.0419 0.1705 0.1953 0.1838 0.0661 0.1177 0.0956
#> [21] 0.0503 0.0343 0.0328 0.0443 0.0445 0.0866 0.0704 0.1291 0.0313 0.0380
#> [31] 0.0354 0.0377
```

Can anyone explain what this value is, and is it different between linear regression and logistic regression?

A. Those would be the diagonal elements of the hat-matrix which describe the leverage each point has on its fitted values.

If one fits:

$$\vec{Y} = \mathbf{X}\vec{\beta} + \vec{\epsilon}$$

then:

$$\mathbf{H} = \mathbf{X}(\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T$$

In this example:

$$\begin{pmatrix} Y_1 \\ \vdots \\ Y_{32} \end{pmatrix} = \begin{pmatrix} 1 & 2.620 \\ \vdots & \\ 1 & 2.780 \end{pmatrix} \cdot \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_{32} \end{pmatrix}$$

Then calculating this **H** matrix results in:

```
library(MASS)

wt <- mtcars[, 6]

X <- matrix(cbind(rep(1, length(wt)), wt), ncol=2)

H <- X %*% ginv(t(X) %*% X) %*% t(X)
```

Where this last matrix is a 32×32 matrix and contains these hat values on the diagonal.

```
X                           32x2
t(X)                        2x32
X %*% t(X)                  32x32
t(X) %*% X                  2x2
ginv(t(X) %*% X)            2x2
ginv(t(X) %*% X) %*% t(X)   2x32
X %*% ginv(t(X) %*% X)      32x2
```

```
dim(ginv(t(X) %*% X) %*% t(X))
#> [1]  2 32
```

```
x1 <- X %*% ginv(t(X) %*% X)
dim(x1)
#> [1] 32  2
dim(x1 %*% t(X))
#> [1] 32 32
```

```
x2 <- ginv(t(X) %*% X) %*% t(X)
dim(x2)
#> [1]  2 32
dim(X %*% x2)
#> [1] 32 32
```

```
# this last matrix is a 32×32 matrix and contains these hat values on the diagonal.
diag(H)
#>  [1] 0.0433 0.0352 0.0584 0.0313 0.0329 0.0332 0.0354 0.0313 0.0314 0.0329
#> [11] 0.0329 0.0558 0.0401 0.0419 0.1705 0.1953 0.1838 0.0661 0.1177 0.0956
#> [21] 0.0503 0.0343 0.0328 0.0443 0.0445 0.0866 0.0704 0.1291 0.0313 0.0380
#> [31] 0.0354 0.0377
```

Chapter 5

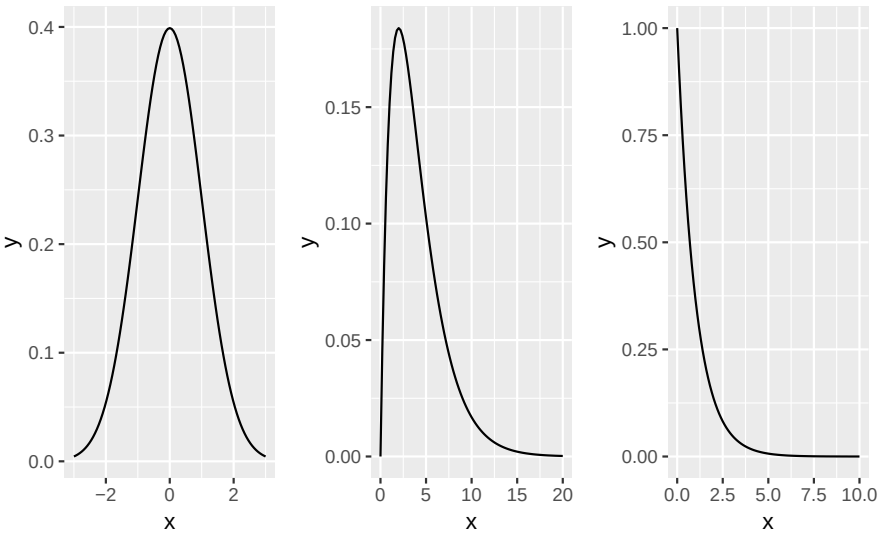
Q-Q normal to compare data to distributions

5.1 Introduction

https://mgimond.github.io/ES218/Week06a.html

Thus far, we have used the quantile-quantile plots to compare the distributions between two empirical (i.e. observational) datasets. This is sometimes referred to as an empirical **Q-Q** plot. We can also use the q-q plot to compare an *empirical* observation to a *theoretical* observation (i.e. one defined mathematically). Such a plot is usually referred to as a **theoretical Q-Q plot**. Examples of popular theoretical observations are the normal distribution (aka the Gaussian distribution), the **chi-square** distribution, and the exponential distribution just to name a few.

```
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
```



5.2 Why we want to compare emprirical vs theoretical distributions

There are many reasons we might want to compare empirical data to theoretical distributions:

- A theoretical distribution is easy to parameterize. For example, if the shape of the distribution of a batch of numbers can be approximated by a normal distribution we can reduce the complexity of our data to just two values: the mean and the standard deviation.
- If data can be approximated by certain theoretical distributions, then many mainstream statistical procedures can be applied to the data.
- In inferential statistics, knowing that a sample was derived from a population whose distribution follows a theoretical distribution allows us to derive certain properties of the population from the sample. For example, if we know that a sample comes from a normally distributed population, we can define confidence intervals for the sample mean using a **t-distribution**.
- Modeling the distribution of the observed data can provide insight into the underlying process that generated the data.

But very few empirical datasets follow any theoretical distributions exactly. So the questions usually ends up being "how well does theoretical distribution X fit my data?"

The theoretical quantile-quantile plot is a tool to explore how a batch of numbers deviates from a theoretical distribution and to visually assess whether the difference is significant for the purpose of the analysis. In the following examples, we will compare empirical data to the normal distribution using the normal quantile-quantile plot.

5.3 The normal q-q plot

The normal q-q plot is just a special case of the empirical q-q plot we've explored so far; the difference being that we assign the normal distribution quantiles to the x-axis.

5.3.1 Drawing a normal q-q plot from scratch

In the following example, we'll compare the Alto 1 group to a normal distribution. First, we'll extract the Alto 1 height values and save them as an atomic vector object using dplyr's piping operations.

However, dplyr's operations will return a dataframe–even if a single column is selected. To force the output to an atomic vector, we'll pipe the subset to `pull(height)` which will extract the height column into a plain vector element.

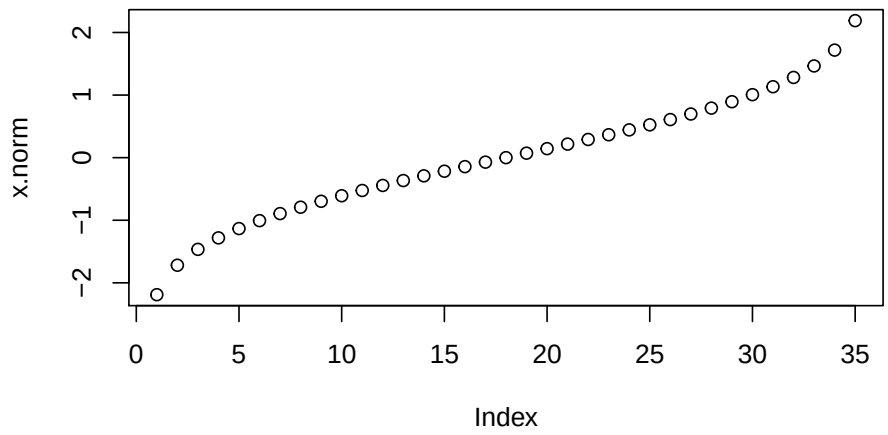
```
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following object is masked from 'package:gridExtra':
#>
#>     combine
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
```

```
df   <- lattice::singer
alto <- df %>%
    filter(voice.part == "Alto 1") %>%
    arrange(height) %>%
    pull(height) %>%
    print
#>  [1] 60 61 61 61 61 62 62 62 63 63 63 63 64 64 64 65 65 65 65 66 66 66 66
#> [24] 66 66 66 67 67 67 67 68 68 69 70 72
```

Next, we need to find the matching normal distribution quantiles. We first find the f-values for alto, then use qnorm to find the matching normal distribution values from those same f-values

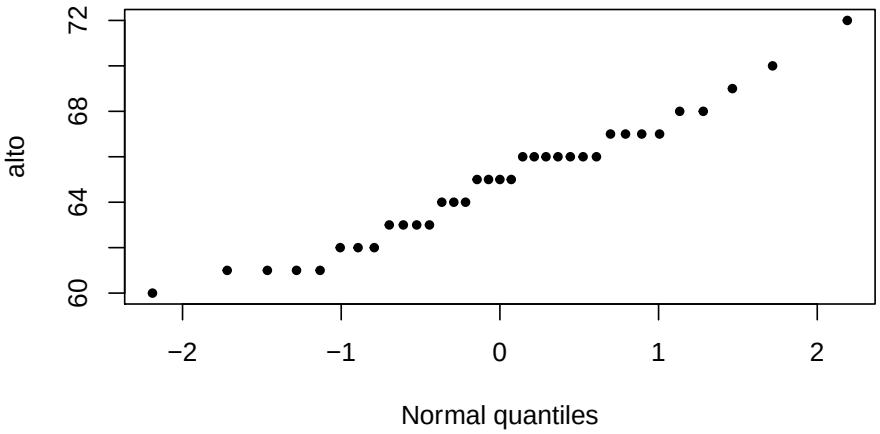
```
i <- 1:length(alto)
fi <- (i - 0.5) / length(alto)
fi
#>  [1] 0.0143 0.0429 0.0714 0.1000 0.1286 0.1571 0.1857 0.2143 0.2429 0.2714
#> [11] 0.3000 0.3286 0.3571 0.3857 0.4143 0.4429 0.4714 0.5000 0.5286 0.5571
#> [21] 0.5857 0.6143 0.6429 0.6714 0.7000 0.7286 0.7571 0.7857 0.8143 0.8429
#> [31] 0.8714 0.9000 0.9286 0.9571 0.9857
x.norm <- qnorm(fi)
x.norm
#>  [1] -2.1893 -1.7185 -1.4652 -1.2816 -1.1332 -1.0063 -0.8938 -0.7916
#>  [9] -0.6971 -0.6085 -0.5244 -0.4439 -0.3661 -0.2905 -0.2165 -0.1437
#> [17] -0.0717  0.0000  0.0717  0.1437  0.2165  0.2905  0.3661  0.4439
#> [25]  0.5244  0.6085  0.6971  0.7916  0.8938  1.0063  1.1332  1.2816
#> [33]  1.4652  1.7185  2.1893
```

```
plot(x.norm)
```



Now we can plot the sorted alto values against the normal values.

```
plot( alto ~ x.norm, type="p", xlab="Normal quantiles", pch=20)
```



When comparing a batch of numbers to a theoretical distribution on a q-q plot, we are looking for significant deviation from a straight line. To make it easier to judge straightness, we can fit a line to the points. Note that we are not creating a 45° (or x=y) slope; the range of values between both sets of numbers do not match. Here, we are only seeking the straightness of the points.

There are many ways one can fit a line to the data, Cleveland opts to fit a line to the first and third quartile of the q-q plot. The following chunk of code identifies the quantiles for both the alto dataset and the theoretical normal distribution. It then computes the slope and intercept from these coordinates.

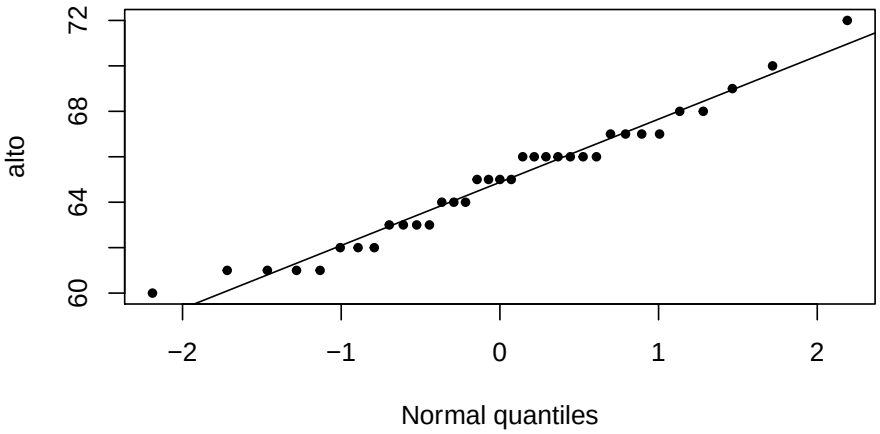
```
# Find 1st and 3rd quartile for the Alto 1 data
y <- quantile(alto, c(0.25, 0.75), type=5)
y
#>  25%  75%
#> 63.0 66.8
```

```
# Find the 1st and 3rd quartile of the normal distribution
x <- qnorm( c(0.25, 0.75))
x
#> [1] -0.674  0.674
```

```
# Now we can compute the intercept and slope of the line that passes
# through these points
slope <- diff(y) / diff(x)
int   <- y[1] - slope * x[1]
```

Next, we add the line to the plot.

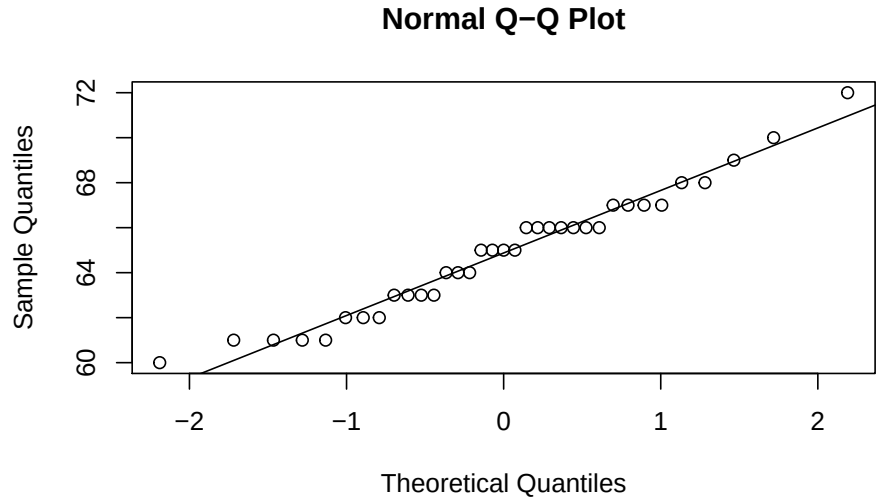
```
plot( alto ~ x.norm, type="p", xlab="Normal quantiles", pch=20)
abline(a=int, b=slope )
```



5.4 Using R's built-in functions

R has two built-in functions that facilitate the plot building task when comparing a batch to a normal distribution: `qqnorm` and `qqline`. Note that the function `qqline` allows the user to define the quantile method via the qtype= parameter. Here, we set it to 5 to match our choice of f-value calculation.

```
qqnorm(alto)           # plot the points
qqline(alto, qtype=5)  # plot the line
```



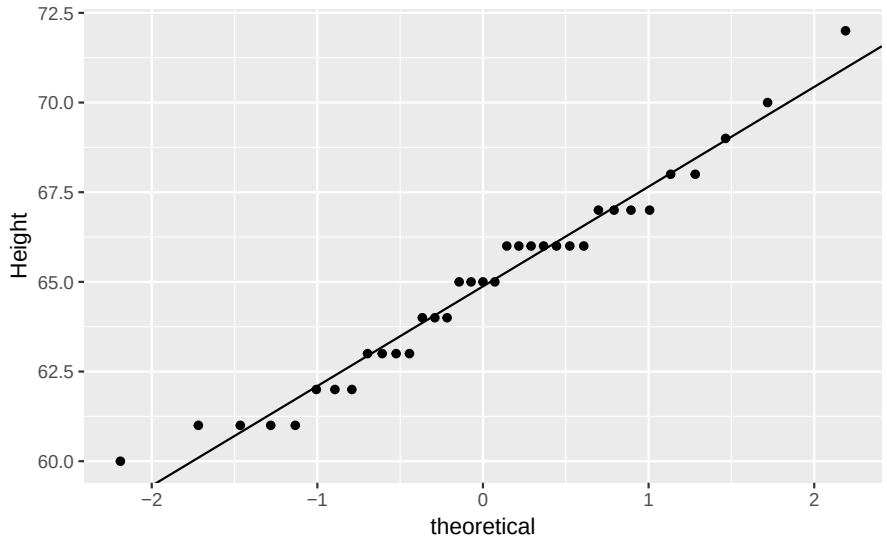
That's it. Just two lines of code!

5.5 Using the ggplot2 plotting environment

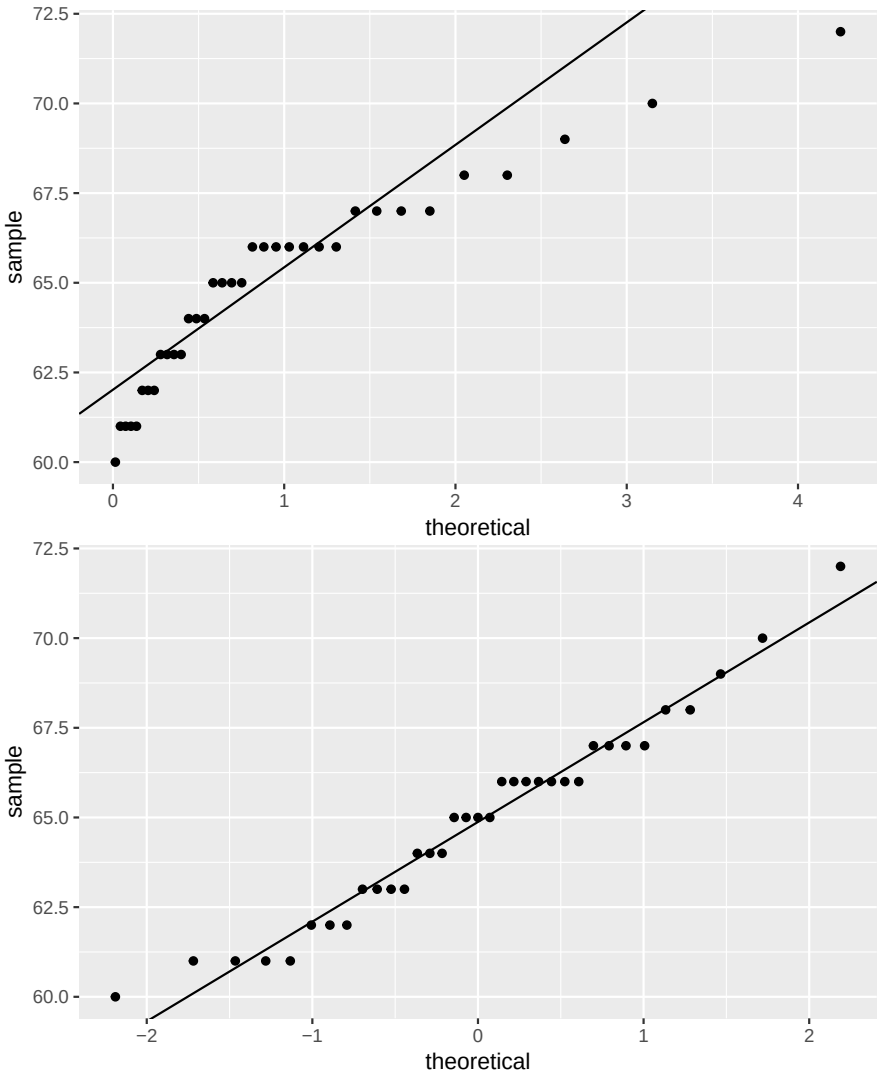
We can take advantage of the `stat_qq()` function to plot the points, but the equation for the line must be computed manually (as was done earlier). Those steps will be repeated here.

```
# normal distribution
library(ggplot2)

# Find the slope and intercept of the line that passes through the 1st and 3rd
# quartile of the normal q-q plot
```

```
y     <- quantile(alto, c(0.25, 0.75), type=5) # Find the 1st and 3rd quartiles
x     <- qnorm( c(0.25, 0.75))                 # Find the matching normal values on the x-axis
slope <- diff(y) / diff(x)                     # Compute the line slope
int   <- y[1] - slope * x[1]                   # Compute the line intercept

# Generate normal q-q plot
ggplot() + aes(sample=alto) +
    stat_qq(distribution=qnorm) +
    geom_abline(intercept=int, slope=slope) +
    ylab("Height")
```




```
qq_any <- function(var, f) {
    # Find the slope and intercept of the line that passes through the 1st and 3rd
    # quartile of the normal q-q plot

    y     <- quantile(var, c(0.25, 0.75), type=5) # Find the 1st and 3rd quartiles
    x     <- f( c(0.25, 0.75))                    # Find the matching normal values x-axis
    slope <- diff(y) / diff(x)                        # Compute the line slope
    int   <- y[1] - slope * x[1]                      # Compute the line intercept
    ggplot() + aes(sample = var) +
    stat_qq(distribution = f) +
    geom_abline(intercept=int, slope=slope)
}

# two function only, for the moment
qq_any(alto, qexp)
qq_any(alto, qnorm)
```



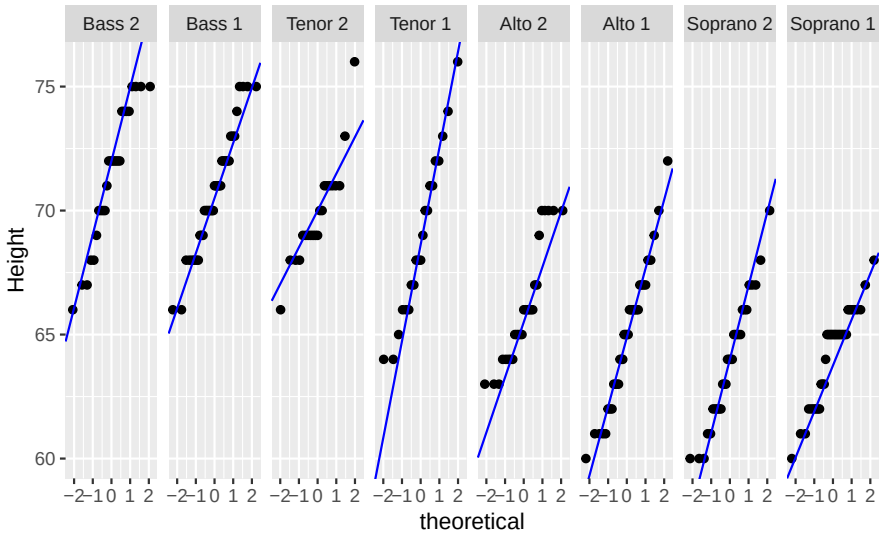
We can, of course, make use of ggplot's faceting function to generate trellised plots. For example, the following plot replicates Cleveland's figure 2.11 (except for the layout which we'll setup as a single row of plots instead). But first, we will need to compute the slopes for each singer group. We'll use dplyr's piping operations to create a new dataframe with singer group name, slope and intercept.

```
library(dplyr)

intsl <- df %>%
    group_by(voice.part) %>%
       summarize(q25    = quantile(height,0.25, type=5),
                 q75    = quantile(height,0.75, type=5),
                 norm25 = qnorm( 0.25),
                 norm75 = qnorm( 0.75),
                 slope  = (q25 - q75) / (norm25 - norm75),
                 int    = q25 - slope * norm25) %>%
       select(voice.part, slope, int) %>%
    print
#> # A tibble: 8 x 3
#>   voice.part slope   int
#>   <fct>      <dbl> <dbl>
```

```
#> 1 Bass 2      2.97  72
#> 2 Bass 1      2.22  70.5
#> 3 Tenor 2     1.48  70
#> 4 Tenor 1     3.89  68.6
#> 5 Alto 2      2.22  65.5
#> 6 Alto 1      2.78  64.9
#> # ... with 2 more rows
```

It's important that the `voice.part` names match those in `df` letter-for-letter so that when ggplot is called, it will know which facet to assign the slope and intercept values to via `geom_abline`.

```
ggplot(df, aes(sample = height)) +
    stat_qq(distribution = qnorm) +
    geom_abline(data=intsl, aes(intercept=int, slope=slope), col="blue") +
    facet_wrap(~voice.part, nrow=1) +
    ylab("Height")
```



Chapter 6

QQ and PP Plots

https://homepage.divms.uiowa.edu/~luke/classes/STAT4580/qqpp.html

6.1 QQ Plot

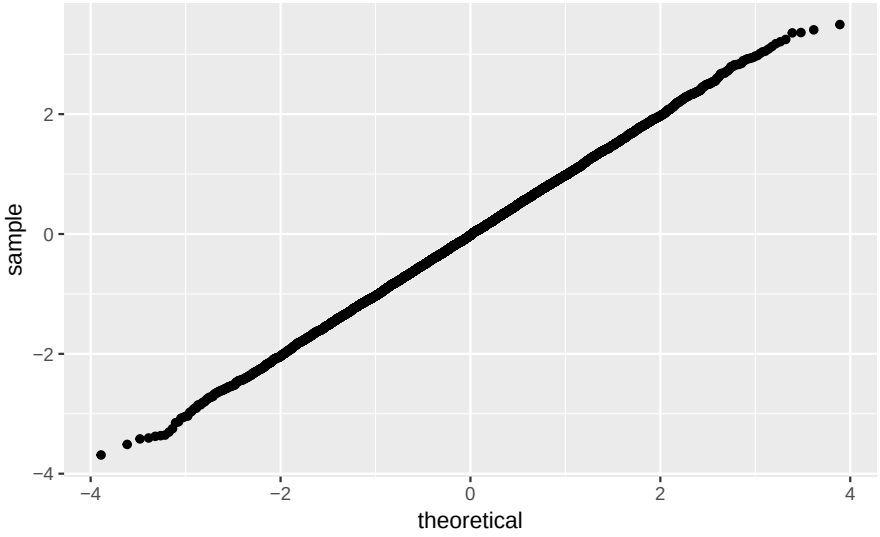
One way to assess how well a particular theoretical model describes a data distribution is to plot data quantiles against theoretical quantiles.

Base graphics provides `qqnorm`, lattice has `qqmath`, and ggplot2 has `geom_qq`.

The default theoretical distribution used in these is a standard normal, but, except for `qqnorm`, these allow you to specify an alternative.

For a large sample from the theoretical distribution the plot should be a straight line through the origin with slope 1:

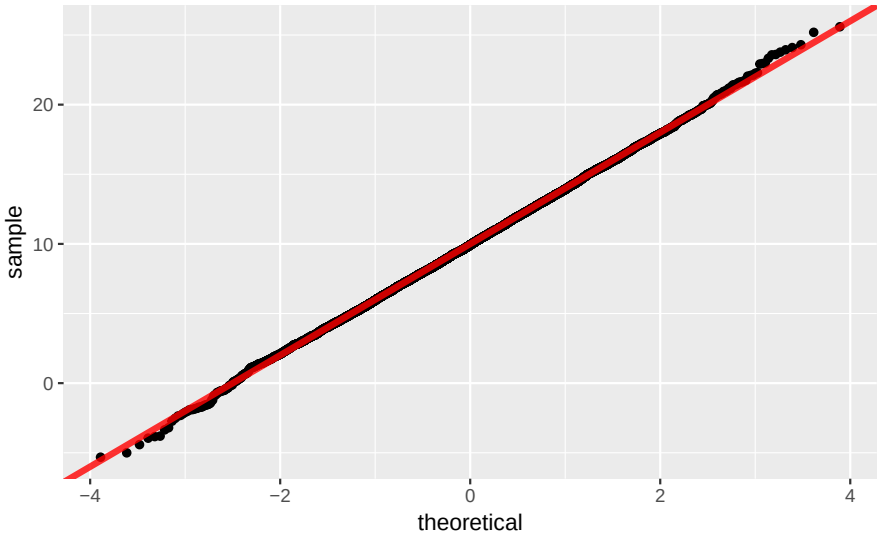
```
library(ggplot2)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang

n <- 10000
ggplot() + geom_qq(aes(sample = rnorm(n)))
```



If the plot is a straight line with a different slope or intercept, then the data distribution corresponds to a location-scale transformation of the theoretical distribution.

The slope is the scale and the intercept is the location:

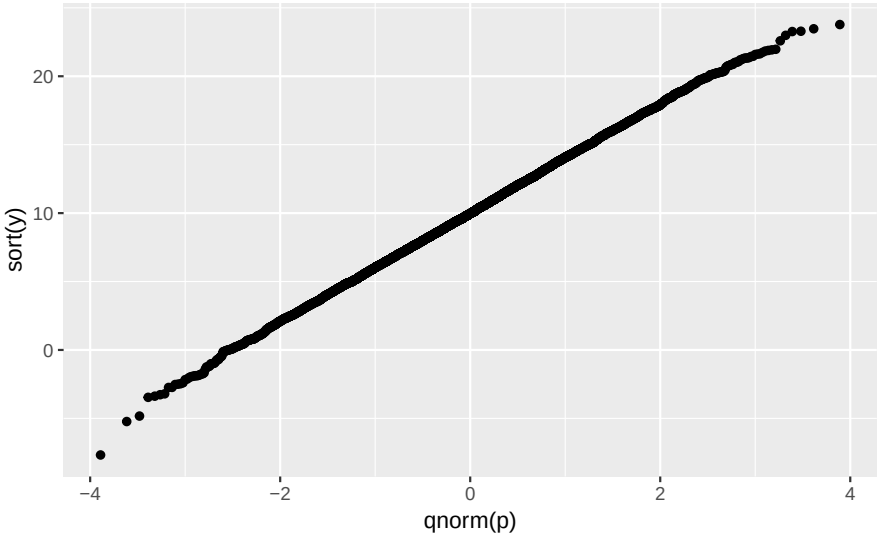
```
ggplot() +
    geom_qq(aes(sample = rnorm(n, 10, 4))) +
    geom_abline(intercept = 10, slope = 4,
                color = "red", size = 1.5, alpha = 0.8)
```



The QQ plot can be constructed directly as a scatterplot of the sorted sample $i = 1, \ldots, n$ against quantiles for

$$p_i = \frac{i}{n} - \frac{1}{2n}$$

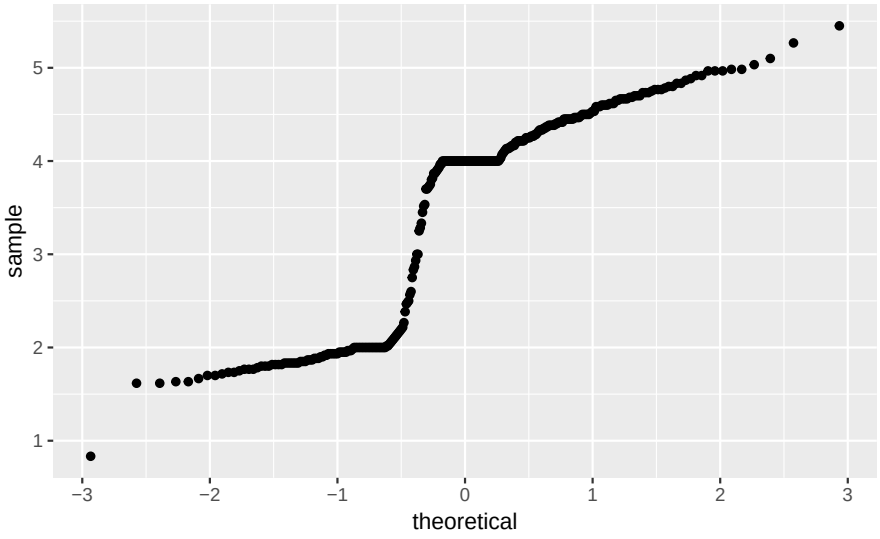
```
p <- (1 : n) / n - 0.5 / n
y <- rnorm(n, 10, 4)
ggplot() + geom_point(aes(x = qnorm(p), y = sort(y)))
```



6.2 Some Examples

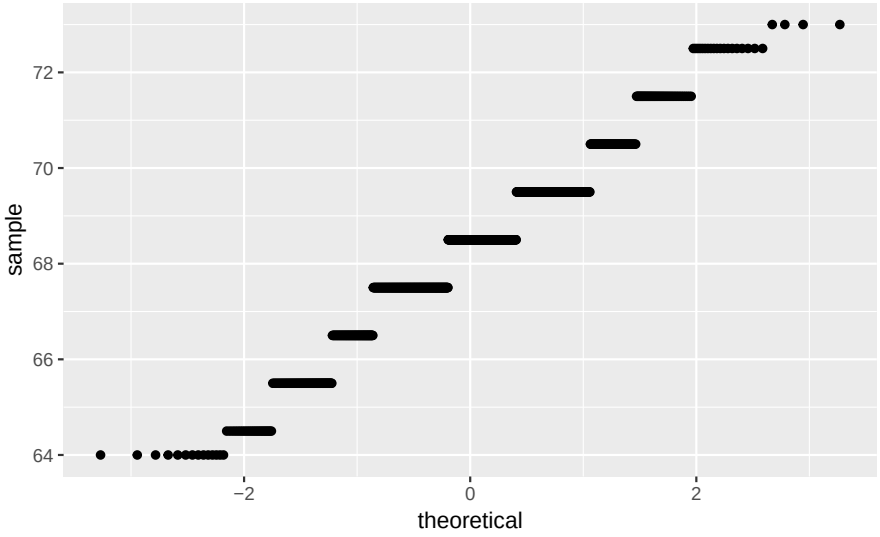
The histograms and density estimates for the duration variable in the `geyser` data set showed that the distribution is far from a normal distribution, and the normal QQ plot shows this as well:

```
library(MASS)
ggplot(geyser) + geom_qq(aes(sample = duration))
```



Except for rounding the parent heights in the Galton data seemed not too fat from normally distributed:

```
library(psych)

ggplot(galton) + geom_qq(aes(sample = parent))
```

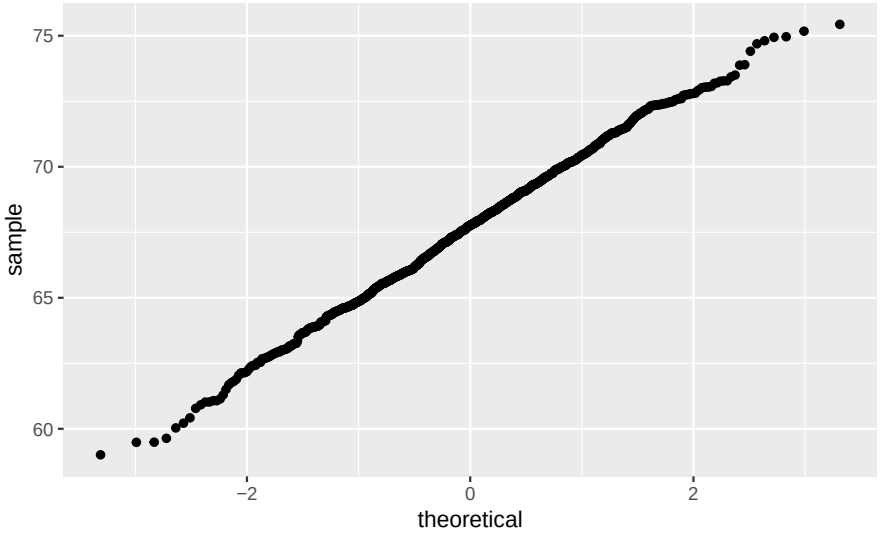


Rounding interferes more with this visualization than with a histogram or a density plot.

Rounding is more visible with this visualization than with a histogram or a density plot.

Another Gatlton dataset available in the UsingR package with less rounding is father.son:

```
library(UsingR)
ggplot(father.son) + geom_qq(aes(sample = fheight))
```



The middle seems to be fairly straight, but the ends are somewhat wiggly.

How can you calibrate your judgment?

6.3 Calibrating the Variability

One approach is to use simulation, sometimes called a graphical bootstrap.

The `nboot` function will simulate R samples from a normal distribution that match a variable x on sample size, sample mean, and sample SD.

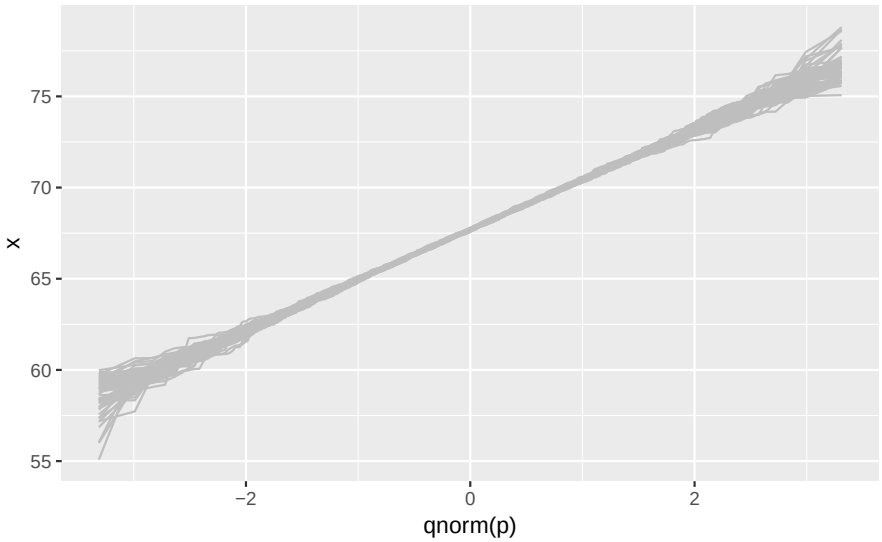
The result is returned in a dataframe suitable for plotting:

```
nsim <- function(n, m = 0, s = 1) {
    z <- rnorm(n)
    m + s * ((z - mean(z)) / sd(z))
}

nboot <- function(x, R) {
    n <- length(x)
    m <- mean(x)
    s <- sd(x)
    do.call(rbind,
            lapply(1 : R,
                   function(i) {
                       xx <- sort(nsim(n, m, s))
                       p <- seq_along(x) / n - 0.5 / n
                       data.frame(x = xx, p = p, sim = i)
    }))
}
```

Plotting these as lines shows the variability in shapes we can expect when sampling from the theoretical normal distribution:

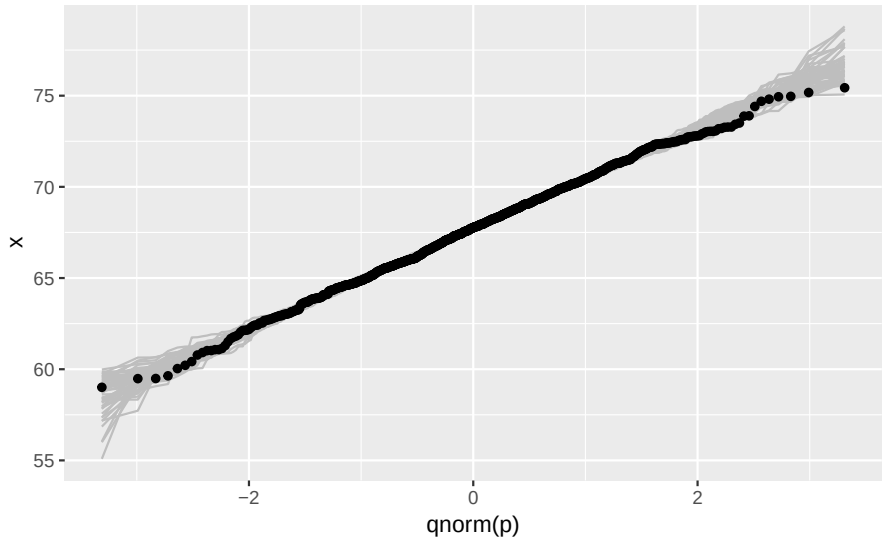
```
gb <- nboot(father.son$fheight, 50)
tibble::as_tibble(gb)
#> # A tibble: 53,900 x 3
#>       x        p   sim
#>   <dbl>    <dbl> <int>
#> 1  59.8 0.000464     1
#> 2  59.9 0.00139      1
#> 3  59.9 0.00232      1
#> 4  60.8 0.00325      1
#> 5  60.8 0.00417      1
#> 6  60.9 0.00510      1
#> # ... with 5.389e+04 more rows
```

```
ggplot() +
    geom_line(aes(x = qnorm(p), y = x, group = sim),
              color = "gray", data = gb)
```



We can then insert this simulation behind our data to help calibrate the visualization:

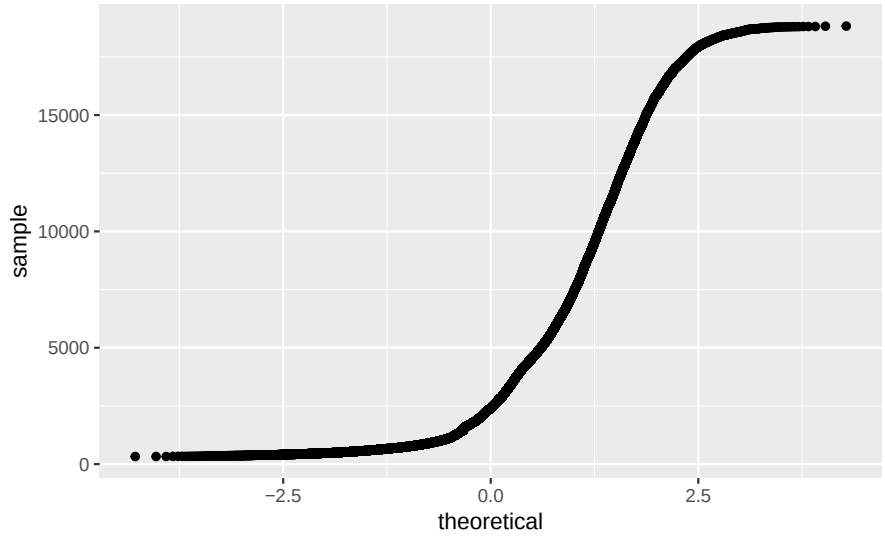
```
ggplot(father.son) +
    geom_line(aes(x = qnorm(p), y = x, group = sim),
              color = "gray", data = gb) +
    geom_qq(aes(sample = fheight))
```



6.4 Scalability

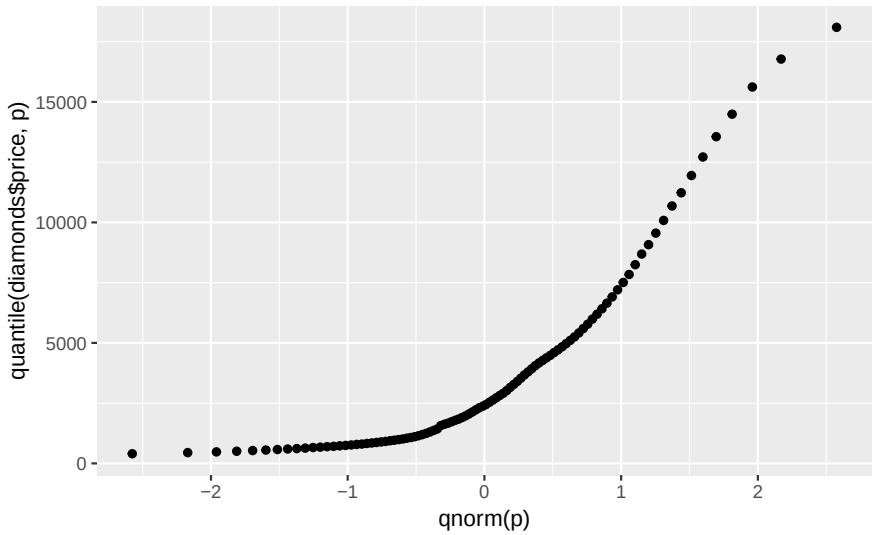
For large sample sizes overplotting will occur:

```
ggplot(diamonds) + geom_qq(aes(sample = price))
```



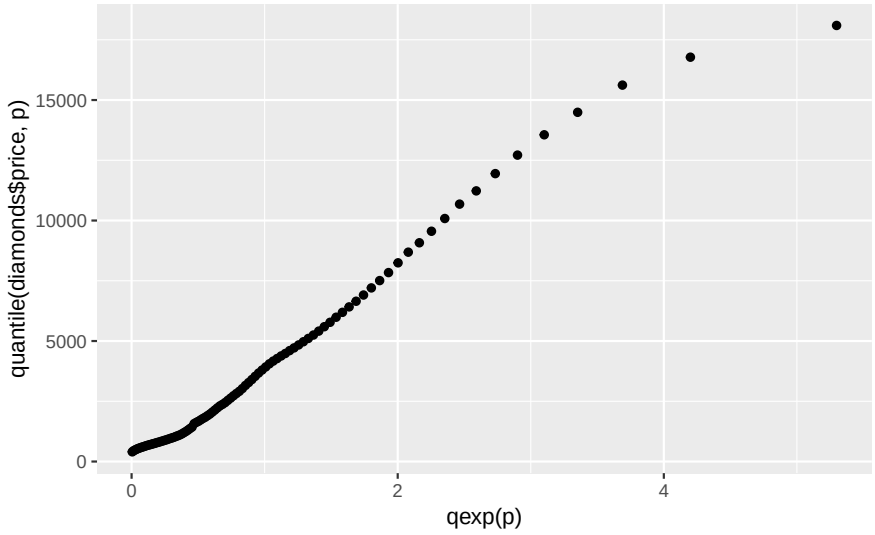
This can be alleviated by using a grid of quantiles:

```
nq <- 100
p <- (1 : nq) / nq - 0.5 / nq
ggplot() + geom_point(aes(x = qnorm(p), y = quantile(diamonds$price, p)))
```



A more reasonable model might be an exponential distribution:

```
ggplot() + geom_point(aes(x = qexp(p), y = quantile(diamonds$price, p)))
```



6.5 Comparing Two Distributions

The QQ plot can also be used to compare two distributions based on a sample from each.

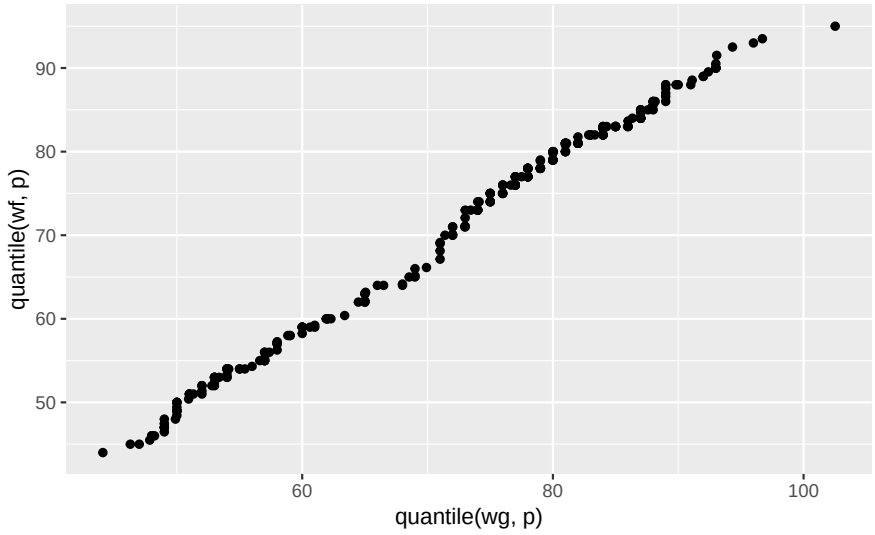
If the samples are the same size then this is just a plot of the ordered sample values against each other.

Choosing a fixed set of quantiles allows samples of unequal size to be compared.

Using a small set of quantiles we can compare the distributions of waiting times between eruptions of Old Faithful from the two different data sets we have looked at:

```
nq <- 31  # user defined
nq <- min(length(geyser$waiting), length(faithful$waiting)) # or take the minimum
p <- (1 : nq) / nq - 0.5 / nq

wg <- geyser$waiting
```

```
wf <- faithful$waiting

ggplot() + geom_point(aes(x = quantile(wg, p), y = quantile(wf, p)))
```



6.6 PP Plots

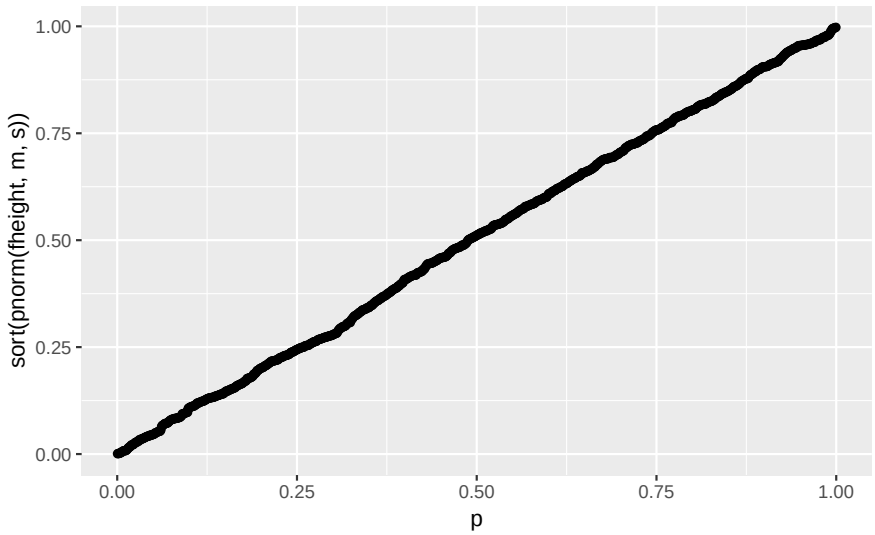
The PP plot for comparing a sample to a theoretical model plots the theoretical proportion less than or equal to each observed value against the actual proportion.

For a theoretical cumulative distribution function F this means plotting

$$F(x(i))pi$$

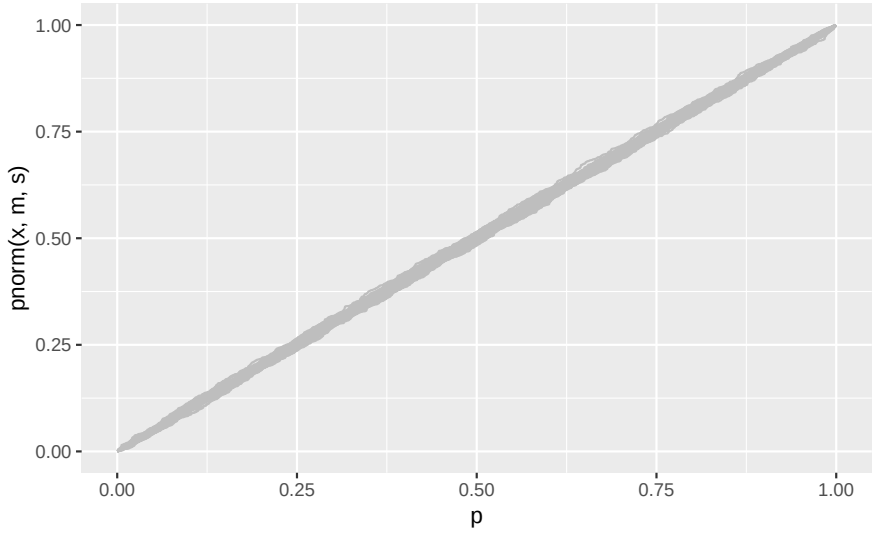
For the `fheight` variable in the `father.son` data:

```
m <- mean(father.son$fheight)
s <- sd(father.son$fheight)
n <- nrow(father.son)
p <- (1 : n) / n - 0.5 / n
ggplot(father.son) + geom_point(aes(x = p, y = sort(pnorm(fheight, m, s))))
```



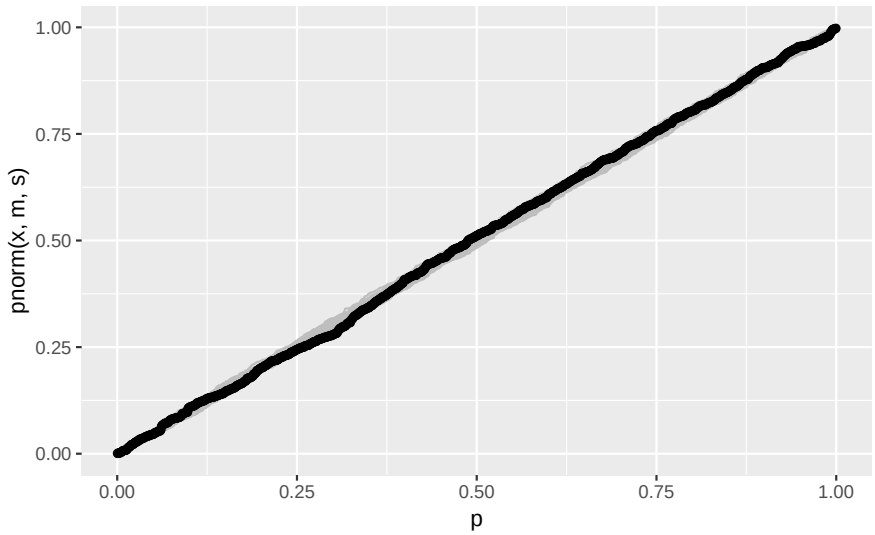
- The values on the vertical axis are the probability integral transform of the data for the theoretical distribution.
- If the data are a sample from the theoretical distribution then these transforms would be uniformly distributed on [0,1].
- The PP plot is a QQ plot of these transformed values against a uniform distribution.
- The PP plot goes through the points (0,0) and (1,1) and so is much less variable in the tails:

```
pp <- ggplot() +
geom_line(aes(x = p, y = pnorm(x, m, s), group = sim),
          color = "gray", data = gb)
pp
```



Adding the data:

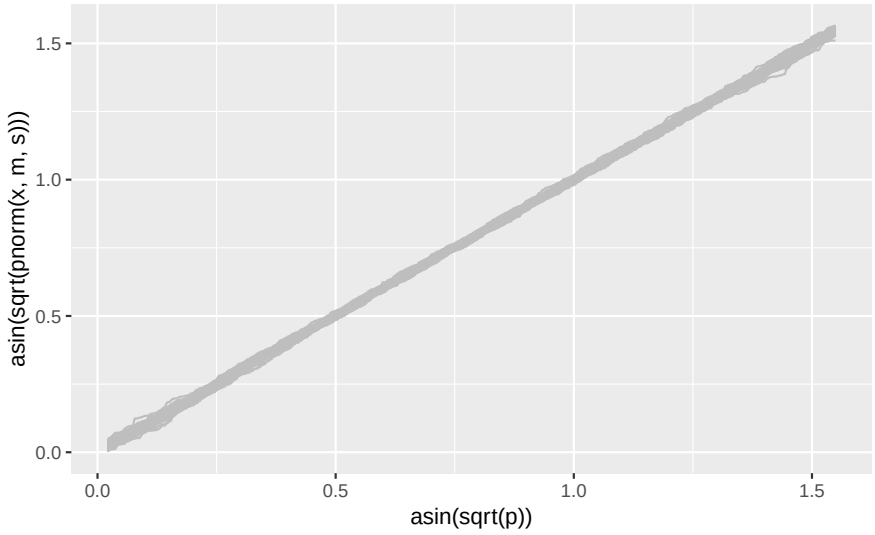
```
pp +
geom_point(aes(x = p, y = sort(pnorm(fheight, m, s))), data = (father.son))
```



The PP plot is also less sensitive to deviations in the tails.

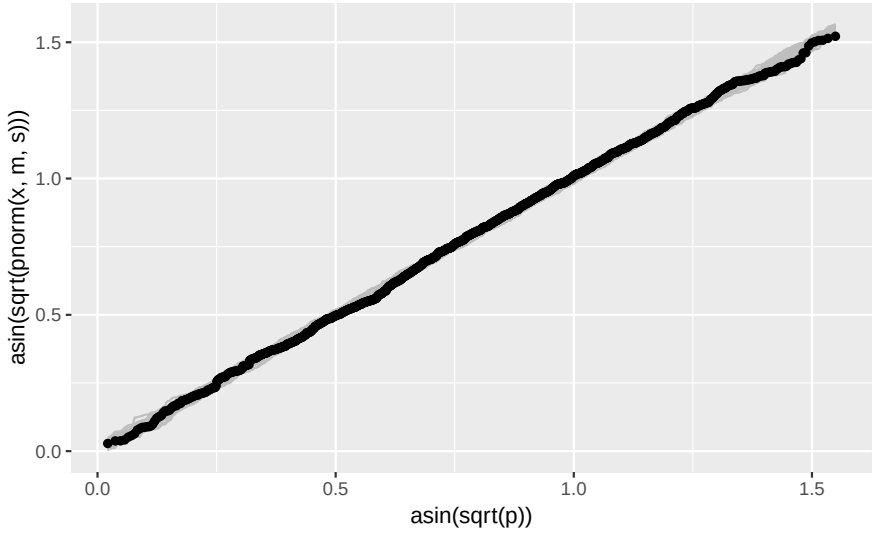
A compromise between the QQ and PP plots uses the arcsine square root variance-stabilizing transformation, which makes the variability approximately constant across the range of the plot:

```
vpp <- ggplot() +
geom_line(aes(x = asin(sqrt(p)), y = asin(sqrt(pnorm(x, m, s))), group = sim),
          color = "gray", data = gb)
vpp
```



Adding the data:

```
vpp +
geom_point(aes(x = asin(sqrt(p)), y = sort(asin(sqrt(pnorm(fheight, m, s))))),
           data = (father.son))
```



6.7 Plots For Assessing Model Fit

- Both QQ and PP plots can be used to asses how well a theoretical family of models fits your data, or your residuals.
- To use a PP plot you have to estimate the parameters first.
- For a location-scale family, like the normal distribution family, you can use a QQ plot with a standard member of the family.
- Some other families can use other transformations that lead to straight lines for family members:

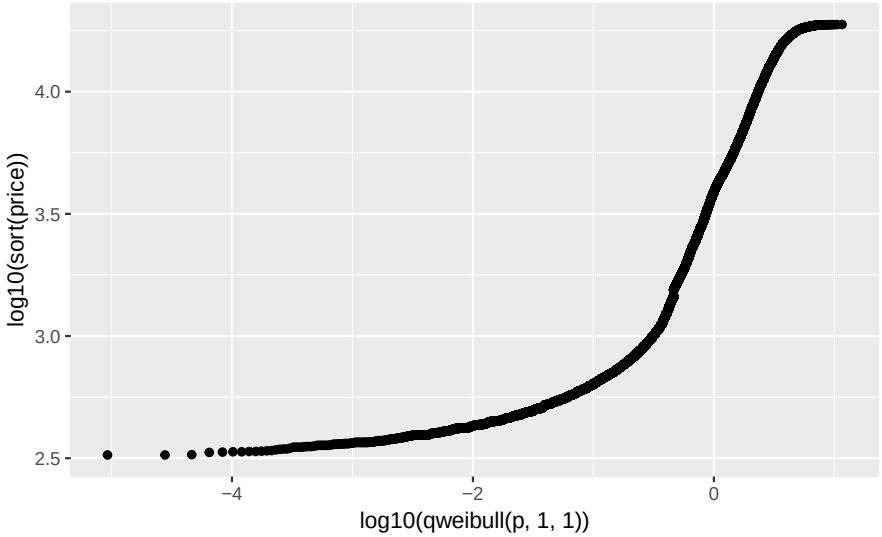
The Weibull family is widely used in reliability modeling; its CDF is

$$F(t) = 1 - \exp\left\{-\left(\frac{t}{b}\right)^a\right\}$$

- The logarithms of Weibull random variables form a location-scale family.
- Special paper used to be available for Weibull probability plots.

A Weibull QQ plot for price in the diamonds data:

```
n <- nrow(diamonds)
p <- (1 : n) / n - 0.5 / n
ggplot(diamonds) +
    geom_point(aes(x = log10(qweibull(p, 1, 1)), y = log10(sort(price))))
```



- The lower tail does not match a Weibull distribution.
- Is this important?
- In engineering applications it often is.
- In selecting a reasonable model to capture the shape of this distribution it may not be.
- QQ plots are helpful for understanding departures from a theoretical model.
- No data will fit a theoretical model perfectly.
- Case-specific judgment is needed to decide whether departures are important.
- George Box: All models are wrong but some are useful.

Chapter 7

Data Visualization: Working with models

7.1 Introduction

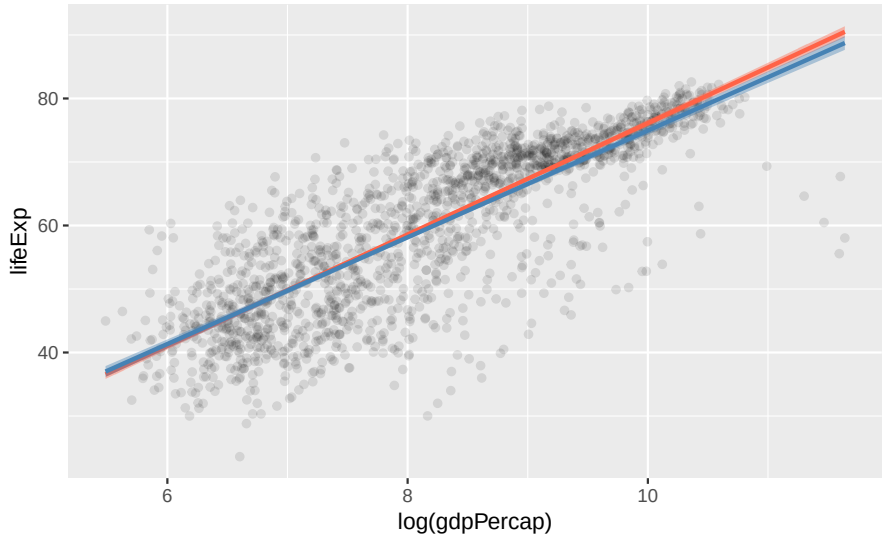
Source: https://socviz.co/modeling.html

Data visualization is about more than generating figures that display the raw numbers from a table of data. Right from the beginning, it involves summarizing or transforming parts of the data, and then plotting the results. Statistical models are a central part of that process. In this Chapter, we will begin by looking briefly at how `ggplot` can use various modeling techniques directly within geoms. Then we will see how to use the `broom` and `margins` libraries to tidily extract and plot estimates from models that we fit ourselves.

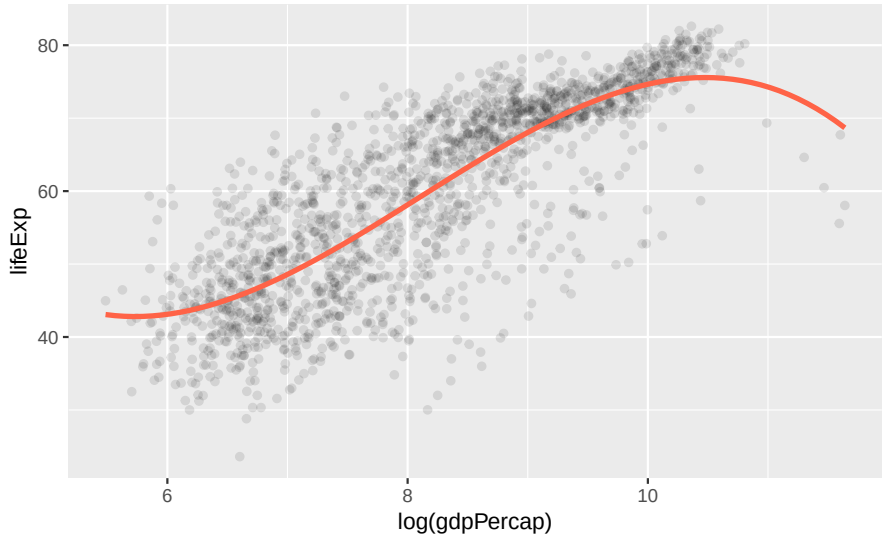
```
# load libraries
library(ggplot2)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method         from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
library(tidyr)
library(purrr)
library(socviz)       # devtools::install_github("kjhealy/socviz")
library(gapminder)
```

```
# plot two lines
p <- ggplot(data = gapminder,
            mapping = aes(x = log(gdpPercap), y = lifeExp))
```

```
p + geom_point(alpha=0.1) +
    geom_smooth(color = "tomato", fill="tomato", method = MASS::rlm) +
    geom_smooth(color = "steelblue", fill="steelblue", method = "lm")
```

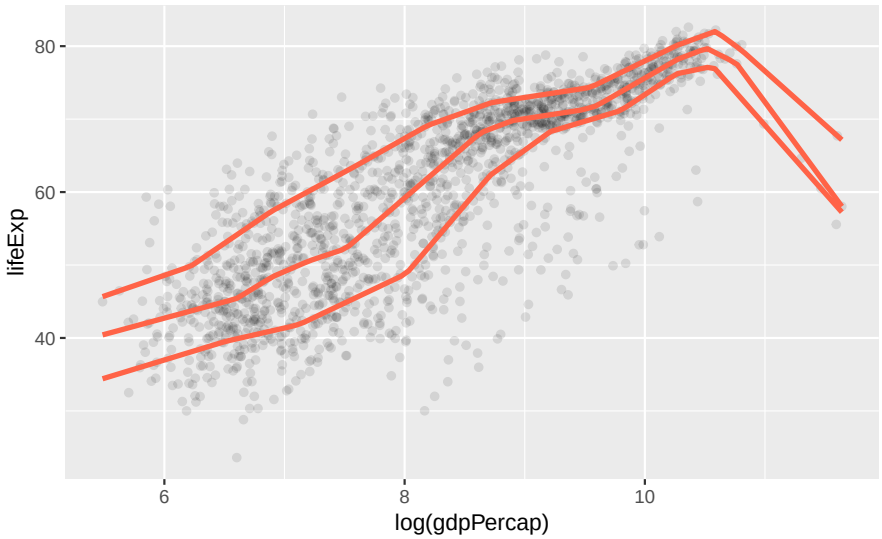



```
# plot spline
p + geom_point(alpha=0.1) +
    geom_smooth(color = "tomato", method = "lm", size = 1.2,
                formula = y ~ splines::bs(x, 3), se = FALSE)
```




```
p + geom_point(alpha=0.1) +
    geom_quantile(color = "tomato", size = 1.2, method = "rqss",
                  lambda = 1, quantiles = c(0.20, 0.5, 0.85))
#> Loading required package: SparseM
#>
#> Attaching package: 'SparseM'
#> The following object is masked from 'package:base':
#>
#>     backsolve
#> Smoothing formula not specified. Using: y ~ qss(x, lambda = 1)
```

```
#> Warning in rq.fit.sfn(x, y, tau = tau, rhs = rhs, control = control, ...): tiny diagonals replaced w
```



Histograms, density plots, boxplots, and other geoms compute either single numbers or new variables before plotting them. As we saw in Section 4.4, these calculations are done by `stat_` functions, each of which works hand-in-hand with its default `geom_` function, and vice versa. Moreover, from the smoothing lines we drew from almost the very first plots we made, we have seen that `stat_` functions can do a fair amount of calculation and even model estimation on the fly. The `geom_smooth()` function can take a range of method arguments to fit LOESS, OLS, and robust regression lines, amongst others.

Both the `geom_smooth()` and `geom_quantile()` functions can also be instructed to use different formulas to produce their fits. In the top panel of Figure 6.1, we access the `MASS` library's `rlm` function to fit a robust regression line. In the second panel, the `bs` function is invoked directly from the `splines` library in the same way, to fit a polynominal curve to the data. This is the same approach to directly accessing functions without loading a whole library that we have already used several times when using functions from the `scales` library. The `geom_quantile()` function, meanwhile, is like a specialized version of `geom_smooth()` that can fit quantile regression lines using a variety of methods. The `quantiles` argument takes a vector specifying the quantiles at which to fit the lines.

7.2 Show several fits at once, with a legend

As we just saw in the first panel of Figure 6.1, where we plotted both an OLS and a robust regression line, we can look at several fits at once on the same plot by layering on new smoothers with `geom_smooth()`. As long as we set the color and fill aesthetics to different values for each fit, we can easily distinguish them visually. However, `ggplot` will not draw a legend that guides us about which fit is which. This is because the smoothers are not logically connected to one another. They exist as separate layers. What if we are comparing several different fits and want a legend describing them?

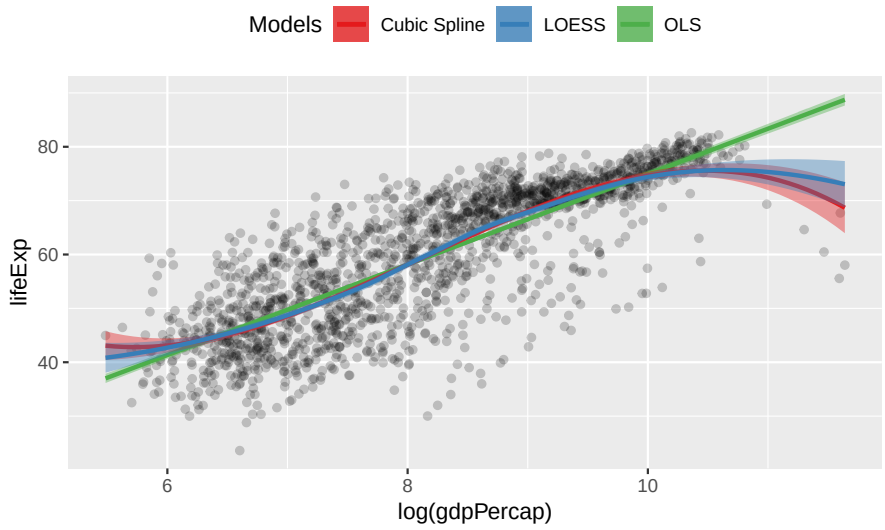
As it turns out, `geom_smooth()` can do this via the slightly unusual route of mapping the color and fill aesthetics to a string describing the model we are fitting, and then using `scale_color_manual()` and `scale_fill_manual()` to create the legend. First we use `brewer.pal()` from the `RColorBrewer` library to extract three qualitatively different colors from a larger palette. The colors are represented as hex values. As before use the `::` convention to use the function without loading the whole library:

```
model_colors <- RColorBrewer::brewer.pal(3, "Set1")
model_colors
```

```
#> [1] "#E41A1C" "#377EB8" "#4DAF4A"
```

Then we create a plot with three different smoothers, mapping the color and fill within the `aes()` function as the name of the smoother:

```
p0 <- ggplot(data = gapminder,
             mapping = aes(x = log(gdpPercap), y = lifeExp))

p1 <- p0 + geom_point(alpha = 0.2) +
    geom_smooth(method = "lm", aes(color = "OLS", fill = "OLS")) +
    geom_smooth(method = "lm", formula = y ~ splines::bs(x, df = 3),
                aes(color = "Cubic Spline", fill = "Cubic Spline")) +
    geom_smooth(method = "loess",
                aes(color = "LOESS", fill = "LOESS"))


p1 + scale_color_manual(name = "Models", values = model_colors) +
    scale_fill_manual(name = "Models", values = model_colors) +
    theme(legend.position = "top")
```



In a way we have cheated a little here to make the plot work. Until now, we have always mapped aesthetics to the names of variables, not to strings like "OLS" or "Cubic Splines". In Chapter 3, when we discussed mapping versus setting aesthetics, we saw what happened when we tried to change the color of the points in a scatterplot by setting them to "purple" inside the `aes()`function. The result was that the points turned red instead, as `ggplot` in effect created a new variable and labeled it with the word "purple". We learned there that the `aes()` function was for mapping variables to aesthetics.

Here we take advantage of that behavior, creating a new single-value variable for the name of each of our models. Ggplot will properly construct the relevant guide if we call `scale_color_manual()` and `scale_fill_manual()`. Remember that we have to call two scale functions because we have two mappings. The result is a single plot containing not just our three smoothers, but also an appropriate legend to guide the reader.

These model-fitting features make `ggplot` very useful for exploratory work, and make it straightforward to generate and compare model-based trends and other summaries as part of the process of descriptive data visualization. The various `stat_` functions are a flexible way to add summary estimates of various kinds to plots. But we will also want more than this, including presenting results from models we fit ourselves.

7.3 Look inside model objects

Covering the details of fitting statistical models in R is beyond the scope of this book. For a comprehensive, modern introduction to that topic you should work your way through (Gelman & Hill, 2018). (Harrell, 2016) is also very good on the many practical connections between modeling and graphing data. Similarly, (Gelman, 2004) provides a detailed discussion of the use of graphics as a tool in model-checking and validation. Here we will discuss some ways to take the models that you fit and extract information that is easy to work with in `ggplot`. Our goal, as always, is to get from however the object is stored to a tidy table of numbers that we can plot. Most classes of statistical model in R will contain the information we need, or will have a special set of functions, or methods, designed to extract it.

We can start by learning a little more about how the output of models is stored in R. Remember, we are always working with objects, and objects have an internal structure consisting of named pieces. Sometimes these are single numbers, sometimes vectors, and sometimes lists of things like vectors, matrices, or formulas.

We have been working extensively with tibbles and data frames. These store tables of data with named columns, perhaps consisting of different classes of variable, such as integers, characters, dates, or factors. Model objects are a little more complicated again.

```
gapminder
#> # A tibble: 1,704 x 6
#>   country     continent  year lifeExp      pop gdpPercap
#>   <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
#> 1 Afghanistan Asia       1952    28.8  8425333      779.
#> 2 Afghanistan Asia       1957    30.3  9240934      821.
#> 3 Afghanistan Asia       1962    32.0 10267083      853.
#> 4 Afghanistan Asia       1967    34.0 11537966      836.
#> 5 Afghanistan Asia       1972    36.1 13079460      740.
#> 6 Afghanistan Asia       1977    38.4 14880372      786.
#> # ... with 1,698 more rows
```

Remember, we can use the `str()` function to learn more about the internal structure of any object. For example, we can get some information on what class (or classes) of object `gapminder` is, how large it is, and what components it has. The output from `str(gapminder)` is somewhat dense:

```
str(gapminder)
#> Classes 'tbl_df', 'tbl' and 'data.frame':    1704 obs. of  6 variables:
#>  $ country  : Factor w/ 142 levels "Afghanistan",..: 1 1 1 1 1 1 1 1 1 1 ...
#>  $ continent: Factor w/ 5 levels "Africa","Americas",..: 3 3 3 3 3 3 3 3 3 3 ...
#>  $ year     : int  1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
#>  $ lifeExp  : num  28.8 30.3 32 34 36.1 ...
#>  $ pop      : int  8425333 9240934 10267083 11537966 13079460 14880372 12881816 13867957 16317921 22
#>  $ gdpPercap: num  779 821 853 836 740 ...
```

There is a lot of information here about the object as a whole and each variable in it. In the same way, statistical models in R have an internal structure. But because models are more complex entities than data tables, their structure is correspondingly more complicated. There are more pieces of information, and more kinds of information, that we might want to use. All of this information is generally stored in or is computable from parts of a model object.

We can create a linear model, an ordinary OLS regression, using the `gapminder` data. This dataset has a country-year structure that makes an OLS specification like this the wrong one to use. But never mind that for now. We use the `lm()` function to run the model, and store it in an object called `out`:

```
out <- lm(formula = lifeExp ~ gdpPercap + pop + continent,
          data = gapminder)
```

The first argument is the formula for the `model. lifeExp` is the dependent variable and the tilde `~` operator is used to designate the left- and right-hand sides of a model (including in cases, as we saw with `facet_wrap()` where the model just has a right-hand side.)

Let's look at the results by asking R to print a summary of the model.

```
summary(out)
#>
#> Call:
#> lm(formula = lifeExp ~ gdpPercap + pop + continent, data = gapminder)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -49.16  -4.49   0.30   5.11  25.17
#>
#> Coefficients:
#>                   Estimate Std. Error t value Pr(>|t|)
#> (Intercept)       4.78e+01   3.40e-01  140.82   <2e-16 ***
#> gdpPercap         4.50e-04   2.35e-05   19.16   <2e-16 ***
#> pop               6.57e-09   1.98e-09    3.33    9e-04 ***
#> continentAmericas 1.35e+01   6.00e-01   22.46   <2e-16 ***
#> continentAsia     8.19e+00   5.71e-01   14.34   <2e-16 ***
#> continentEurope   1.75e+01   6.25e-01   27.97   <2e-16 ***
#> continentOceania  1.81e+01   1.78e+00   10.15   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 8.37 on 1697 degrees of freedom
#> Multiple R-squared:  0.582,  Adjusted R-squared:  0.581
#> F-statistic:  394 on 6 and 1697 DF,  p-value: <2e-16
```

When we use the `summary()` function on out, we are not getting a simple feed of what's in the model object. Instead, like any function, `summary()` takes its input, performs some actions, and produces output. In this case, what is printed to the console is partly information that is stored inside the model object, and partly information that the `summary()` function has calculated and formated for display on the screen. Behind the scenes, `summary()` gets help from other functions. Objects of different classes have default methods associated with them, so that when the generic `summary()` function is applied to a linear model object, the function knows to pass the work on to a more specialized function that does a bunch of calculations and formatting appropriate to a linear model object. We use the same generic `summary()` function on data frames, as in `summary(gapminder)`, but in that case a different default method is applied.

Schematic view of a linear model object.Figure 6.3: Schematic view of a linear model object.

The output from `summary()` gives a precis of the model, but we can't really do any further analysis with it directly. For example, what if we want to plot something from the model? The information necessary to make plots is inside the out object, but it is not obvious how to use it.

If we take a look at the structure of the model object with `str(out)` we will find that there is a lot of information in there. Like most complex objects in R, out is organized as a list of components or elements. Several of these elements are themselves lists. Figure 6.3 gives you a schematic view of the contents of a linear model object. In this list of items, elements are single values, some are data frames, and some are additional lists of simpler items. Again, remember our earlier discussion where we said objects could be

thought of as being organized like a filing system: cabinets contain drawers, and drawer may contain which may contain pages of information, whole documents, or groups of folders with more documents inside. As an alternative analogy, and sticking with the image of a list, you can think of a master to-do list for a project, where the top-level headings lead to contain additional lists of tasks of different kinds.

The `out` object created by `lm` contains several different named elements. Some, like the residual degrees of freedom in the model, are just a single number. Try `out$df.residual` at the console. Others are much larger entities, such as the data frame used to fit the model, which is retained by default. Try `out$model`, but be prepared for a lot of stuff to be printed at the console. Other elements have been computed by R and then stored, such as the coefficients of the model and other quantities. You can try `out$coefficients`, `out$residuals`, and `out$fitted.values`, for instance. Others are lists themselves (like `qr`). So you can see that the `summary()` function is selecting and printing only a small amount of core information, in comparison to what is stored in the model object.

Just like the tables of data we saw earlier in Section A.1.3, the output of `summary()` is presented in a way that is compact and efficient in terms of getting information across, but also untidy when considered from the point of view of further manipulation. There is a table of coefficients, but the variable names are in the rows. The column names are awkward, and some information (e.g. at the bottom of the output) has been calculated and printed out, but is not stored in the model object.

7.4 Get model-based graphics right

Figures based on statistical models face all the ordinary challenges of effective data visualization, and then some. This is because model results usually carry a considerable extra burden of interpretation and necessary background knowledge. The more complex the model, the trickier it becomes to convey this information effectively, and the easier it becomes to lead one's audience or oneself into error. Within the social sciences, our ability to clearly and honestly present model-based graphics has greatly improved over the past ten or fifteen years. Over the same period, it has become clearer that some kinds of models are quite tricky to understand, even ones that had previously been seen as straightforward elements of the modeling toolkit (Ai & Norton, 2003; Brambor, Clark, & Golder, 2006).

Plotting model estimates is closely connected to properly estimating models in the first place. This means there is no substitute for learning the statistics. You should not use graphical methods as a substitute for understanding the model used to produce them. While this book cannot teach you that material, we can make a few general points about what good model-based graphics look like, and work through some examples of how `ggplot` and some additional libraries can make it easier to get good results.

7.4.1 Present your findings in substantive terms

Useful model-based plots show results in ways that are substantively meaningful and directly interpretable with respect to the questions the analysis is trying to answer. This means showing results in a context where other variables in the analysis are held at sensible values, such as their means or medians. With continuous variables, it can often be useful to generate predicted values that cover some substantively meaningful move across the distribution, such as from the **25th to the 75th percentile**, rather than a single-unit increment in the variable of interest. For unordered categorical variables, predicted values might be presented with respect to the modal category in the data, or for a particular category of theoretical interest. Presenting substantively interpretable findings often also means using (and sometimes converting to) a scale that readers can easily understand. If your model reports results in log-odds, for example, converting the estimates to predicted probabilities will make it easier to interpret. All of this advice is quite general. Each of these points applies equally well to the presentation of summary results in a table rather than a graph. There is nothing distinctively graphical about putting the focus on the substantive meaning of your findings.

7.4.2 Show your degree of confidence

Much the same applies to presenting the degree of uncertainty or confidence you have in your results. Model estimates come with various measures of precision, confidence, credence, or significance. Presenting and interpreting these measures is notoriously prone to misinterpretation, or over-interpretation, as researchers and audiences both demand more from things like confidence intervals and p-values than these statistics can deliver. At a minimum, having decided on an appropriate measure of model fit or the right assessment of confidence, you should show their range when you present your results. A family of related `ggplot` geoms allow you to show a range or interval defined by position on the x-axis and then a `ymin` and `ymax` range on the y-axis. These geoms include `geom_pointrange()` and `geom_errorbar()`, which we will see in action shortly. A related geom, `geom_ribbon()` uses the same arguments to draw filled areas, and is useful for plotting ranges of y-axis values along some continuously varying x-axis.

7.4.3 Show your data when you can

Plotting the results from a multivariate model generally means one of two things. First, we can show what is in effect a table of coefficients with associated measures of confidence, perhaps organizing the coefficients into meaningful groups, or by the size of the predicted association, or both. Second, we can show the predicted values of some variables (rather than just a model's coefficients) across some range of interest. The latter approach lets us show the original data points if we wish. The way `ggplot` builds graphics layer by layer allows us to easily combine model estimates (e.g. a regression line and an associated range) and the underlying data. In effect these are manually-constructed versions of the automatically-generated plots that we have been producing with `geom_smooth()` since the beginning of this book.

7.5 Generate predictions to graph

Having fitted a model, then, we might want to get a picture of the estimates it produces over the range of some particular variable, holding other covariates constant at some sensible values. The `predict()` function is a generic way of using model objects to produce this kind of prediction. In R, "generic" functions take their inputs and pass them along to more specific functions behind the scenes, ones that are suited to working with the particular kind of model object we have. The details of getting predicted values from a **OLS** model, for instance, will be somewhat different from getting predictions out of a logistic regression. But in each case we can use the same `predict()` function, taking care to check the documentation to see what form the results are returned in for the kind of model we are working with. Many of the most commonly-used functions in R are generic in this way. The `summary()` function, for example, works on objects of many different classes, from vectors to data frames and statistical models, producing appropriate output in each case by way of a class-specific function in the background.

For `predict()` to calculate the new values for us, it needs some new data to fit the model to. We will generate a new data frame whose columns have the same names as the variables in the model's original data, but where the rows have new values. A very useful function called `expand.grid()` will help us do this. We will give it a list of variables, specifying the range of values we want each variable to take. Then `expand.grid()` will generate the will multiply out the full range of values for all combinations of the values we give it, thus creating a new data frame with the new data we need.

In the following bit of code, we use `min()` and `max()` to get the minimum and maximum values for per capita GDP, and then create a vector with one hundred evenly-spaced elements between the minimum and the maximum. We hold population constant at its median, and we let continent take all of its five available values.

```
min_gdp <- min(gapminder$gdpPercap)
max_gdp <- max(gapminder$gdpPercap)
med_pop <- median(gapminder$pop)
```

```
pred_df <- expand.grid(gdpPercap = (seq(from = min_gdp,
                                        to = max_gdp,
                                        length.out = 100)),
                       pop = med_pop,
                       continent = c("Africa", "Americas",
                                     "Asia", "Europe", "Oceania"))

dim(pred_df)
#> [1] 500   3
```

```
head(pred_df)
#>   gdpPercap     pop continent
#> 1       241 7023596    Africa
#> 2      1385 7023596    Africa
#> 3      2530 7023596    Africa
#> 4      3674 7023596    Africa
#> 5      4818 7023596    Africa
#> 6      5962 7023596    Africa
```

Now we can use `predict()`. If we give the function our new data and model, without any further argument, it will calculate the fitted values for every row in the data frame. If we specify `interval = 'predict'` as an argument, it will calculate 95% prediction intervals in addition to the point estimate.

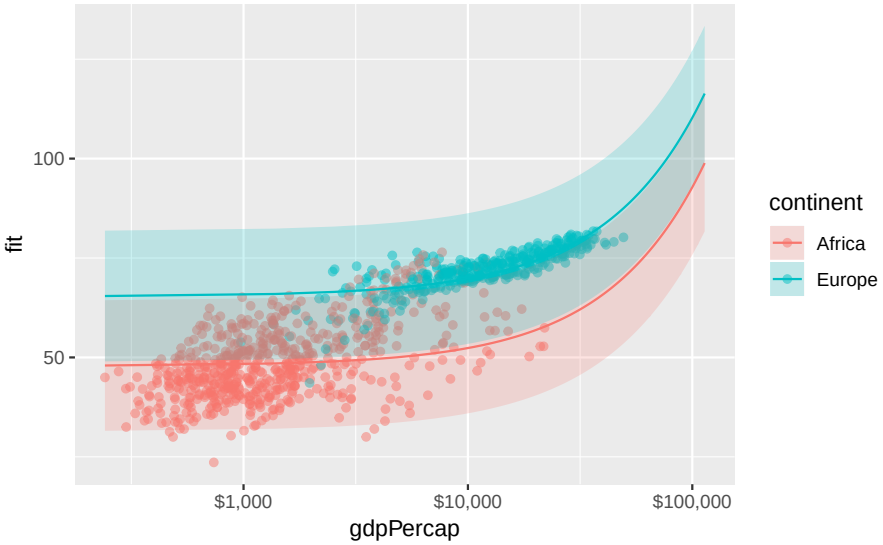
```
pred_out <- predict(object = out,
                    newdata = pred_df,
                    interval = "predict")
head(pred_out)
#>    fit  lwr  upr
#> 1 48.0 31.5 64.4
#> 2 48.5 32.1 64.9
#> 3 49.0 32.6 65.4
#> 4 49.5 33.1 65.9
#> 5 50.0 33.6 66.4
#> 6 50.5 34.1 67.0
```

Because we know that, by construction, the cases in `pred_df` and `pred_out` correspond row for row, we can bind the two data frames together by column. This method of joining or merging tables is definitely not recommended when you are dealing with data.

```
pred_df <- cbind(pred_df, pred_out)
head(pred_df)
#>   gdpPercap     pop continent  fit  lwr  upr
#> 1       241 7023596    Africa 48.0 31.5 64.4
#> 2      1385 7023596    Africa 48.5 32.1 64.9
#> 3      2530 7023596    Africa 49.0 32.6 65.4
#> 4      3674 7023596    Africa 49.5 33.1 65.9
#> 5      4818 7023596    Africa 50.0 33.6 66.4
#> 6      5962 7023596    Africa 50.5 34.1 67.0
```

The end result is a tidy data frame, containing the predicted values from the model for the range of values we specified. Now we can plot the results. Because we produced a full range of predicted values, we can decide whether or not to use all of them. Here we further subset the predictions to just those for Europe and Africa.

```
p <- ggplot(data = subset(pred_df, continent %in% c("Europe", "Africa")),
            aes(x = gdpPercap,
                y = fit, ymin = lwr, ymax = upr,
                color = continent,
                fill = continent,
                group = continent))

p + geom_point(data = subset(gapminder,
                             continent %in% c("Europe", "Africa")),
               aes(x = gdpPercap, y = lifeExp,
                   color = continent),
               alpha = 0.5,
               inherit.aes = FALSE) +
    geom_line() +
    geom_ribbon(alpha = 0.2, color = FALSE) +
    scale_x_log10(labels = scales::dollar)
```



We use a new geom here to draw the area covered by the prediction intervals: `geom_ribbon()`. It takes an x argument like a line, but a ymin and ymax argument as specified in the `ggplot()` aesthetic mapping. This defines the lower and upper limits of the prediction interval.

In practice, you may not use `predict()` directly all that often. Instead, you might write code using additional libraries that encapsulate the process of producing predictions and plots from models. These are especially useful when your model is a little more complex and the interpretation of coefficients becomes trickier. This happens, for instance, when you have a binary outcome variable and need to convert the results of a logistic regression into predicted probabilities, or when you have interaction terms amongst your predictions. We will discuss some of these helper libraries in the next few sections. However, bear in mind that `predict()` and its ability to work safely with different classes of model underpins many of those libraries. So it's useful to see it in action first hand in order to understand what it is doing.

7.6 Tidy model objects with `broom`

The `predict` method is very useful, but there are a lot of other things we might want to do with our model output. We will use David Robinson's `broom` package to help us out. It is a library of functions that help us get from the model results that R generates to numbers that we can plot. It will take model objects and

turn pieces of them into data frames that you can use easily with ggplot.

```
library(broom)
```

Broom takes ggplot's approach to tidy data and extends it to the model objects that R produces. Its methods can tidily extract three kinds of information. First, we can see component-level information about aspects of the model itself, such as coefficients and t-statistics. Second, we can obtain observation-level information about the model's connection to the underlying data. This includes the fitted values and residuals for each observation in the data. And finally we can get model-level information that summarizes the fit as a whole, such as an F-statistic, the model deviance, or the r-squared. There is a broom function for each of these tasks.

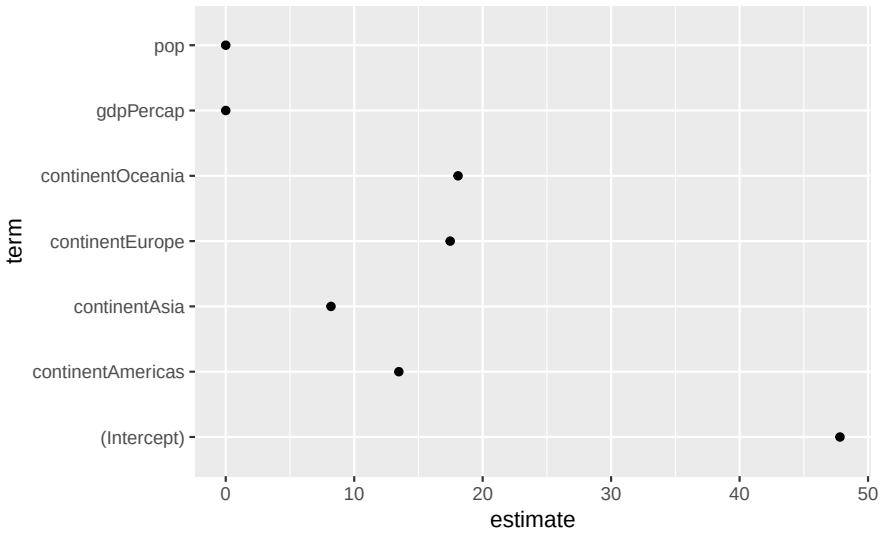
7.6.1 Get component-level statistics with tidy()

The tidy() function takes a model object and returns a data frame of component-level information. We can work with this to make plots in a familiar way, and much more easily than fishing inside the model object to extract the various terms. Here is an example, using the default results as just returned. For a more convenient display of the results, we will pipe the object we create with tidy() through a function that rounds the numeric columns of the data frame to two decimal places. This doesn't change anything about the object itself, of course.

```
out_comp <- tidy(out)
out_comp %>% round_df()
#> # A tibble: 7 x 5
#>   term              estimate std.error statistic p.value
#>   <chr>                <dbl>     <dbl>     <dbl>   <dbl>
#> 1 (Intercept)          47.8      0.34     141.          0
#> 2 gdpPercap             0        0         19.2         0
#> 3 pop                   0        0          3.33        0
#> 4 continentAmericas    13.5      0.6       22.5         0
#> 5 continentAsia         8.19     0.570     14.3         0
#> 6 continentEurope      17.5      0.62      28.0         0
#> # ... with 1 more row
```

We are now able to treat this dataframe just like all the other data that we have seen so far.

```
p <- ggplot(out_comp, mapping = aes(x = term,
                                    y = estimate))

p + geom_point() + coord_flip()
```



We can extend and clean up this plot in a variety of ways. For example, we can tell `tidy()` to calculate confidence intervals for the estimates, using R's `confint()` function.

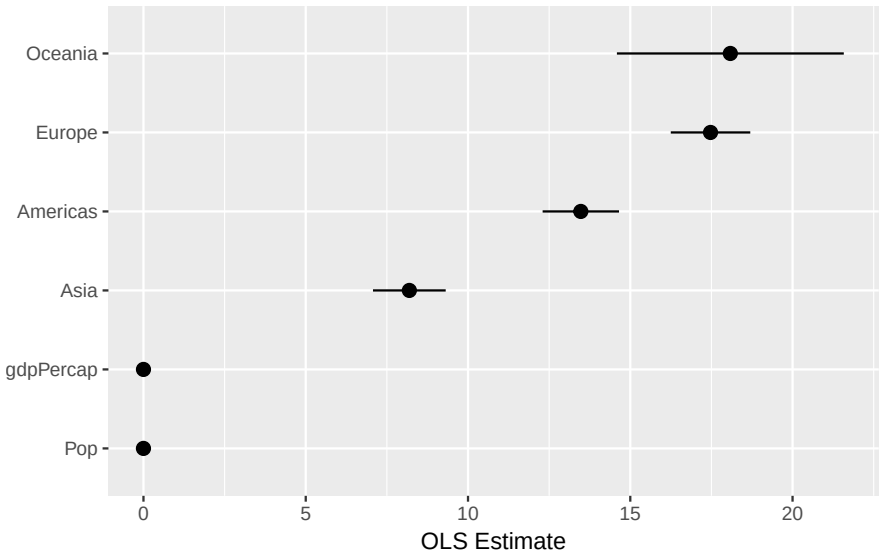
```
out_conf <- tidy(out, conf.int = TRUE)
out_conf %>% round_df()
#> # A tibble: 7 x 7
#>   term              estimate std.error statistic p.value conf.low conf.high
#>   <chr>                <dbl>     <dbl>     <dbl>   <dbl>    <dbl>     <dbl>
#> 1 (Intercept)          47.8      0.34     141.          0    47.2      48.5
#> 2 gdpPercap             0         0        19.2         0     0         0
#> 3 pop                   0         0         3.33        0     0         0
#> 4 continentAmericas    13.5       0.6      22.5         0    12.3      14.6
#> 5 continentAsia         8.19      0.570    14.3         0     7.07      9.31
#> 6 continentEurope      17.5       0.62     28.0         0    16.2      18.7
#> # ... with 1 more row
```

The convenience "not in" operator `%nin%` is available via the `socviz` library. It does the opposite of `%in%` and selects only the items in a first vector of characters that are not in the second. We'll use it to drop the intercept term from the table. We also want to something about the labels. When fitting a model with categorical variables, R will create coefficient names based on the variable name and the category name, like continentAmericas. Normally we like to clean these up before plotting. Most commonly, we just want to strip away the variable name at the beginning of the coefficient label. For this we can use `prefix_strip()`, a convenience function in the `socviz` library. We tell it which prefixes to drop, using it to create a new column variable in `out_conf` that corresponds to the terms column, but that has nicer labels.

```
out_conf <- subset(out_conf, term %nin% "(Intercept)")
out_conf$nicelabs <- prefix_strip(out_conf$term, "continent")
```

Now we can use `geom_pointrange()`to make a figure that displays some information about our confidence in the variable estimates, as opposed to just the coefficients. As with the boxplots earlier, we use `reorder()` to sort the names of the model's terms by the estimate variable, thus arranging our plot of effects from largest to smallest in magnitude.

```
p <- ggplot(out_conf, mapping = aes(x = reorder(nicelabs, estimate),
                                    y = estimate, ymin = conf.low, ymax = conf.high))
p + geom_pointrange() + coord_flip() + labs(x="", y="OLS Estimate")
```



Dotplots of this kind can be very compact. The vertical axis can often be compressed quite a bit, with no loss in comprehension. In fact, they are often easier to read with much less room between the rows than given by a default square shape.

7.6.2 Get observation-level statistics with `augment()`

The values returned by `augment()` are all statistics calculated at the level of the original observations. As such, they can be added on to the data frame that the model is based on. Working from a call to `augment()` will return a data frame with all the original observations used in the estimation of the model, together with columns like the following:

- **.fitted** — The fitted values of the model.
- **.se.fit** — The standard errors of the fitted values.
- **.resid** — The residuals.
- **.hat** — The diagonal of the hat matrix.
- **.sigma** — An estimate of residual standard deviation when the corresponding observation is dropped from the model.
- **.cooksd** — Cook's distance, a common regression diagnostic; and
- **.std.resid** — The standardized residuals.

Each of these variables is named with a leading `dot`, for example `.hat` rather than hat, and so on. This is to guard against accidentally confusing it with (or accidentally overwriting) an existing variable in your data with this name. The columns of values return will differ slightly depending on the class of model being fitted.

```
out_aug <- augment(out)
head(out_aug) %>% round_df()
#> # A tibble: 6 x 11
#>   lifeExp gdpPercap    pop continent .fitted .se.fit .resid  .hat .sigma
#>     <dbl>     <dbl>  <dbl> <fct>       <dbl>   <dbl>  <dbl> <dbl>  <dbl>
#> 1    28.8      779. 8.43e6 Asia         56.4    0.47  -27.6     0   8.34
#> 2    30.3      821. 9.24e6 Asia         56.4    0.47  -26.1     0   8.34
#> 3    32        853. 1.03e7 Asia         56.5    0.47  -24.5     0   8.35
#> 4    34.0      836. 1.15e7 Asia         56.5    0.47  -22.4     0   8.35
#> 5    36.1      740. 1.31e7 Asia         56.4    0.47  -20.3     0   8.35
#> 6    38.4      786. 1.49e7 Asia         56.5    0.47  -18.0     0   8.36
```

```
#> # ... with 2 more variables: .cooksd <dbl>, .std.resid <dbl>
```

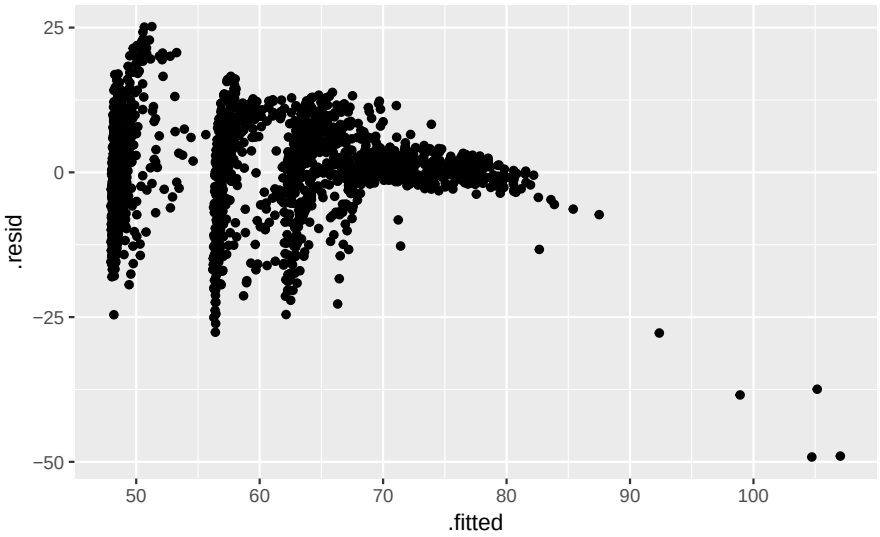
By default, `augment()` will extract the available data from the model object. This will usually include the variables used in the model itself, but not any additional ones contained in the original data frame. Sometimes it is useful to have these. We can add them by specifying the `data` argument:

```
out_aug <- augment(out, data = gapminder)
head(out_aug) %>% round_df()
#> # A tibble: 6 x 13
#>   country continent  year lifeExp    pop gdpPercap .fitted .se.fit .resid
#>   <fct>   <fct>     <dbl>   <dbl>  <dbl>     <dbl>   <dbl>   <dbl>  <dbl>
#> 1 Afghan~ Asia       1952    28.8 8.43e6      779.    56.4    0.47  -27.6
#> 2 Afghan~ Asia       1957    30.3 9.24e6      821.    56.4    0.47  -26.1
#> 3 Afghan~ Asia       1962    32   1.03e7      853.    56.5    0.47  -24.5
#> 4 Afghan~ Asia       1967    34.0 1.15e7      836.    56.5    0.47  -22.4
#> 5 Afghan~ Asia       1972    36.1 1.31e7      740.    56.4    0.47  -20.3
#> 6 Afghan~ Asia       1977    38.4 1.49e7      786.    56.5    0.47  -18.0
#> # ... with 4 more variables: .hat <dbl>, .sigma <dbl>, .cooksd <dbl>,
#> #   .std.resid <dbl>
```

If some rows containing missing data were dropped to fit the model, then these will not be carried over to the augmented dataframe.

The new columns created by `augment()` can be used to create some standard regression plots. For example, we can plot the residuals versus the fitted values. Figure 6.7 suggests, unsurprisingly, that our country-year data has rather more structure than is captured by our `OLS` model.

```
p <- ggplot(data = out_aug,
            mapping = aes(x = .fitted, y = .resid))
p + geom_point()
```



7.6.3 Get model-level statistics with `glance()`

This function organizes the information typically presented at the bottom of a model's `summary()` output. By itself, it usually just returns a table with a single row in it. But as we shall see in a moment, the real

power of broom's approach is the way that it can scale up to cases where we are grouping or subsampling our data.

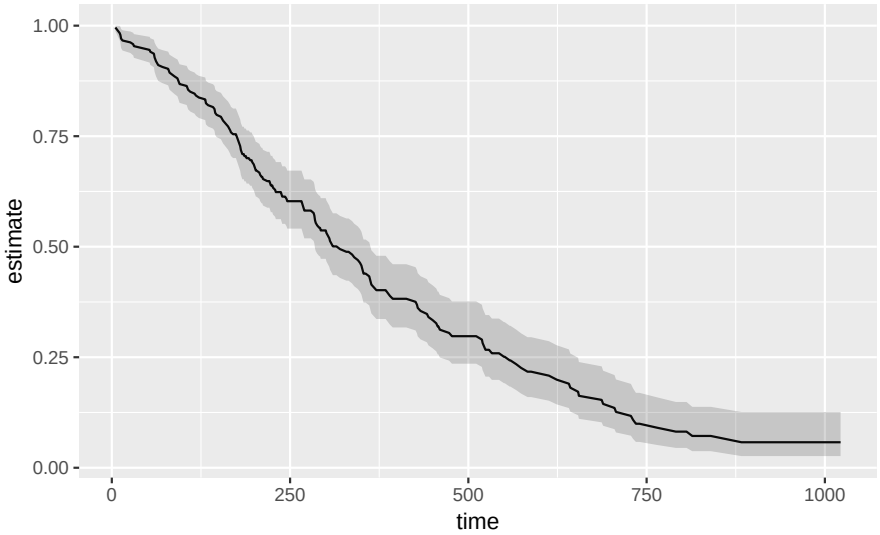
```
glance(out) %>% round_df()
#> # A tibble: 1 x 11
#>   r.squared adj.r.squared sigma statistic p.value    df logLik    AIC
#>       <dbl>         <dbl> <dbl>     <dbl>   <dbl> <dbl>  <dbl>  <dbl>
#> 1     0.580         0.580  8.37      394.       0     7 -6034. 12084.
#> # ... with 3 more variables: BIC <dbl>, deviance <dbl>, df.residual <dbl>
```

Broom is able to `tidy` (and `augment`, and `glance` at) a wide range of model types. Not all functions are available for all classes of model. Consult broom's documentation for more details on what is available. For example, here is a plot created from the tidied output of an event-history analysis. First we generate a Cox proportional hazards model of some `survival` data.

```
library(survival)
#>
#> Attaching package: 'survival'
#> The following object is masked from 'package:quantreg':
#>
#>     untangle.specials

out_cph <- coxph(Surv(time, status) ~ age + sex, data = lung)
out_surv <- survfit(out_cph)
```

The details of the fit are not important here, but in the first step the `Surv()` function creates the response or outcome variable for the proportional hazards model that is then fitted by the `coxph()`function. Then the `survfit()` function creates the survival curve from the model, much like we used `predict()` to generate predicted values earlier. Try `summary(out_cph)` to see the model, and `summary(out_surv)` to see the table of predicted values that will form the basis for our plot. Next we tidy `out_surv` to get a data frame, and plot it.

```
# Figure 6.8: A Kaplan-Meier plot.
out_tidy <- tidy(out_surv)

p <- ggplot(data = out_tidy, mapping = aes(time, estimate))
p + geom_line() +
    geom_ribbon(mapping = aes(ymin = conf.low, ymax = conf.high), alpha = .2)
```



7.7 Grouped analysis and list-columns

Broom makes it possible to quickly fit models to different subsets of your data and get consistent and usable tables of results out the other end. For example, let's say we wanted to look at the gapminder data by examining the relationship between life expectancy and GDP by continent, for each year in the data.

The `gapminder` data is at bottom organized by country-years. That is the unit of observation in the rows. If we wanted, we could take a slice of the data manually, such as "all countries observed in Asia, in 1962" or "all in Africa, 2002". Here is "Europe, 1977":

```
eu77 <- gapminder %>% filter(continent == "Europe", year == 1977)
```

We could then see what the relationship between life expectancy and GDP looked like for that continent-year group:

```
fit <- lm(lifeExp ~ log(gdpPercap), data = eu77)
summary(fit)
#>
#> Call:
#> lm(formula = lifeExp ~ log(gdpPercap), data = eu77)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -7.496 -1.031  0.093  1.176  3.712
#>
#> Coefficients:
#>                Estimate Std. Error t value Pr(>|t|)
#> (Intercept)      29.489      7.161    4.12  0.00031 ***
#> log(gdpPercap)    4.488      0.756    5.94  2.2e-06 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 2.11 on 28 degrees of freedom
#> Multiple R-squared:  0.557,  Adjusted R-squared:  0.541
#> F-statistic: 35.2 on 1 and 28 DF,  p-value: 2.17e-06
```

With `dplyr` and `broom` we can do this for every continent-year slice of the data in a compact and tidy way. We start with our table of data, and then (`%>%`) group the countries by continent and year using the `group_by()` function. We introduced this grouping operation in Chapter 4. Our data is reorganized first by continent, and within continent by year. Here we will take one further step and nest the data that make up each group:

```
out_le <- gapminder %>%
    group_by(continent, year) %>%
    nest()

out_le
#> # A tibble: 60 x 3
#>   continent  year data
#>   <fct>     <int> <list>
#> 1 Asia       1952 <tibble [33 x 4]>
#> 2 Asia       1957 <tibble [33 x 4]>
#> 3 Asia       1962 <tibble [33 x 4]>
#> 4 Asia       1967 <tibble [33 x 4]>
```

```
#> 5 Asia       1972 <tibble [33 x 4]>
#> 6 Asia       1977 <tibble [33 x 4]>
#> # ... with 54 more rows
```

Think of what `nest()` does as a more intensive version what `group_by()` does. The resulting object is has the tabular form we expect (it is a tibble) but it looks a little unusual. The first two columns are the familiar continent and year. But we now also have a new column, data, that contains a small table of data corresponding to each continent-year group. This is a list-column, something we have not seen before. It turns out to be very useful for bundling together complex objects (structured, in this case, as a list of tibbles, each being a 33x4 table of data) within the rows of our data (which remains tabular). Our "Europe 1977" fit is in there. We can look at it, if we like, by filtering the data and then unnesting the list column.

```
out_le %>% filter(continent == "Europe" & year == 1977) %>% unnest()
#> # A tibble: 30 x 6
#>   continent  year country                lifeExp     pop gdpPercap
#>   <fct>     <int> <fct>                    <dbl>   <int>     <dbl>
#> 1 Europe     1977 Albania                   68.9 2509048     3533.
#> 2 Europe     1977 Austria                   72.2 7568430    19749.
#> 3 Europe     1977 Belgium                   72.8 9821800    19118.
#> 4 Europe     1977 Bosnia and Herzegovina    69.9 4086000     3528.
#> 5 Europe     1977 Bulgaria                  70.8 8797022     7612.
#> 6 Europe     1977 Croatia                   70.6 4318673    11305.
#> # ... with 24 more rows
```

List-columns are useful because we can act on them in a compact and tidy way. In particular, we can pass functions along to each row of the list-column and make something happen. For example, a moment ago we ran a regression of life expectancy and logged GDP for European countries in 1977. We can do that for every continent-year combination in the data. We first create a convenience function called `fit_ols()` that takes a single argument, `df` (for data frame) and that fits the linear model we are interested in. Then we map that function to each of our list-column rows in turn. Recall from Chapter 4 that mutate creates new variables or columns on the fly within a pipeline.

The `map` action is an important idea in functional programming. If you have written code in other, more imperative languages you can think of it as a compact alternative to writing `for … next` loops. You can of course write loops like this in R. Computationally they are often not any less efficient than their functional alternatives. But mapping functions to arrays is more easily integrated into a sequence of data transformations.

```
fit_ols <- function(df) {
    lm(lifeExp ~ log(gdpPercap), data = df)
}

out_le <- gapminder %>%
    group_by(continent, year) %>%
    nest() %>%
    mutate(model = map(data, fit_ols))

out_le
#> # A tibble: 60 x 4
#>   continent  year data              model
#>   <fct>     <int> <list>            <list>
#> 1 Asia       1952 <tibble [33 x 4]> <lm>
#> 2 Asia       1957 <tibble [33 x 4]> <lm>
#> 3 Asia       1962 <tibble [33 x 4]> <lm>
#> 4 Asia       1967 <tibble [33 x 4]> <lm>
```

```
#> 5 Asia       1972 <tibble [33 x 4]> <lm>
#> 6 Asia       1977 <tibble [33 x 4]> <lm>
#> # ... with 54 more rows
```

Before starting the pipeline we create a new function: It is a convenience function whose only job is to estimate a particular OLS model on some data. Like almost everything in R, functions are a kind of object. To make a new one, we use the slightly special `function()` function. (Nerds love that sort of thing.) There is a little more detail on creating functions in the Appendix. To see what fit_ols() looks like once it is created, type `fit_ols` without parentheses at the Console. To see what it does, try `fit_ols(df = gapminder)`, or `summary(fit_ols(gapminder))`.

Now we have two list-columns: `data`, and `model.` The latter was created by mapping the `fit_ols()` function to each row of data. Inside each element of model is a linear model for that continent-year. So we now have sixty OLS fits, one for every continent-year grouping. Having the models inside the list column is not much use to us in and of itself. But we can extract the information we want while keeping things in a tidy tabular form. For clarity we will run the pipeline from the beginning again, this time adding a few new steps.

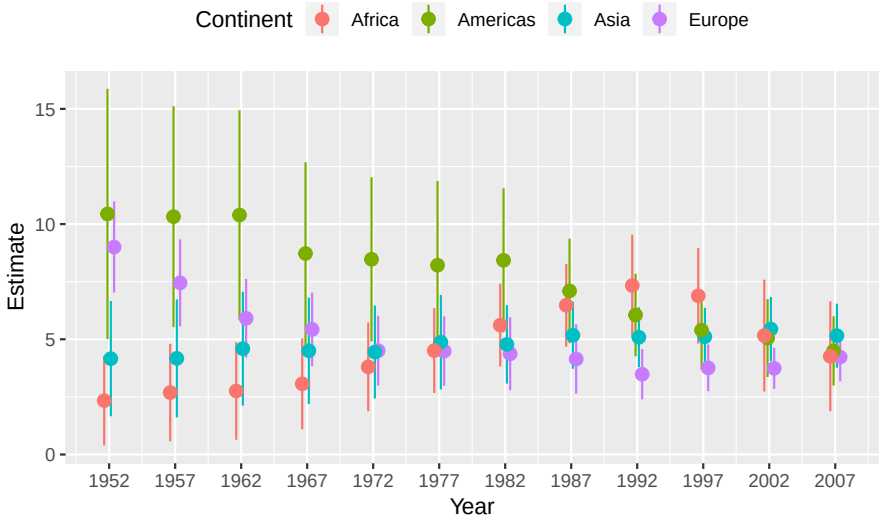
First we extract summary statistics from each model by mapping the `tidy()` function from `broom` to the model list column. Then we unnest the result, dropping the other columns in the process. Finally, we filter out all the Intercept terms, and also drop all observations from Oceania. In the case of the Intercepts we do this just out of convenience. Oceania we drop just because there are so few observations. We put the results in an object called `out_tidy`.

```
fit_ols <- function(df) {
    lm(lifeExp ~ log(gdpPercap), data = df)
}

out_tidy <- gapminder %>%
    group_by(continent, year) %>%
    nest() %>%
    mutate(model = map(data, fit_ols),
           tidied = map(model, tidy)) %>%
    unnest(tidied, .drop = TRUE) %>%
    filter(term %nin% "(Intercept)" &
           continent %nin% "Oceania")

out_tidy %>% sample_n(5)
#> # A tibble: 5 x 7
#>   continent  year term           estimate std.error statistic       p.value
#>   <fct>     <int> <chr>             <dbl>     <dbl>     <dbl>         <dbl>
#> 1 Americas   1992 log(gdpPercap)     6.06     0.895      6.77 0.000000664
#> 2 Europe     2002 log(gdpPercap)     3.74     0.445      8.40 0.00000000391
#> 3 Asia       2007 log(gdpPercap)     5.16     0.694      7.43 0.0000000226
#> 4 Americas   1952 log(gdpPercap)    10.4      2.72       3.84 0.000827
#> 5 Americas   1957 log(gdpPercap)    10.3      2.40       4.31 0.000261
```

We now have tidy regression output with an estimate of the association between `log` GDP per capita and life expectancy for each year, within continents. We can plot these estimates in a way that takes advantage of their groupiness.

```
# Figure 6.9: Yearly estimates of the association between GDP and Life Expectancy, pooled by continent.
p <- ggplot(data = out_tidy,
            mapping = aes(x = year, y = estimate,
                          ymin = estimate - 2*std.error,
```

```
                        ymax = estimate + 2*std.error,
                        group = continent, color = continent))

p + geom_pointrange(position = position_dodge(width = 1)) +
    scale_x_continuous(breaks = unique(gapminder$year)) +
    theme(legend.position = "top") +
    labs(x = "Year", y = "Estimate", color = "Continent")
```



The call to `position_dodge()` within `geom_pointrange()` allows the point ranges for each continent to be near each other within years, instead of being plotted right on top of one another. We could have faceted the results by continent, but doing it this way lets us see differences in the yearly estimates much more easily. This technique is very useful not just for cases like this, but also when you want to compare the coefficients given by different kinds of statistical model. This sometimes happens when we're interested in seeing how, say, OLS performs against some other model specification.

7.8 Plot marginal effects

Our earlier discussion of `predict()` was about obtaining estimates of the average effect of some coefficient, net of the other terms in the model. Over the past decade, estimating and plotting partial or marginal effects from a model has become an increasingly common way of presenting accurate and interpretively useful predictions. Interest in marginal effects plots was stimulated by the realization that the interpretation of terms in logistic regression models, in particular, was trickier than it seemed—especially when there were interaction terms in the model (Ai & Norton, 2003). Thomas Leeper's `margins` package can make these plots for us.

```
library(margins)
```

To see it in action, we'll take another look at the General Social Survey data in `gss_sm`, this time focusing on the binary variable, obama.As is common with retrospective questions on elections, rather more people claim to have voted for Obama than is consistent with the vote share he received in the election. It is coded 1 if the respondent said they voted for Barack Obama in the 2012 presidential election, and 0 otherwise. In this case, mostly for convenience here, the zero code includes all other answers to the question, including those who said they voted for Mitt Romney, those who said they did not vote, those who refused to answer, and those who said they didn't know who they voted for. We will fit a logistic regression on obama, with `age`, `polviews`, `race`, and `sex` as the predictors. The `age` variable is the respondent's age in years. The `sex` variable is coded as "Male" or "Female" with "Male" as the reference category. The `race` variable is coded

as "White", "Black", or "Other" with "White" as the reference category. The polviews measure is a self-reported scale of the respondent's political orientation from "Extremely Conservative" through "Extremely Liberal", with "Moderate" in the middle. We take `polviews` and create a new variable, `polviews_m`, using the `relevel()` function to recode "Moderate" to be the reference category. We fit the model with the `glm()` function, and specify an interaction between race and sex.

```
gss_sm$polviews_m <- relevel(gss_sm$polviews, ref = "Moderate")

out_bo <- glm(obama ~ polviews_m + sex*race,
              family = "binomial", data = gss_sm)
summary(out_bo)
#>
#> Call:
#> glm(formula = obama ~ polviews_m + sex * race, family = "binomial",
#>     data = gss_sm)
#>
#> Deviance Residuals:
#>    Min      1Q  Median      3Q     Max
#> -2.905  -0.554   0.177   0.542   2.244
#>
#> Coefficients:
#>                                 Estimate Std. Error z value Pr(>|z|)
#> (Intercept)                      0.29649    0.13409    2.21   0.0270 *
#> polviews_mExtremely Liberal      2.37295    0.52504    4.52  6.2e-06 ***
#> polviews_mLiberal                2.60003    0.35667    7.29  3.1e-13 ***
#> polviews_mSlightly Liberal       1.29317    0.24843    5.21  1.9e-07 ***
#> polviews_mSlightly Conservative -1.35528    0.18129   -7.48  7.7e-14 ***
#> polviews_mConservative          -2.34746    0.20038  -11.71  < 2e-16 ***
#> polviews_mExtremely Conservative -2.72738    0.38721   -7.04  1.9e-12 ***
#> sexFemale                        0.25487    0.14537    1.75   0.0796 .
#> raceBlack                        3.84953    0.50132    7.68  1.6e-14 ***
#> raceOther                       -0.00214    0.43576    0.00   0.9961
#> sexFemale:raceBlack             -0.19751    0.66007   -0.30   0.7648
#> sexFemale:raceOther              1.57483    0.58766    2.68   0.0074 **
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> (Dispersion parameter for binomial family taken to be 1)
#>
#>     Null deviance: 2247.9  on 1697  degrees of freedom
#> Residual deviance: 1345.9  on 1686  degrees of freedom
#>   (1169 observations deleted due to missingness)
#> AIC: 1370
#>
#> Number of Fisher Scoring iterations: 6
```

The summary reports the coefficients and other information. We can now graph the data in any one of several ways. Using `margins()` we calculate the marginal effects for each variable:

```
bo_m <- margins(out_bo)
summary(bo_m)
#>                          factor     AME     SE        z      p   lower
#>            polviews_mConservative -0.4119 0.0283 -14.5394 0.0000 -0.4674
#>  polviews_mExtremely Conservative -0.4538 0.0420 -10.7971 0.0000 -0.5361
#>       polviews_mExtremely Liberal  0.2681 0.0295   9.0996 0.0000  0.2103
```

```
#>                 polviews_mLiberal  0.2768 0.0229  12.0736 0.0000  0.2319
#>  polviews_mSlightly Conservative -0.2658 0.0330  -8.0596 0.0000 -0.3304
#>       polviews_mSlightly Liberal  0.1933 0.0303   6.3896 0.0000  0.1340
#>                        raceBlack  0.4032 0.0173  23.3568 0.0000  0.3694
#>                        raceOther  0.1247 0.0386   3.2297 0.0012  0.0490
#>                        sexFemale  0.0443 0.0177   2.5073 0.0122  0.0097
#>    upper
#>  -0.3564
#>  -0.3714
#>   0.3258
#>   0.3218
#>  -0.2011
#>   0.2526
#>   0.4371
#>   0.2005
#>   0.0789
```

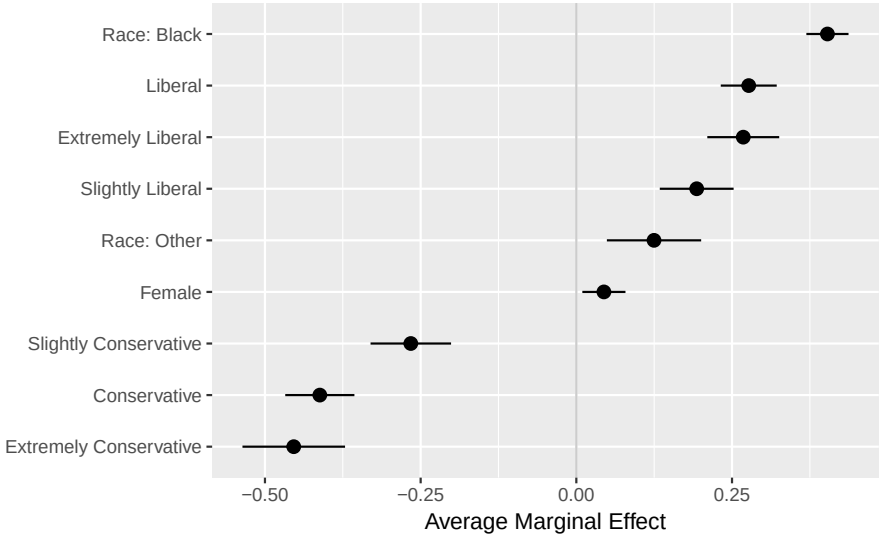
The `margins` library comes with several plot methods of its own. If you wish, at this point you can just try `plot(bo_m)` to see a plot of the average marginal effects, produced with the general look of a Stata graphic. Other plot methods in the margins library include `cplot()`, which visualizes marginal effects conditional on a second variable, and `image()`, which shows predictions or marginal effects as a filled heatmap or contour plot.

Alternatively, we can take results from `margins()` and plot them ourselves. To clean up the summary a little a little, we convert it to a tibble, then use `prefix_strip()` and `prefix_replace()` to tidy the labels. We want to strip the polviews_m and sex prefixes, and (to avoid ambiguity about "Other"), adjust the race prefix.

```
bo_gg <- as_tibble(summary(bo_m))
prefixes <- c("polviews_m", "sex")
bo_gg$factor <- prefix_strip(bo_gg$factor, prefixes)
bo_gg$factor <- prefix_replace(bo_gg$factor, "race", "Race: ")

bo_gg %>% select(factor, AME, lower, upper)
#> # A tibble: 9 x 4
#>   factor                    AME  lower  upper
#>   <chr>                   <dbl>  <dbl>  <dbl>
#> 1 Conservative          -0.412 -0.467 -0.356
#> 2 Extremely Conservative -0.454 -0.536 -0.371
#> 3 Extremely Liberal       0.268  0.210  0.326
#> 4 Liberal                 0.277  0.232  0.322
#> 5 Slightly Conservative  -0.266 -0.330 -0.201
#> 6 Slightly Liberal        0.193  0.134  0.253
#> # ... with 3 more rows
```

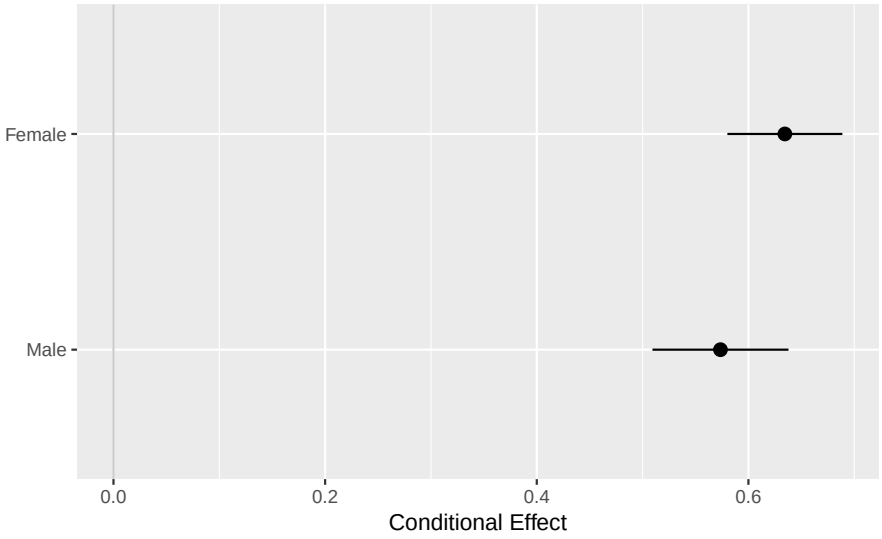
Now we have a table that we can plot as we have learned:

```
p <- ggplot(data = bo_gg, aes(x = reorder(factor, AME),
                              y = AME, ymin = lower, ymax = upper))

p + geom_hline(yintercept = 0, color = "gray80") +
    geom_pointrange() + coord_flip() +
    labs(x = NULL, y = "Average Marginal Effect")
```



If we are just interested in getting conditional effects for a particular variable, then conveniently we can ask the plot methods in the margins library to do the work calculating effects for us but without drawing their plot. Instead, they can return the results in a format we can easily use in ggplot, and with less need for clean up, for the clean-up. For example, with `cplot()`:

```
pv_cp <- cplot(out_bo, x = "sex", draw = FALSE)
#>    xvals yvals upper lower
#> 1   Male 0.574 0.638 0.509
#> 2 Female 0.634 0.689 0.580

p <- ggplot(data = pv_cp, aes(x = reorder(xvals, yvals),
                              y = yvals, ymin = lower, ymax = upper))

p + geom_hline(yintercept = 0, color = "gray80") +
    geom_pointrange() + coord_flip() +
    labs(x = NULL, y = "Conditional Effect")
```



The `margins` package is under active development. It can do much more than described here. The vignettes that come with the package provide more extensive discussion and numerous examples.

7.9 Plots from complex surveys

Social scientists often work with data collected using a complex survey design. Survey instruments may be stratified by region or some other characteristic, contain replicate weights to make them comparable to a reference population, have a clustered structure, and so on. In Chapter 4 we learned how calculate and then plot frequency tables of categorical variables, using some data from the General Social Survey (GSS). However, if we want accurate estimates of US households from the GSS, we will need to take the survey's design into account, and use the survey weights provided in the dataset. Thomas Lumley's `survey` library provides a comprehensive set of tools for addressing these issues. The tools and the theory behind them are discussed in detail in Lumley (2010), and an overview of the package is provided in Lumley (2004). While the functions in the `survey` package are straightforward to use and return results in a generally tidy form, the package predates the tidyverse and its conventions by several years. This means we cannot use survey functions directly with `dplyr`. However, Greg Freedman Ellis has written a helper package, `srvyr`, that solves this problem for us, and lets us use the `survey` library's functions within a data analysis pipeline in a familiar way.

For example, the `gss_lon` data contains a small subset of measures from every wave of the GSS since its inception in 1972. It also contains several variables that describe the design of the survey and provide replicate weights for observations in various years. These technical details are described in the GSS documentation. Similar information is typically provided by other complex surveys. Here we will use this design information to calculate weighted estimates of the distribution of educational attainment by race, for selected survey years from 1976 to 2016.

To begin, we load the `survey` and `srvyr` libraries.

```
library(survey)
#> Loading required package: grid
#> Loading required package: Matrix
#>
#> Attaching package: 'Matrix'
#> The following object is masked from 'package:tidyr':
#>
#>     expand
#>
#> Attaching package: 'survey'
#> The following object is masked from 'package:graphics':
#>
#>     dotchart
library(srvyr)
#>
#> Attaching package: 'srvyr'
#> The following object is masked from 'package:stats':
#>
#>     filter
```

Next, we take our `gss_lon` dataset and use the survey tools to create a new object that contains the data, as before, but with some additional information about the survey's design:

```
options(survey.lonely.psu = "adjust")
options(na.action="na.pass")

gss_wt <- subset(gss_lon, year > 1974) %>%
    mutate(stratvar = interaction(year, vstrat)) %>%
    as_survey_design(ids = vpsu,
```

```
                    strata = stratvar,
                    weights = wtssall,
                    nest = TRUE)
```

The two options set at the beginning provide some information to the survey library about how to behave. You should consult Lumley (2010) and the survey package documentation for details. The subsequent operations create `gss_wt`, an object with one additional column (`stratvar`), describing the yearly sampling strata. We use the `interaction()` function to do this. It multiplies the `vstrat` variable by the `year` variable to get a vector of stratum information for each year. We have to do this because of the way the GSS codes its stratum information. In the next step, we use the `as_survey_design()` function to add the key pieces of information about the survey design. It adds information about the sampling identifiers (`ids`), the strata (`strata`), and the replicate weights (`weights`). With those in place we can take advantage of a large number of specialized functions in the survey library that allow us to calculate properly weighted survey means or estimate models with the correct sampling specification. For example, we can easily calculate the distribution of education by race for a series of years from 1976 to 2016. We use `survey_mean()` to do this:

```
out_grp <- gss_wt %>%
    filter(year %in% seq(1976, 2016, by = 4)) %>%
    group_by(year, race, degree) %>%
    summarize(prop = survey_mean(na.rm = TRUE))
#> Warning: Factor `degree` contains implicit NA, consider using
#> `forcats::fct_explicit_na`

out_grp
#> # A tibble: 150 x 5
#>    year race  degree           prop prop_se
#>   <dbl> <fct> <fct>           <dbl>   <dbl>
#> 1  1976 White Lt High School 0.328  0.0160
#> 2  1976 White High School    0.518  0.0162
#> 3  1976 White Junior College 0.0129 0.00298
#> 4  1976 White Bachelor       0.101  0.00960
#> 5  1976 White Graduate       0.0393 0.00644
#> 6  1976 Black Lt High School 0.562  0.0611
#> # ... with 144 more rows
```

The results returned in `out_grp` include standard errors. We can also ask `survey_mean()` to calculate confidence intervals for us, if we wish.

Grouping with `group_by()` lets us calculate counts or means for the innermost variable, grouped by the next variable "up" or "out", in this case, degree by race, such that the proportions for degree will sum to one for each group in race, and this will be done separately for each value of year. If we want the marginal frequencies, such that the values for all combinations of race and degree sum to one within each year, we first have to interact the variables we are cross-classifying. Then we group by the new interacted variable and do the calculation as before:

```
out_mrg <- gss_wt %>%
    filter(year %in% seq(1976, 2016, by = 4)) %>%
    mutate(racedeg = interaction(race, degree)) %>%
    group_by(year, racedeg) %>%
    summarize(prop = survey_mean(na.rm = TRUE))
#> Warning: Factor `racedeg` contains implicit NA, consider using
#> `forcats::fct_explicit_na`

out_mrg
```

```
#> # A tibble: 150 x 4
#>    year racedeg                 prop prop_se
#>   <dbl> <fct>                  <dbl>   <dbl>
#> 1  1976 White.Lt High School 0.298   0.0146
#> 2  1976 Black.Lt High School 0.0471  0.00840
#> 3  1976 Other.Lt High School 0.00195 0.00138
#> 4  1976 White.High School    0.471   0.0160
#> 5  1976 Black.High School    0.0283  0.00594
#> 6  1976 Other.High School    0.00325 0.00166
#> # ... with 144 more rows
```

This gives us the numbers that we want and returns them in a tidy data frame. The `interaction()` function produces variable labels that are a compound of the two variables we interacted, with each combination of categories separated by a period, (such as White.Graduate. However, perhaps we would like to see these categories as two separate columns, one for race and one for education, as before. Because the variable labels are organized in a predictable way, we can use one of the convenient functions in the tidyverse's `tidyr` library to separate the single variable into two columns while correctly preserving the row values. Appropriately, this function is called `separate()`.

```
out_mrg <- gss_wt %>%
    filter(year %in% seq(1976, 2016, by = 4)) %>%
    mutate(racedeg = interaction(race, degree)) %>%
    group_by(year, racedeg) %>%
    summarize(prop = survey_mean(na.rm = TRUE)) %>%
    separate(racedeg, sep = "\\.", into = c("race", "degree"))
#> Warning: Factor `racedeg` contains implicit NA, consider using
#> `forcats::fct_explicit_na`

out_mrg
#> # A tibble: 150 x 5
#>    year race  degree             prop prop_se
#>   <dbl> <chr> <chr>             <dbl>   <dbl>
#> 1  1976 White Lt High School 0.298   0.0146
#> 2  1976 Black Lt High School 0.0471  0.00840
#> 3  1976 Other Lt High School 0.00195 0.00138
#> 4  1976 White High School    0.471   0.0160
#> 5  1976 Black High School    0.0283  0.00594
#> 6  1976 Other High School    0.00325 0.00166
#> # ... with 144 more rows
```

The call to `separate()` says to take the `racedeg` column, split each value when it sees a period, and reorganize the results into two columns, race and degree. This gives us a tidy table much like `out_grp`, but for the marginal frequencies.

Reasonable people can disagree over how best to plot a small multiple of a frequency table while faceting by year, especially when there is some measure of uncertainty attached. A barplot is the obvious approach for a single case, but when there are many years it can become difficult to compare bars across panels. This is especially the case when standard errors or confidence intervals are used in conjunction with bars.Sometimes it may be preferable to show that the underlying variable is categorical, as a bar chart makes clear, and not continuous, as a line graph suggests. Here the trade-off is in favor of the line graphs as the bars are very hard to compare across facets. This is sometimes called a "dynamite plot", not because it looks amazing but because the t-shaped error bars on the tops of the columns make them look like cartoon dynamite plungers. An alternative is to use a line graph to join up the time observations, faceting on educational categories instead of year. Figure 6.12 shows the results for our GSS data in *dynamite-plot* form, where the error bars are defined as twice the standard error in either direction around the point estimate.

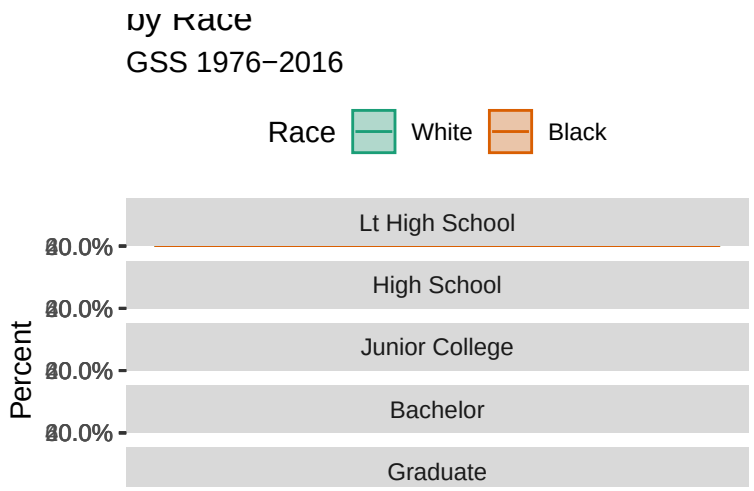
```
p <- ggplot(data = subset(out_grp, race %nin% "Other"),
            mapping = aes(x = degree, y = prop,
                          ymin = prop - 2*prop_se,
                          ymax = prop + 2*prop_se,
                          fill = race,
                          color = race,
                          group = race))

dodge <- position_dodge(width=0.9)

p + geom_col(position = dodge, alpha = 0.2) +
    geom_errorbar(position = dodge, width = 0.2) +
    scale_x_discrete(labels = scales::wrap_format(10)) +
    scale_y_continuous(labels = scales::percent) +
    scale_color_brewer(type = "qual", palette = "Dark2") +
    scale_fill_brewer(type = "qual", palette = "Dark2") +
    labs(title = "Educational Attainment by Race",
         subtitle = "GSS 1976-2016",
         fill = "Race",
         color = "Race",
         x = NULL, y = "Percent") +
    facet_wrap(~ year, ncol = 2) +
    theme(legend.position = "top")
```

This plot has a few cosmetic details and adjustments that we will learn more about in Chapter 8. As before, I encourage you to peel back the plot from the bottom, one instruction at a time, to see what changes. One useful adjustment to notice is the new call to the `scales` library to adjust the labels on the x-axis. The adjustment on the y-axis is familiar, `scales::percent` to convert the proportion to a percentage. On the x-axis, the issue is that several of the labels are rather long. If we do not adjust them they will print over one another. The `scales::wrap_format()` function will break long labels into lines. It takes a single numerical argument (here 10) that is the maximum length a string can be before it is wrapped onto a new line.

Faceting by education instead.Figure 6.13: Faceting by education instead. A graph like this is true to the categorical nature of the data, while showing the breakdown of groups within each year. But you should experiment with some alternatives. For example, we might decide that it is better to facet by degree category instead, and put the year on the x-axis within each panel. If we do that, then we can use

`geom_line()` to show a time trend, which is more natural, and `geom_ribbon()` to show the error range. This is perhaps a better way to show the data, especially as it brings out the time trends within each degree category, and allows us to see the similarities and differences by racial classification at the same time.

```
p <- ggplot(data = subset(out_grp, race %nin% "Other"),
            mapping = aes(x = year, y = prop, ymin = prop - 2*prop_se,
                          ymax = prop + 2*prop_se, fill = race, color = race,
                          group = race))

p + geom_ribbon(alpha = 0.3, aes(color = NULL)) +
    geom_line() +
    facet_wrap(~ degree, ncol = 1) +
    scale_y_continuous(labels = scales::percent) +
    scale_color_brewer(type = "qual", palette = "Dark2") +
    scale_fill_brewer(type = "qual", palette = "Dark2") +
    labs(title = "Educational Attainment\nby Race",
         subtitle = "GSS 1976-2016", fill = "Race",
         color = "Race", x = NULL, y = "Percent") +
    theme(legend.position = "top")
```



7.10 Where to go next

In general, when you estimate models and want to plot the results, the difficult step is not the plotting but rather calculating and extracting the right numbers. Generating predicted values and measures of confidence or uncertainty from models requires that you understand the model you are fitting, and the function you use to fit it, especially when it involves interactions, cross-level effects, or transformations of the predictor or response scales. The details can vary substantially from model type to model type, and also with the goals of any particular analysis. It is unwise to approach them mechanically. That said, several tools exist to help you work with model objects and produce a default set of plots from them.

7.10.1 Default plots for models

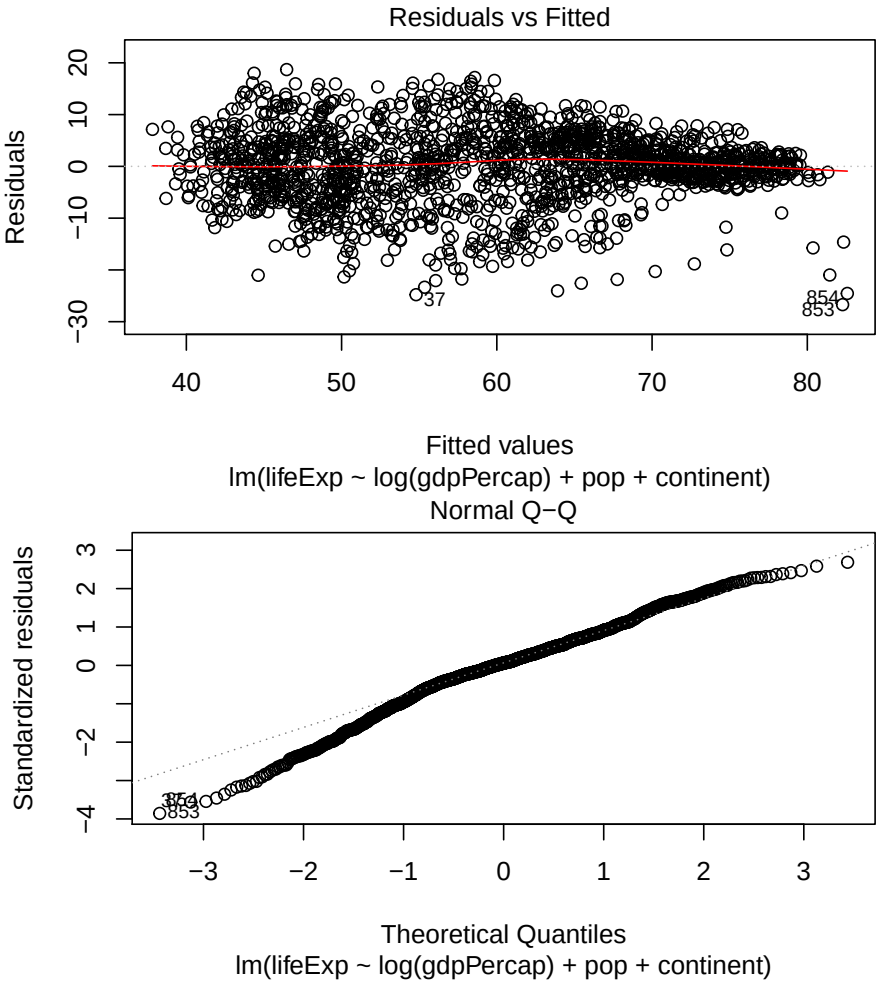
Just as model objects in R usually have a default `summary()` method, printing out an overview tailored to the type of model it is, they will usually have a default `plot()` method, too. Figures produced by `plot()` are typically not generated via `ggplot`, but it is usually worth exploring them. They typically make use of

either R's base graphics or the `lattice` library (Sarkar, 2008). These are two plotting systems that we do not cover in this book. Default plot methods are easy to examine. Let's take a look again at our simple OLS model.

```
out <- lm(formula = lifeExp ~ log(gdpPercap) + pop + continent, data = gapminder)
```

To look at some of R's default plots for this model, use the `plot()` function.

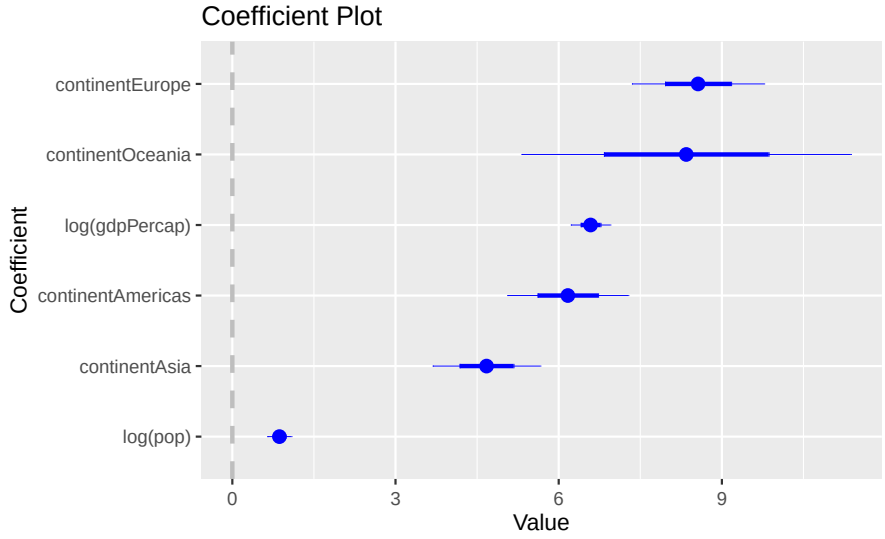
```
# Plot not shown
plot(out, which = c(1,2), ask=FALSE)
```



The `which()` statement here selects the first two of four default plots for this kind of model. If you want to easily reproduce base R's default model graphics using `ggplot`, the `ggfortify` library is worth examining. It is in some ways similar to `broom`, in that it tidies the output of model objects, but it focuses on producing a standard plot (or group of plots) for a wide variety of model types. It does this by defining a function called `autoplot()`. The idea is to be able to use `autoplot()` with the output of many different kinds of model.

A second option worth looking at is the `coefplot` library. It provides a quick way to produce good-quality plots of point estimates and confidence intervals. It has the advantage of managing the estimation of interaction effects and other occasionally tricky calculations.

```
library(coefplot)
out <- lm(formula = lifeExp ~ log(gdpPercap) + log(pop) + continent, data = gapminder)

coefplot(out, sort = "magnitude", intercept = FALSE)
```



7.10.2 Tools in development

Tidyverse tools for modeling and model exploration are being actively developed. The `broom` and `margins` libraries continue to get more and more useful. There are also other projects worth paying attention to. The `infer` packageinfer.netlify.com is in its early stages but can already do useful things in a pipeline-friendly way. You can install it from CRAN with `install.packages("infer")`.

7.10.3 Extensions to ggplot

The `GGally` package provides a suite of functions designed to make producing standard but somewhat complex plots a little easier. For instance, it can produce generalized pairs plots, a useful way of quickly examining possible relationships between several different variables at once. This sort of plot is like the visual version of a correlation matrix. It shows a bivariate plot for all pairs of variables in the data. This is relatively straightforward when all the variables are continuous measures. Things get more complex when, as is often the case in the social sciences, some or all variables are categorical or otherwise limited in the range of values they can take. A generalized pairs plot can handle these cases. For example, Figure ?? shows a generalized pairs plot for five variables from the `organdata` dataset.

```
library(GGally)

organdata_sm <- organdata %>%
    select(donors, pop_dens, pubhealth,
           roads, consent_law)

ggpairs(data = organdata_sm,
        mapping = aes(color = consent_law),
        upper = list(continuous = wrap("density"), combo = "box_no_facet"),
        lower = list(continuous = wrap("points"), combo = wrap("dot_no_facet")))
```

Multi-panel plots like this are intrinsically very rich in information. When combined with several within-panel types of representation, or any more than a modest number of variables, they can become quite complex. They should be used less for the presentation of finished work, although it is possible. More often they are a useful tool for the working researcher to quickly investigate aspects of a dataset. The goal is not to pithily summarize a single point one already knows, but to open things up for further exploration.

Bibliography