Applications of Machine Learning

Alfonso R. Reyes

2019-05-14

Contents

Chapter 1

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 2

PCA: prcomp vs princomp

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

2.1 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).

2.2 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

2.3 factoextra

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

```
library(factoextra)
```

2.4 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    11.04      7.58    14.83      2.07 49.81        14.69
#> CLAY      10.76      7.40    14.26      1.86 49.37        14.05
#> BERNARD   11.02      7.23    14.25      1.92 48.93        14.99
#> YURKOV    11.34      7.09    15.19      2.10 50.42        15.31
#> ZSIVOCZKY 11.13      7.30    13.48      2.01 48.62        14.17
#> McMULLEN  10.83      7.31    13.76      2.13 49.91        14.38
```

```
decathlon2.supplementary <- decathlon2[24:27, 1:10]
head(decathlon2.supplementary[, 1:6])
```

```
#>         X100m Long.jump Shot.put High.jump X400m X110m.hurdle
#> KARPOV  11.02      7.30    14.77      2.04 48.37        14.09
#> WARNERS 11.11      7.60    14.31      1.98 48.68        14.23
#> Nool    10.80      7.53    14.26      1.88 48.81        14.80
#> Drews   10.87      7.38    13.07      1.88 48.51        14.01
```

2.5 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

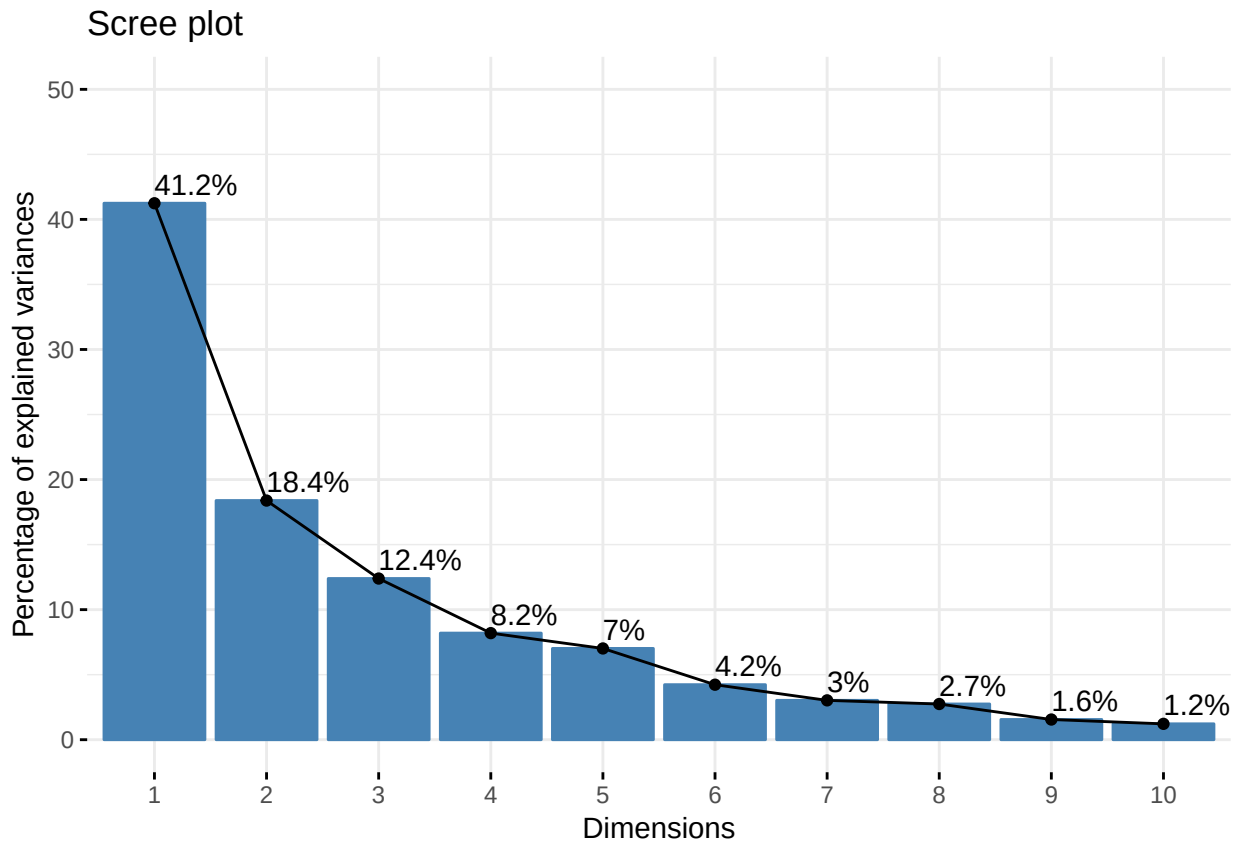
```
library(factoextra)
```

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))
```

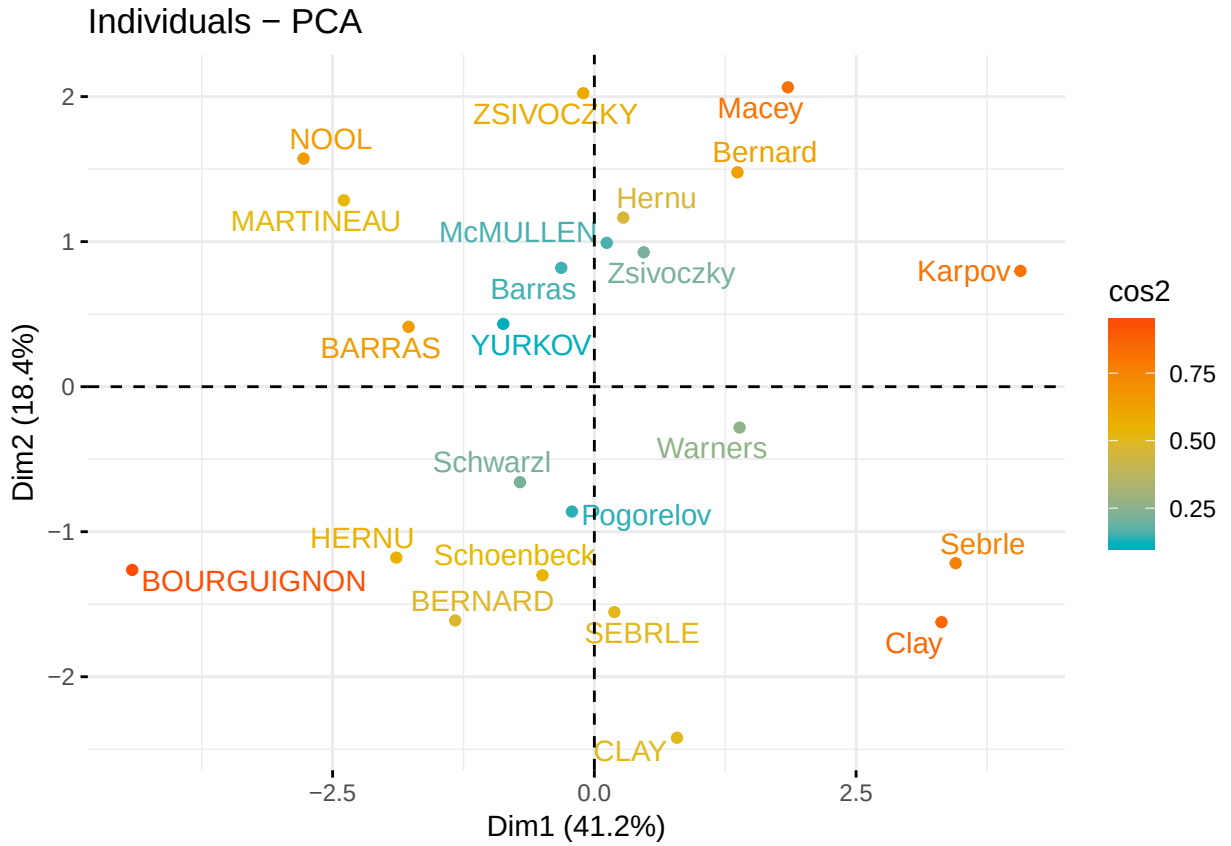


From the plot above, we might want to stop at the fifth principal component. 87% of the information (variances) contained in the data are retained by the first five principal components.

2.6 Plots: quality and contribution

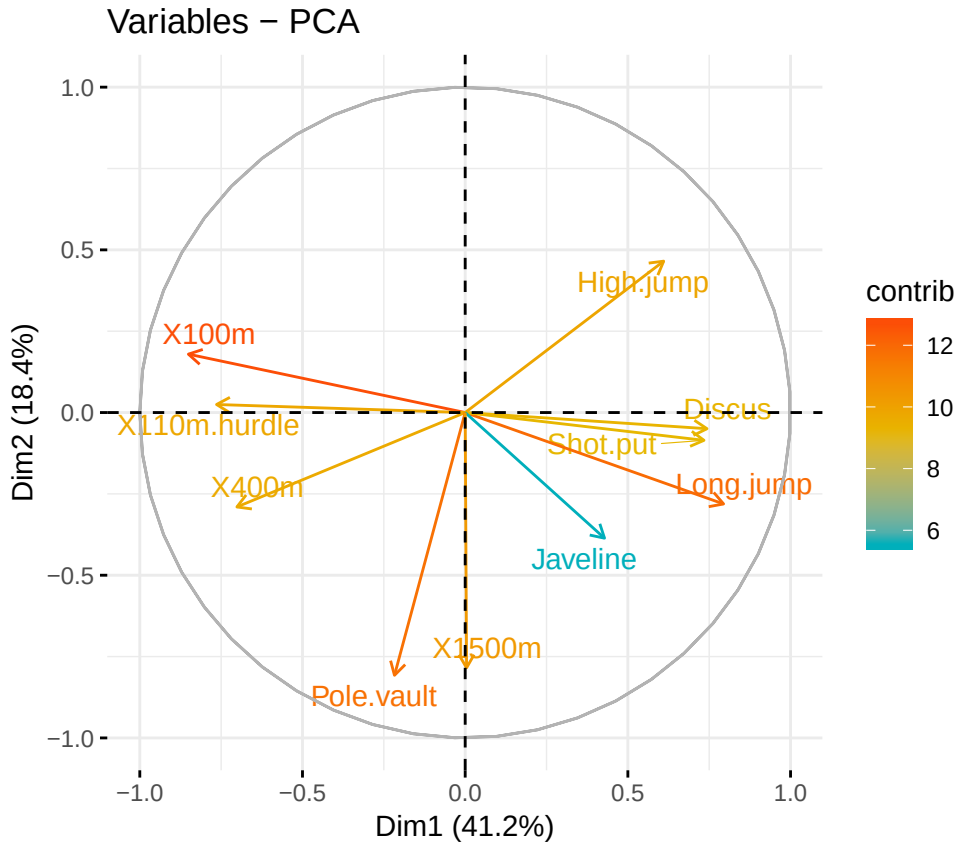
4. Graph of individuals. Individuals with a similar profile are grouped together.

```
# 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
             )
```



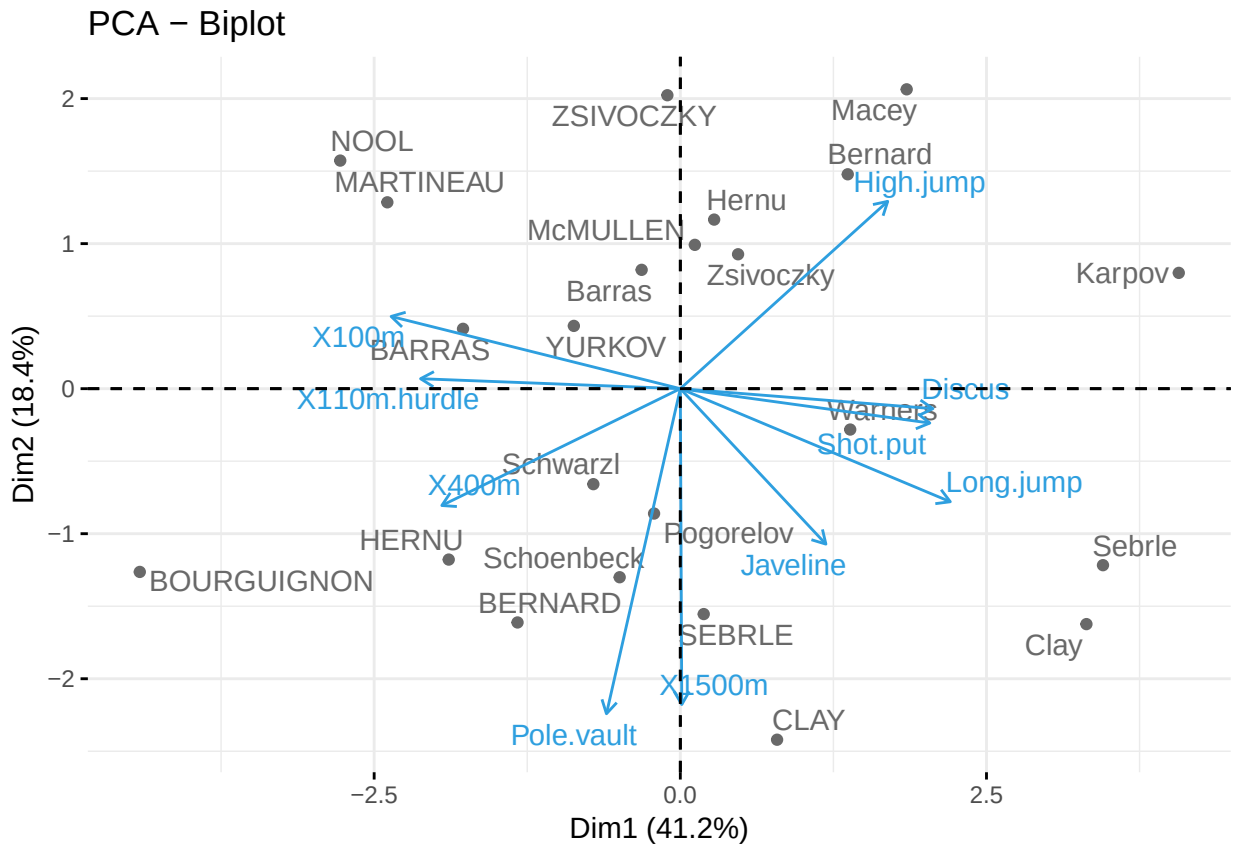
5. Graph of variables. Positive correlated variables point to the same side of the plot. Negative correlated variables point to opposite sides of the graph.

```
# 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
                )
```



2.7 Access to the PCA results

```
library(factoextra)
# Eigenvalues
eig.val <- get_eigenvalue(res.pca)
eig.val
```

```
#>        eigenvalue variance.percent cumulative.variance.percent
#> Dim.1   4.1242133        41.242133                    41.24213
#> Dim.2   1.8385309        18.385309                    59.62744
#> Dim.3   1.2391403        12.391403                    72.01885
#> Dim.4   0.8194402         8.194402                    80.21325
#> Dim.5   0.7015528         7.015528                    87.22878
#> Dim.6   0.4228828         4.228828                    91.45760
#> Dim.7   0.3025817         3.025817                    94.48342
#> Dim.8   0.2744700         2.744700                    97.22812
#> Dim.9   0.1552169         1.552169                    98.78029
#> Dim.10  0.1219710         1.219710                   100.00000
```

```
# Results for Variables
res.var <- get_pca_var(res.pca)
res.var$coord          # Coordinates
```

```
#>                   Dim.1       Dim.2       Dim.3       Dim.4      Dim.5
#> X100m        -0.850625692  0.17939806 -0.30155643  0.03357320 -0.1944440
```

```
#> Long.jump     0.794180641 -0.28085695  0.19054653 -0.11538956  0.2331567
#> Shot.put      0.733912733 -0.08540412 -0.51759781  0.12846837 -0.2488129
#> High.jump     0.610083985  0.46521415 -0.33008517  0.14455012  0.4027002
#> X400m        -0.701603377 -0.29017826 -0.28353292  0.43082552  0.1039085
#> X110m.hurdle -0.764125197  0.02474081 -0.44888733 -0.01689589  0.2242200
#> Discus        0.743209016 -0.04966086 -0.17652518  0.39500915 -0.4082391
#> Pole.vault   -0.217268042 -0.80745110 -0.09405773 -0.33898477 -0.2216853
#> Javeline      0.428226639 -0.38610928 -0.60412432 -0.33173454  0.1978128
#> X1500m        0.004278487 -0.78448019  0.21947068  0.44800961  0.2632527
#>                      Dim.6        Dim.7        Dim.8       Dim.9
#> X100m         0.035374780 -0.091336386 -0.104716925 -0.30306448
#> Long.jump    -0.033727883 -0.154330810 -0.397380703 -0.05158951
#> Shot.put     -0.239789034 -0.009886612  0.024359049  0.04778655
#> High.jump    -0.284644846  0.028157465  0.084405578 -0.11213822
#> X400m        -0.049289996  0.286106008 -0.233552216  0.08216041
#> X110m.hurdle  0.002632395 -0.370072158 -0.008344682  0.16176025
#> Discus        0.198544870 -0.142725641 -0.039559255  0.01336209
#> Pole.vault   -0.327464549 -0.010393176  0.032914942 -0.02576874
#> Javeline      0.362097598  0.133564318  0.052841099 -0.04045397
#> X1500m        0.042050151 -0.111367083  0.194469730 -0.10224014
#>                   Dim.10
#> X100m         0.044417974
#> Long.jump     0.029719453
#> Shot.put      0.217451948
#> High.jump    -0.133566774
#> X400m        -0.034170673
#> X110m.hurdle -0.015629914
#> Discus       -0.172590426
#> Pole.vault   -0.137211339
#> Javeline     -0.003854347
#> X1500m        0.062834809
```

```
res.var$contrib        # Contributions to the PCs
```

```
#>                     Dim.1      Dim.2      Dim.3       Dim.4     Dim.5
#> X100m        1.754429e+01  1.7505098  7.3386590  0.13755240  5.389252
#> Long.jump    1.529317e+01  4.2904162  2.9300944  1.62485936  7.748815
#> Shot.put     1.306014e+01  0.3967224 21.6204325  2.01407269  8.824401
#> High.jump    9.024811e+00 11.7715838  8.7928883  2.54987951 23.115504
#> X400m        1.193554e+01  4.5799296  6.4876363 22.65090599  1.539012
#> X110m.hurdle 1.415754e+01  0.0332933 16.2612611  0.03483735  7.166193
#> Discus       1.339309e+01  0.1341398  2.5147385 19.04132022 23.755756
#> Pole.vault   1.144592e+00 35.4618611  0.7139512 14.02307063  7.005084
#> Javeline     4.446377e+00  8.1086683 29.4531777 13.42963254  5.577615
#> X1500m       4.438531e-04 33.4728757  3.8871610 24.49386930  9.878367
#>                     Dim.6       Dim.7       Dim.8      Dim.9      Dim.10
#> X100m         0.295915322  2.75705260  3.99520353 59.1740009  1.61756139
#> Long.jump     0.269003613  7.87159392 57.53322220  1.7146826  0.72414393
#> Shot.put     13.596858744  0.03230371  0.21618512  1.4712015 38.76768578
#> High.jump    19.159607001  0.26202607  2.59565787  8.1015517 14.62649091
#> X400m         0.574509906 27.05274658 19.87344405  4.3489667  0.95730504
#> X110m.hurdle  0.001638634 45.26163460  0.02537025 16.8579392  0.20028870
#> Discus        9.321746508  6.73226823  0.57016606  0.1150295 24.42174410
#> Pole.vault   25.357622290  0.03569883  0.39472201  0.4278065 15.43559151
#> Javeline     31.004964393  5.89573984  1.01729950  1.0543458  0.01217993
```

```
#> X1500m        0.418133591  4.09893563 13.77872941  6.7344755  3.23700871
```

```
res.var$cos2           # Quality of representation
```

```
#>                     Dim.1        Dim.2       Dim.3        Dim.4      Dim.5
#> X100m        7.235641e-01 0.0321836641 0.090936280 0.0011271597 0.03780845
#> Long.jump    6.307229e-01 0.0788806285 0.036307981 0.0133147506 0.05436203
#> Shot.put     5.386279e-01 0.0072938636 0.267907488 0.0165041211 0.06190783
#> High.jump    3.722025e-01 0.2164242070 0.108956221 0.0208947375 0.16216747
#> X400m        4.922473e-01 0.0842034209 0.080390914 0.1856106269 0.01079698
#> X110m.hurdle 5.838873e-01 0.0006121077 0.201499837 0.0002854712 0.05027463
#> Discus       5.523596e-01 0.0024662013 0.031161138 0.1560322304 0.16665918
#> Pole.vault   4.720540e-02 0.6519772763 0.008846856 0.1149106765 0.04914437
#> Javeline     1.833781e-01 0.1490803723 0.364966189 0.1100478063 0.03912992
#> X1500m       1.830545e-05 0.6154091638 0.048167378 0.2007126089 0.06930197
#>                     Dim.6        Dim.7        Dim.8        Dim.9
#> X100m        1.251375e-03 0.0083423353 1.096563e-02 0.0918480768
#> Long.jump    1.137570e-03 0.0238179990 1.579114e-01 0.0026614779
#> Shot.put     5.749878e-02 0.0000977451 5.933633e-04 0.0022835540
#> High.jump    8.102269e-02 0.0007928428 7.124302e-03 0.0125749811
#> X400m        2.429504e-03 0.0818566479 5.454664e-02 0.0067503333
#> X110m.hurdle 6.929502e-06 0.1369534023 6.963371e-05 0.0261663784
#> Discus       3.942007e-02 0.0203706085 1.564935e-03 0.0001785453
#> Pole.vault   1.072330e-01 0.0001080181 1.083393e-03 0.0006640282
#> Javeline     1.311147e-01 0.0178394271 2.792182e-03 0.0016365234
#> X1500m       1.768215e-03 0.0124026272 3.781848e-02 0.0104530472
#>                    Dim.10
#> X100m        1.972956e-03
#> Long.jump    8.832459e-04
#> Shot.put     4.728535e-02
#> High.jump    1.784008e-02
#> X400m        1.167635e-03
#> X110m.hurdle 2.442942e-04
#> Discus       2.978746e-02
#> Pole.vault   1.882695e-02
#> Javeline     1.485599e-05
#> X1500m       3.948213e-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
#> SEBRLE       0.1912074 -1.5541282 -0.62836882  0.08205241  1.1426139415
#> CLAY         0.7901217 -2.4204156  1.35688701  1.26984296 -0.8068483724
#> BERNARD     -1.3292592 -1.6118687 -0.19614996 -1.92092203  0.0823428202
#> YURKOV      -0.8694134  0.4328779 -2.47398223  0.69723814  0.3988584116
#> ZSIVOCZKY   -0.1057450  2.0233632  1.30493117 -0.09929630 -0.1970241089
#> McMULLEN     0.1185550  0.9916237  0.84355824  1.31215266  1.5858708644
#> MARTINEAU   -2.3923532  1.2849234 -0.89816842  0.37309771 -2.2433515889
#> HERNU       -1.8910497 -1.1784614 -0.15641037  0.89130068 -0.1267412520
#> BARRAS      -1.7744575  0.4125321  0.65817750  0.22872866 -0.2338366980
#> NOOL        -2.7770058  1.5726757  0.60724821 -1.55548081  1.4241839810
#> BOURGUIGNON -4.4137335 -1.2635770 -0.01003734  0.66675478  0.4191518468
#> Sebrle       3.4514485 -1.2169193 -1.67816711 -0.80870696 -0.0250530746
#> Clay         3.3162243 -1.6232908 -0.61840443 -0.31679906  0.5691645854
```

```
#> Karpov       4.0703560  0.7983510  1.01501662  0.31336354 -0.7974259553
#> Macey        1.8484623  2.0638828 -0.97928455  0.58469073 -0.0002157834
#> Warners      1.3873514 -0.2819083  1.99969621 -1.01959817 -0.0405401497
#> Zsivoczky    0.4715533  0.9267436 -1.72815525 -0.18483138  0.4073029909
#> Hernu        0.2763118  1.1657260  0.17056375 -0.84869401 -0.6894795441
#> Bernard      1.3672590  1.4780354  0.83137913  0.74531557  0.8598016482
#> Schwarzl    -0.7102777 -0.6584251  1.04075176 -0.92717510 -0.2887568007
#> Pogorelov   -0.2143524 -0.8610557  0.29761010  1.35560294 -0.0150531057
#> Schoenbeck  -0.4953166 -1.3000530  0.10300360 -0.24927712 -0.6452257128
#> Barras      -0.3158867  0.8193681 -0.86169481 -0.58935985 -0.7797389436
#>                    Dim.6       Dim.7        Dim.8        Dim.9      Dim.10
#> SEBRLE      -0.46389755 -0.20796012  0.043460568 -0.659344137  0.03273238
#> CLAY         1.30420016 -0.21291866  0.617240611 -0.060125359 -0.31716015
#> BERNARD     -0.40062867 -0.40643754  0.703856040  0.170083313 -0.09908142
#> YURKOV       0.10286344 -0.32487448  0.114996135 -0.109524039 -0.11969720
#> ZSIVOCZKY    0.89554111  0.08825624 -0.202341299 -0.523103099 -0.34842265
#> McMULLEN     0.18657283  0.47828432  0.293089967 -0.105623196 -0.39317797
#> MARTINEAU   -0.45666350 -0.29975522 -0.291628488 -0.223417655 -0.61640509
#> HERNU        0.43623496 -0.56609980 -1.529404317  0.006184409  0.55368016
#> BARRAS       0.09026010  0.21594095  0.682583078 -0.669282042  0.53085420
#> NOOL         0.49716399 -0.53205687 -0.433385655 -0.115777808 -0.09622142
#> BOURGUIGNON -0.08200220 -0.59833739  0.563619921  0.525814030  0.05855882
#> Sebrle      -0.08279306  0.01016177 -0.030585843 -0.847210682  0.21970353
#> Clay         0.77715960  0.25750851 -0.580638301  0.409776590 -0.61601933
#> Karpov      -0.32958134 -1.36365568  0.345306381  0.193055107  0.21721852
#> Macey       -0.19728082 -0.26927772 -0.363219506  0.368260269  0.21249474
#> Warners     -0.55673300 -0.26739400 -0.109470797  0.180283071  0.24208420
#> Zsivoczky   -0.11383190  0.03991159  0.538039776  0.585966156 -0.14271715
#> Hernu       -0.33168404  0.44308686  0.247293566  0.066908586 -0.20868256
#> Bernard     -0.32806564  0.36357920  0.006165316  0.279488675  0.32067773
#> Schwarzl    -0.68891640  0.56568604 -0.687053339 -0.008358849 -0.30211546
#> Pogorelov   -1.59379599  0.78370119 -0.037623661 -0.130531397 -0.03697576
#> Schoenbeck   0.16172381  0.85752368 -0.255850722  0.564222295  0.29680481
#> Barras       1.17415412  0.94512710  0.365550568  0.102255763  0.61186706
```

```
res.ind$contrib        # Contributions to the PCs
```

```
#>                   Dim.1      Dim.2        Dim.3       Dim.4        Dim.5
#> SEBRLE       0.03854254  5.7118249 1.385418e+00  0.03572215 8.091161e+00
#> CLAY         0.65814114 13.8541889 6.460097e+00  8.55568792 4.034555e+00
#> BERNARD      1.86273218  6.1441319 1.349983e-01 19.57827284 4.202070e-02
#> YURKOV       0.79686310  0.4431309 2.147558e+01  2.57939100 9.859373e-01
#> ZSIVOCZKY    0.01178829  9.6816398 5.974848e+00  0.05231437 2.405750e-01
#> McMULLEN     0.01481737  2.3253860 2.496789e+00  9.13531719 1.558646e+01
#> MARTINEAU    6.03367104  3.9044125 2.830527e+00  0.73858431 3.118936e+01
#> HERNU        3.76996156  3.2842176 8.583863e-02  4.21505626 9.955149e-02
#> BARRAS       3.31942012  0.4024544 1.519980e+00  0.27758505 3.388731e-01
#> NOOL         8.12988880  5.8489726 1.293851e+00 12.83761115 1.257025e+01
#> BOURGUIGNON 20.53729577  3.7757623 3.534995e-04  2.35877858 1.088816e+00
#> Sebrle      12.55838616  3.5020697 9.881482e+00  3.47006223 3.889859e-03
#> Clay        11.59361384  6.2315181 1.341828e+00  0.53250375 2.007648e+00
#> Karpov      17.46609555  1.5072627 3.614914e+00  0.52101693 3.940874e+00
#> Macey        3.60207087 10.0732890 3.364879e+00  1.81387486 2.885677e-07
#> Warners      2.02910262  0.1879390 1.403071e+01  5.51585696 1.018550e-02
#> Zsivoczky    0.23441891  2.0310492 1.047894e+01  0.18126182 1.028128e+00
```

```
#> Hernu         0.08048777  3.2136178 1.020764e-01  3.82170515 2.946148e+00
#> Bernard       1.97075488  5.1661961 2.425213e+00  2.94737426 4.581507e+00
#> Schwarzl      0.53184785  1.0252129 3.800546e+00  4.56119277 5.167449e-01
#> Pogorelov     0.04843819  1.7533304 3.107757e-01  9.75034337 1.404313e-03
#> Schoenbeck    0.25864068  3.9969003 3.722687e-02  0.32970059 2.580092e+00
#> Barras        0.10519467  1.5876667 2.605305e+00  1.84296038 3.767994e+00
#>                    Dim.6        Dim.7        Dim.8        Dim.9       Dim.10
#> SEBRLE        2.21256620  0.621426384 2.992045e-02 12.177477305  0.03819185
#> CLAY         17.48801877  0.651413899 6.035125e+00  0.101262442  3.58568943
#> BERNARD       1.65019840  2.373652810 7.847747e+00  0.810319793  0.34994507
#> YURKOV        0.10878629  1.516564073 2.094806e-01  0.336009790  0.51072064
#> ZSIVOCZKY     8.24561722  0.111923276 6.485544e-01  7.664919832  4.32741147
#> McMULLEN      0.35788945  3.287016354 1.360753e+00  0.312501167  5.51053518
#> MARTINEAU     2.14409841  1.291109482 1.347216e+00  1.398195851 13.54402896
#> HERNU         1.95655942  4.604850849 3.705288e+01  0.001071345 10.92781554
#> BARRAS        0.08376135  0.670038259 7.380544e+00 12.547331617 10.04537028
#> NOOL          2.54127369  4.067669683 2.975270e+00  0.375477289  0.33003418
#> BOURGUIGNON   0.06913582  5.144247534 5.032108e+00  7.744571086  0.12223626
#> Sebrle        0.07047579  0.001483775 1.481898e-02 20.105546253  1.72063803
#> Clay          6.20972751  0.952824148 5.340583e+00  4.703566841 13.52708188
#> Karpov        1.11680500 26.720158115 1.888802e+00  1.043988269  1.68193477
#> Macey         0.40014909  1.041910483 2.089853e+00  3.798767930  1.60957713
#> Warners       3.18673563  1.027384225 1.898339e-01  0.910422384  2.08904756
#> Zsivoczky     0.13322327  0.022889042 4.585705e+00  9.617852173  0.72605208
#> Hernu         1.13110069  2.821027418 9.687304e-01  0.125399768  1.55234328
#> Bernard       1.10655655  1.899449022 6.021268e-04  2.188071254  3.66566729
#> Schwarzl      4.87961053  4.598122119 7.477531e+00  0.001957159  3.25357879
#> Pogorelov    26.11665608  8.825322559 2.242329e-02  0.477268755  0.04873597
#> Schoenbeck    0.26890572 10.566272800 1.036933e+00  8.917302863  3.14020004
#> Barras       14.17432302 12.835417603 2.116763e+00  0.292892746 13.34533825
```

```
res.ind$cos2           # Quality of representation
```

```
#>                   Dim.1      Dim.2        Dim.3       Dim.4        Dim.5
#> SEBRLE      0.007530179 0.49747323 8.132523e-02 0.001386688 2.689027e-01
#> CLAY        0.048701249 0.45701660 1.436281e-01 0.125791741 5.078506e-02
#> BERNARD     0.197199804 0.28996555 4.294015e-03 0.411819183 7.567259e-04
#> YURKOV      0.096109800 0.02382571 7.782303e-01 0.061812637 2.022798e-02
#> ZSIVOCZKY   0.001574385 0.57641944 2.397542e-01 0.001388216 5.465497e-03
#> McMULLEN    0.002175437 0.15219499 1.101379e-01 0.266486530 3.892621e-01
#> MARTINEAU   0.404013915 0.11654676 5.694575e-02 0.009826320 3.552552e-01
#> HERNU       0.399282749 0.15506199 2.731529e-03 0.088699901 1.793538e-03
#> BARRAS      0.616241975 0.03330700 8.478249e-02 0.010239088 1.070152e-02
#> NOOL        0.489872515 0.15711146 2.342405e-02 0.153694675 1.288433e-01
#> BOURGUIGNON 0.859698130 0.07045912 4.446015e-06 0.019618511 7.753120e-03
#> Sebrle      0.675380606 0.08395940 1.596674e-01 0.037079012 3.558507e-05
#> Clay        0.687592867 0.16475409 2.391051e-02 0.006274965 2.025440e-02
#> Karpov      0.783666922 0.03014772 4.873187e-02 0.004644764 3.007790e-02
#> Macey       0.363436037 0.45308203 1.020057e-01 0.036362957 4.952707e-09
#> Warners     0.255651956 0.01055582 5.311341e-01 0.138081100 2.182965e-04
#> Zsivoczky   0.045053176 0.17401353 6.051030e-01 0.006921739 3.361236e-02
#> Hernu       0.024824321 0.44184663 9.459148e-03 0.234196727 1.545686e-01
#> Bernard     0.289347476 0.33813318 1.069834e-01 0.085980212 1.144234e-01
#> Schwarzl    0.116721435 0.10030142 2.506043e-01 0.198892209 1.929118e-02
#> Pogorelov   0.007803472 0.12591966 1.504272e-02 0.312101619 3.848427e-05
```

```
#> Schoenbeck  0.067070098 0.46204603 2.900467e-03 0.016987442 1.138116e-01
#> Barras      0.018972684 0.12765099 1.411800e-01 0.066043061 1.156018e-01
#>                    Dim.6        Dim.7        Dim.8        Dim.9
#> SEBRLE      0.0443241299 8.907507e-03 3.890334e-04 8.954067e-02
#> CLAY        0.1326907339 3.536548e-03 2.972084e-02 2.820119e-04
#> BERNARD     0.0179131165 1.843634e-02 5.529104e-02 3.228572e-03
#> YURKOV      0.0013453555 1.341980e-02 1.681440e-03 1.525225e-03
#> ZSIVOCZKY   0.1129176906 1.096685e-03 5.764478e-03 3.852703e-02
#> McMULLEN    0.0053876990 3.540616e-02 1.329562e-02 1.726733e-03
#> MARTINEAU   0.0147210347 6.342774e-03 6.003515e-03 3.523552e-03
#> HERNU       0.0212478795 3.578167e-02 2.611676e-01 4.270425e-06
#> BARRAS      0.0015944528 9.126203e-03 9.118662e-02 8.766746e-02
#> NOOL        0.0157010551 1.798232e-02 1.193105e-02 8.514912e-04
#> BOURGUIGNON 0.0002967459 1.579887e-02 1.401866e-02 1.220108e-02
#> Sebrle      0.0003886276 5.854423e-06 5.303795e-05 4.069384e-02
#> Clay        0.0377627839 4.145976e-03 2.107924e-02 1.049876e-02
#> Karpov      0.0051379747 8.795817e-02 5.639959e-03 1.762907e-03
#> Macey       0.0041397727 7.712721e-03 1.403282e-02 1.442502e-02
#> Warners     0.0411689767 9.496848e-03 1.591742e-03 4.317040e-03
#> Zsivoczky   0.0026253777 3.227467e-04 5.865332e-02 6.956790e-02
#> Hernu       0.0357707217 6.383462e-02 1.988402e-02 1.455601e-03
#> Bernard     0.0166586433 2.046050e-02 5.883405e-06 1.209056e-02
#> Schwarzl    0.1098063093 7.403638e-02 1.092132e-01 1.616543e-05
#> Pogorelov   0.4314162233 1.043115e-01 2.404103e-04 2.893750e-03
#> Schoenbeck  0.0071500829 2.010275e-01 1.789520e-02 8.702893e-02
#> Barras      0.2621297474 1.698426e-01 2.540745e-02 1.988116e-03
#>                   Dim.10
#> SEBRLE      0.0002206741
#> CLAY        0.0078471026
#> BERNARD     0.0010956493
#> YURKOV      0.0018217256
#> ZSIVOCZKY   0.0170924251
#> McMULLEN    0.0239268142
#> MARTINEAU   0.0268211980
#> HERNU       0.0342288717
#> BARRAS      0.0551531863
#> NOOL        0.0005881295
#> BOURGUIGNON 0.0001513277
#> Sebrle      0.0027366539
#> Clay        0.0237264222
#> Karpov      0.0022318265
#> Macey       0.0048028954
#> Warners     0.0077841113
#> Zsivoczky   0.0041268259
#> Hernu       0.0141595965
#> Bernard     0.0159167991
#> Schwarzl    0.0211173850
#> Pogorelov   0.0002322016
#> Schoenbeck  0.0240826922
#> Barras      0.0711836486
```

Chapter 3

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  11.02      7.30    14.77      2.04 48.37        14.09
#> WARNERS 11.11      7.60    14.31      1.98 48.68        14.23
#> Nool    10.80      7.53    14.26      1.88 48.81        14.80
#> Drews   10.87      7.38    13.07      1.88 48.51        14.01
```

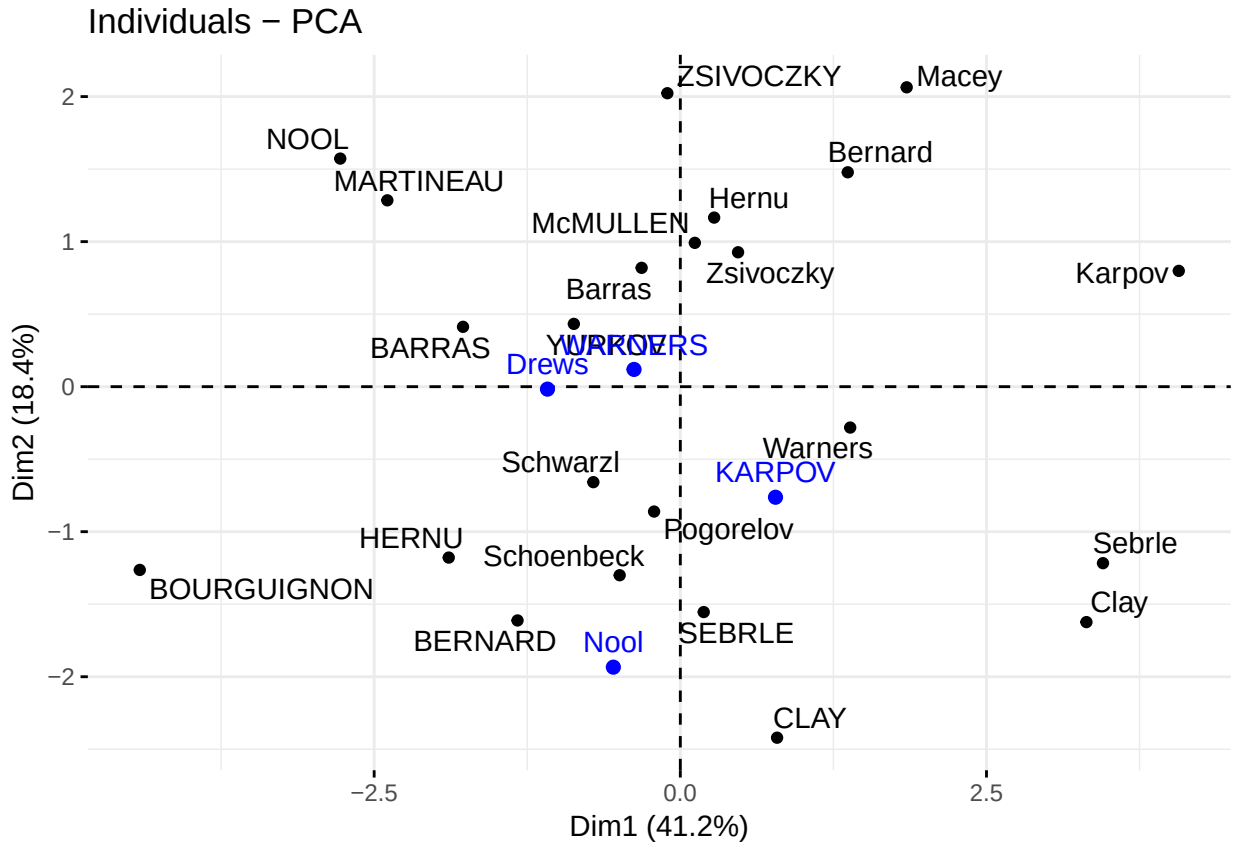
2. Predict the coordinates of new individuals data. Use the R base function predict():

```
ind.sup.coord <- predict(res.pca, newdata = ind.sup)
ind.sup.coord[, 1:4]
```

```
#>                PC1         PC2       PC3        PC4
#> KARPOV   0.7772521 -0.76237804 1.5971253  1.6863286
#> WARNERS -0.3779697  0.11891968 1.7005146 -0.6908084
#> Nool    -0.5468405 -1.93402211 0.4724184 -2.2283706
#> Drews   -1.0848227 -0.01703198 2.9818031 -1.5006207
```

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   0.7772521 -0.76237804 1.5971253  1.6863286
#> WARNERS -0.3779697  0.11891968 1.7005146 -0.6908084
#> Nool    -0.5468405 -1.93402211 0.4724184 -2.2283706
#> Drews   -1.0848227 -0.01703198 2.9818031 -1.5006207
```

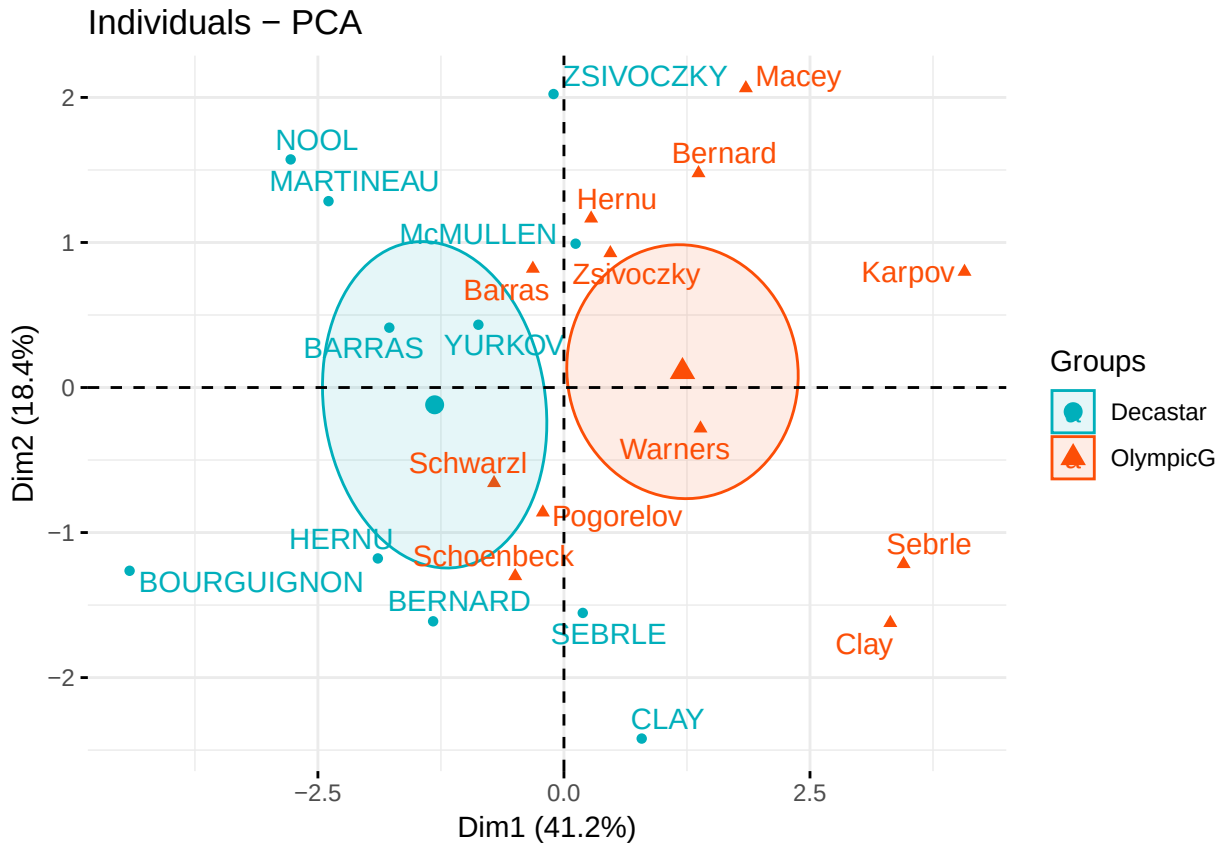
3.1 Supplementary variables

3.1.1 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

# 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)
    )
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
```

3.1.2 Quantitative variables

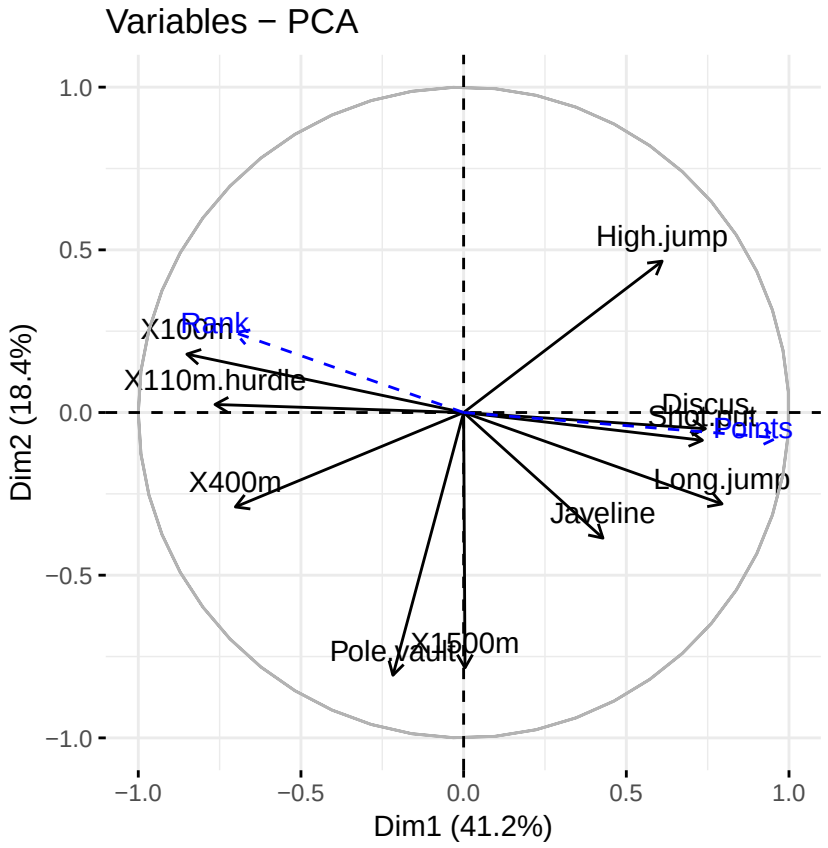
Data: columns 11:12. Should be of same length as the number of active individuals (here 23)

```
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")
```



3.2 Theory behind PCA results

3.2.1 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        -0.8506257  0.17939806 -0.3015564  0.03357320
#> Long.jump     0.7941806 -0.28085695  0.1905465 -0.11538956
```

```
#> Shot.put      0.7339127 -0.08540412 -0.5175978  0.12846837
#> High.jump     0.6100840  0.46521415 -0.3300852  0.14455012
#> X400m        -0.7016034 -0.29017826 -0.2835329  0.43082552
#> X110m.hurdle -0.7641252  0.02474081 -0.4488873 -0.01689589
```

```
# Compute Cos2
#:::::::::::::::::::::::::::::::::::::
var.cos2 <- var.coord^2
head(var.cos2[, 1:4])
```

```
#>                    PC1          PC2        PC3          PC4
#> X100m        0.7235641 0.0321836641 0.09093628 0.0011271597
#> Long.jump    0.6307229 0.0788806285 0.03630798 0.0133147506
#> Shot.put     0.5386279 0.0072938636 0.26790749 0.0165041211
#> High.jump    0.3722025 0.2164242070 0.10895622 0.0208947375
#> X400m        0.4922473 0.0842034209 0.08039091 0.1856106269
#> X110m.hurdle 0.5838873 0.0006121077 0.20149984 0.0002854712
```

```
# 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        17.544293  1.7505098  7.338659  0.13755240
#> Long.jump    15.293168  4.2904162  2.930094  1.62485936
#> Shot.put     13.060137  0.3967224 21.620432  2.01407269
#> High.jump     9.024811 11.7715838  8.792888  2.54987951
#> X400m        11.935544  4.5799296  6.487636 22.65090599
#> X110m.hurdle 14.157544  0.0332933 16.261261  0.03483735
```

3.2.2 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     0.1912074 -1.5541282 -0.6283688  0.08205241
#> CLAY       0.7901217 -2.4204156  1.3568870  1.26984296
#> BERNARD   -1.3292592 -1.6118687 -0.1961500 -1.92092203
#> YURKOV    -0.8694134  0.4328779 -2.4739822  0.69723814
#> ZSIVOCZKY -0.1057450  2.0233632  1.3049312 -0.09929630
#> McMULLEN   0.1185550  0.9916237  0.8435582  1.31215266
```

```
# 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     0.007530179 0.49747323 0.081325232 0.001386688
#> CLAY       0.048701249 0.45701660 0.143628117 0.125791741
#> BERNARD    0.197199804 0.28996555 0.004294015 0.411819183
#> YURKOV     0.096109800 0.02382571 0.778230322 0.061812637
#> ZSIVOCZKY 0.001574385 0.57641944 0.239754152 0.001388216
#> McMULLEN   0.002175437 0.15219499 0.110137872 0.266486530
```

```
# 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     0.03854254  5.7118249  1.3854184  0.03572215
#> CLAY       0.65814114 13.8541889  6.4600973  8.55568792
#> BERNARD    1.86273218  6.1441319  0.1349983 19.57827284
#> YURKOV     0.79686310  0.4431309 21.4755770  2.57939100
#> ZSIVOCZKY 0.01178829  9.6816398  5.9748485  0.05231437
#> McMULLEN   0.01481737  2.3253860  2.4967890  9.13531719
```

Chapter 4

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)
```

```
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  11.04      7.58    14.83      2.07 49.81        14.69
:> CLAY    10.76      7.40    14.26      1.86 49.37        14.05
:> BERNARD 11.02      7.23    14.25      1.92 48.93        14.99
:> YURKOV  11.34      7.09    15.19      2.10 50.42        15.31
```

4.1 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.

4.2 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   4.1242133        41.242133                     41.24213
:> Dim.2   1.8385309        18.385309                     59.62744
:> Dim.3   1.2391403        12.391403                     72.01885
:> Dim.4   0.8194402         8.194402                     80.21325
:> Dim.5   0.7015528         7.015528                     87.22878
:> Dim.6   0.4228828         4.228828                     91.45760
:> Dim.7   0.3025817         3.025817                     94.48342
:> Dim.8   0.2744700         2.744700                     97.22812
:> Dim.9   0.1552169         1.552169                     98.78029
:> Dim.10  0.1219710         1.219710                    100.00000
```

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.

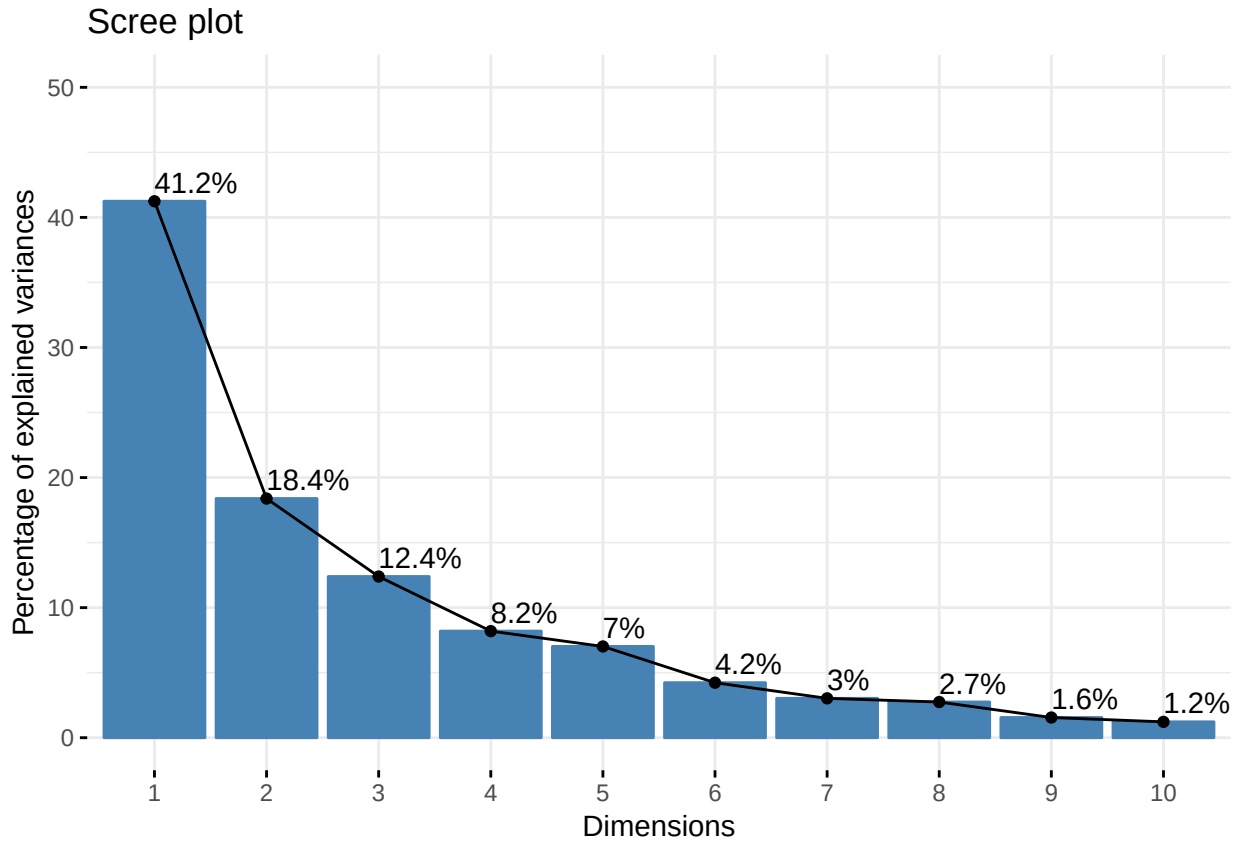
Unfortunately, there is no well-accepted objective way to decide how many principal components are enough. This will depend on the specific field of application and the specific data set. In practice, we tend to look at the first few principal components in order to find interesting patterns in the data.

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.

4.3 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        -0.8506257 -0.17939806  0.3015564  0.03357320 -0.1944440
:> Long.jump     0.7941806  0.28085695 -0.1905465 -0.11538956  0.2331567
:> Shot.put      0.7339127  0.08540412  0.5175978  0.12846837 -0.2488129
:> High.jump     0.6100840 -0.46521415  0.3300852  0.14455012  0.4027002
:> X400m        -0.7016034  0.29017826  0.2835329  0.43082552  0.1039085
:> X110m.hurdle -0.7641252 -0.02474081  0.4488873 -0.01689589  0.2242200
```

```
# Cos2: quality on the factore map
head(var$cos2)
```

```
:>                  Dim.1        Dim.2      Dim.3        Dim.4      Dim.5
:> X100m        0.7235641 0.0321836641 0.09093628 0.0011271597 0.03780845
:> Long.jump    0.6307229 0.0788806285 0.03630798 0.0133147506 0.05436203
:> Shot.put     0.5386279 0.0072938636 0.26790749 0.0165041211 0.06190783
:> High.jump    0.3722025 0.2164242070 0.10895622 0.0208947375 0.16216747
:> X400m        0.4922473 0.0842034209 0.08039091 0.1856106269 0.01079698
:> X110m.hurdle 0.5838873 0.0006121077 0.20149984 0.0002854712 0.05027463
```

```
# Contributions to the principal components
head(var$contrib)
```

```
:>                  Dim.1      Dim.2     Dim.3       Dim.4     Dim.5
:> X100m        17.544293  1.7505098  7.338659  0.13755240  5.389252
:> Long.jump    15.293168  4.2904162  2.930094  1.62485936  7.748815
:> Shot.put     13.060137  0.3967224 21.620432  2.01407269  8.824401
:> High.jump     9.024811 11.7715838  8.792888  2.54987951 23.115504
:> X400m        11.935544  4.5799296  6.487636 22.65090599  1.539012
:> X110m.hurdle 14.157544  0.0332933 16.261261  0.03483735  7.166193
```

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.

4.4 Correlation circle

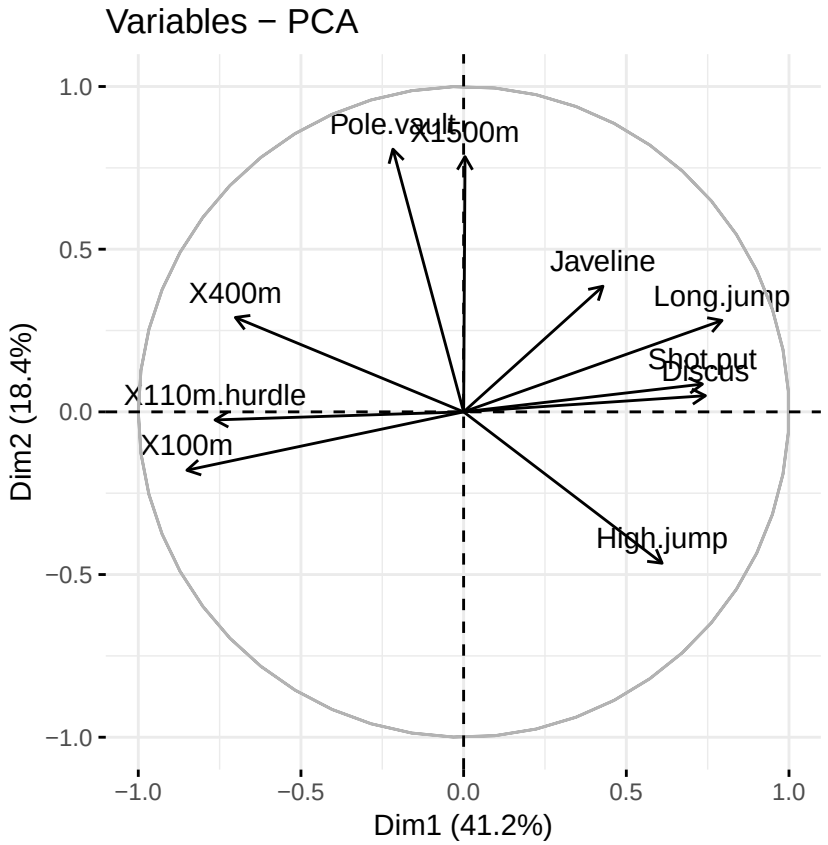
The correlation between a variable and a principal component (PC) is used as the coordinates of the variable on the PC. The representation of variables differs from the plot of the observations: The observations are represented by their projections, but the variables are represented by their correlations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

```
# Coordinates of variables
head(var$coord, 4)
```

```
:>                Dim.1       Dim.2      Dim.3      Dim.4      Dim.5
:> X100m     -0.8506257 -0.17939806  0.3015564  0.0335732 -0.1944440
:> Long.jump  0.7941806  0.28085695 -0.1905465 -0.1153896  0.2331567
:> Shot.put   0.7339127  0.08540412  0.5175978  0.1284684 -0.2488129
:> High.jump  0.6100840 -0.46521415  0.3300852  0.1445501  0.4027002
```

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.

4.5 Quality of representation

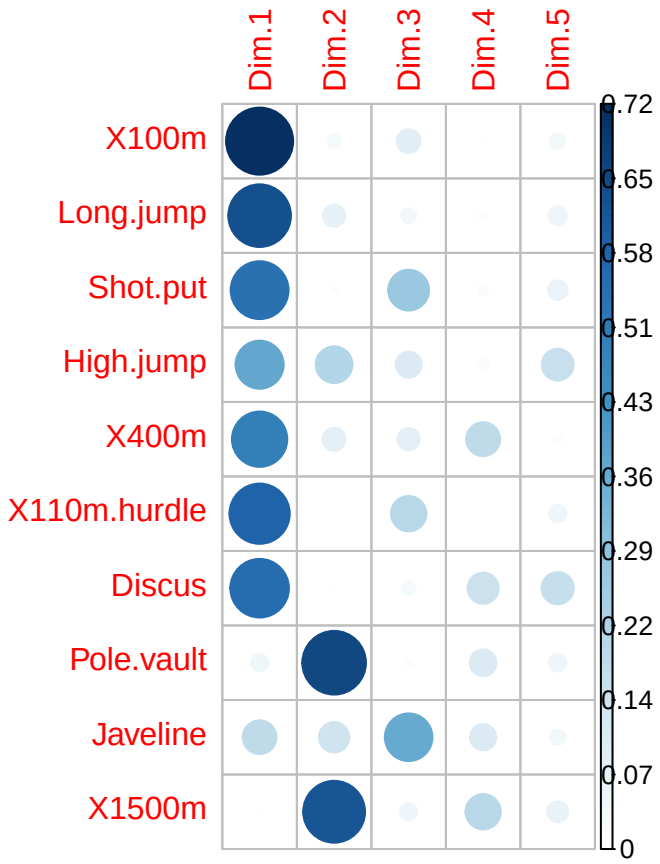
The quality of representation of the variables on factor map is called cos2 (square cosine, squared coordinates) . You can access to the cos2 as follow:

```
head(var$cos2, 4)
```

```
:>               Dim.1       Dim.2      Dim.3      Dim.4      Dim.5
:> X100m     0.7235641 0.032183664 0.09093628 0.00112716 0.03780845
:> Long.jump 0.6307229 0.078880629 0.03630798 0.01331475 0.05436203
:> Shot.put  0.5386279 0.007293864 0.26790749 0.01650412 0.06190783
:> High.jump 0.3722025 0.216424207 0.10895622 0.02089474 0.16216747
```

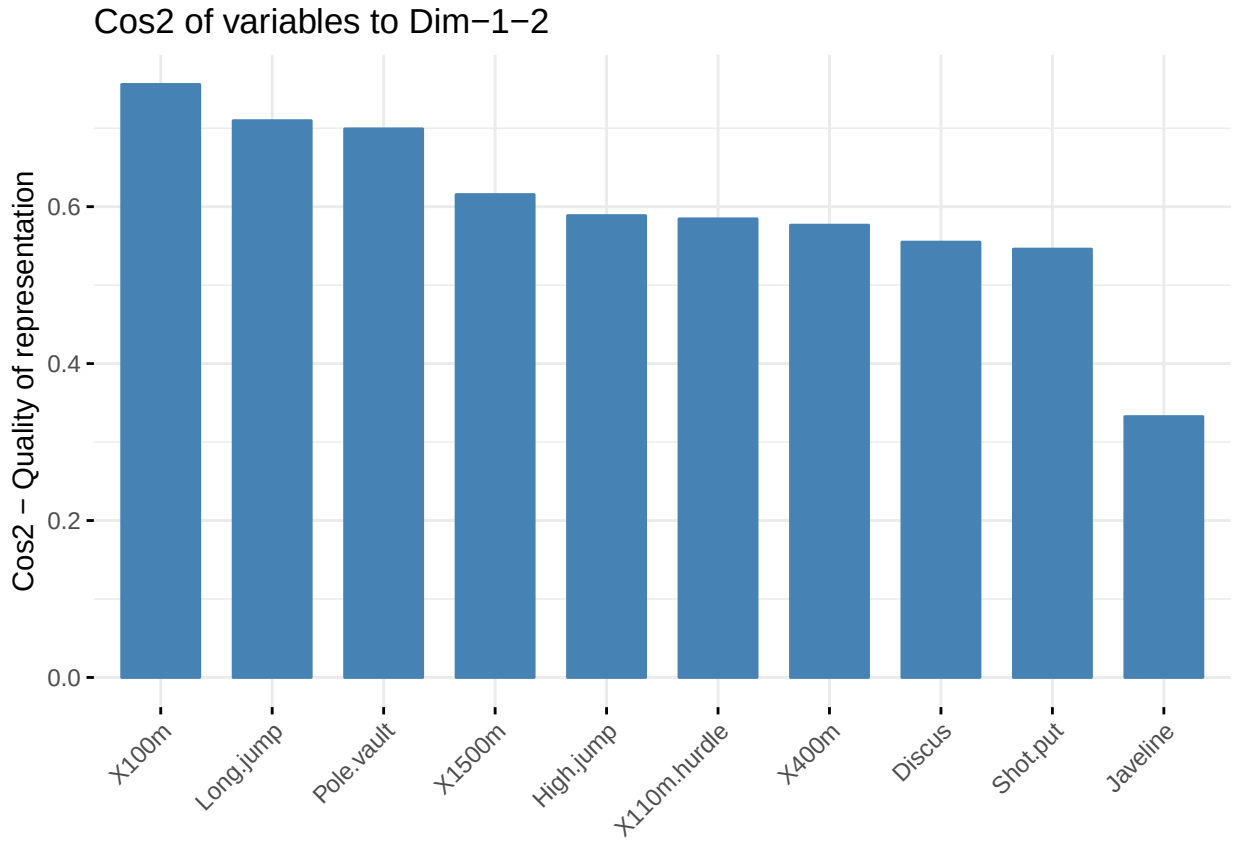
You can visualize the cos2 of variables on all the dimensions using the corrplot package:

```
library(corrplot)
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.

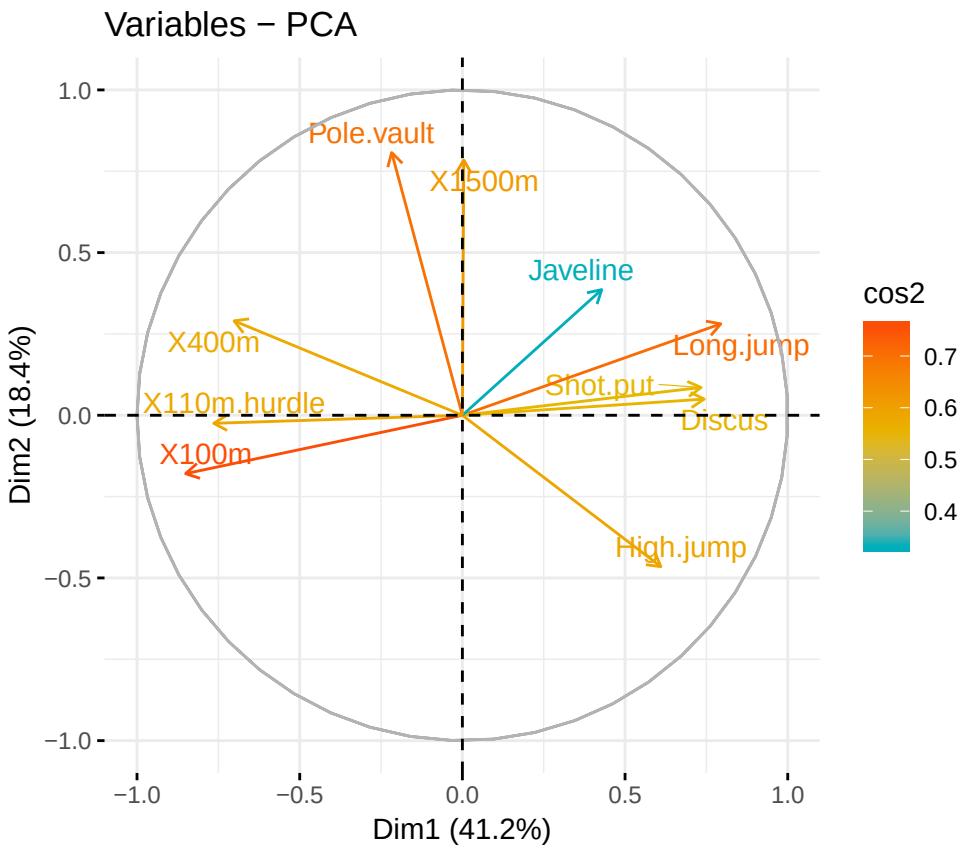
In summary:

- 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:

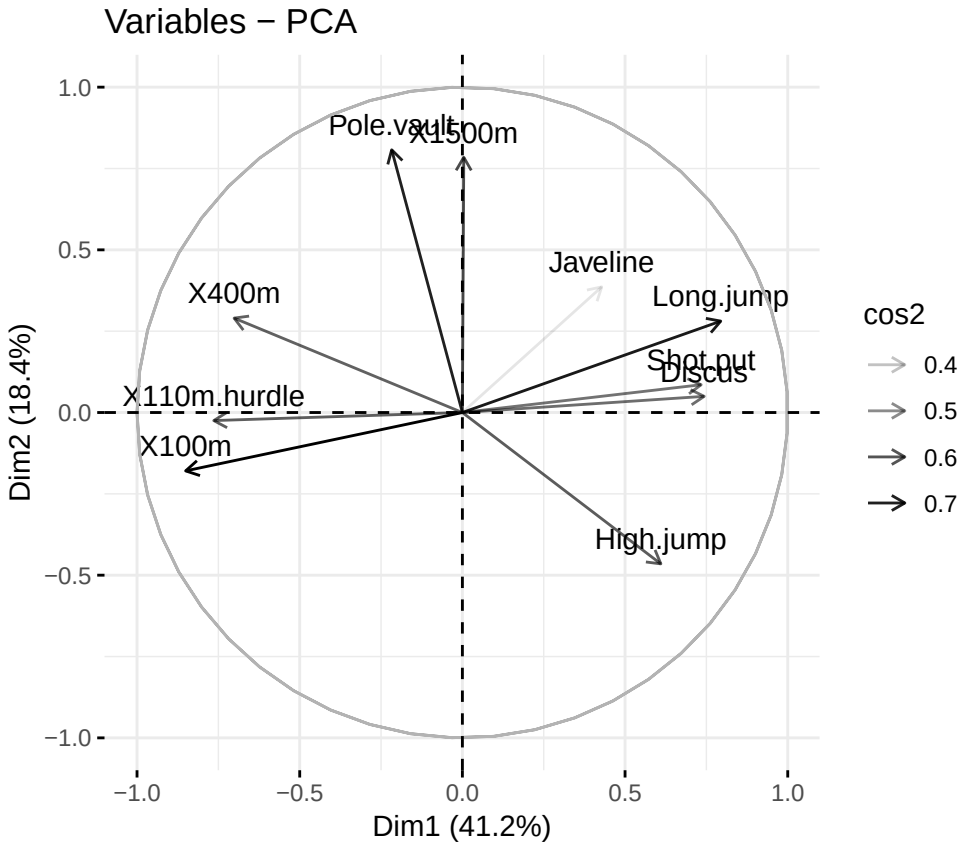
- variables with low cos2 values will be colored in "white"
- variables with mid cos2 values will be colored in "blue"
- variables with high cos2 values will be colored in red

```
# 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")
```



4.6 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 :

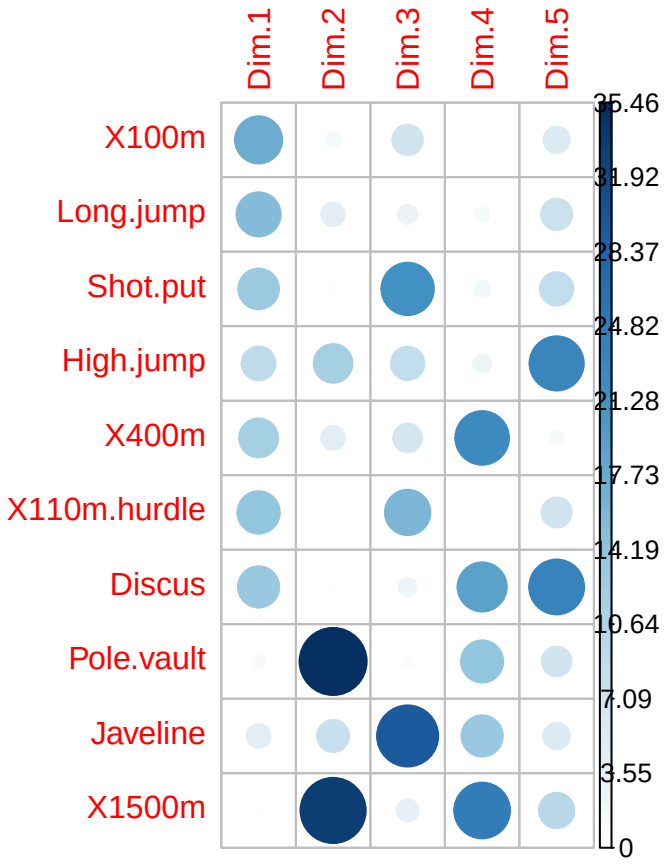
```
head(var$contrib, 4)
```

```
:>               Dim.1      Dim.2     Dim.3     Dim.4     Dim.5
:> X100m     17.544293  1.7505098  7.338659 0.1375524  5.389252
:> Long.jump 15.293168  4.2904162  2.930094 1.6248594  7.748815
:> Shot.put  13.060137  0.3967224 21.620432 2.0140727  8.824401
:> High.jump  9.024811 11.7715838  8.792888 2.5498795 23.115504
```

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

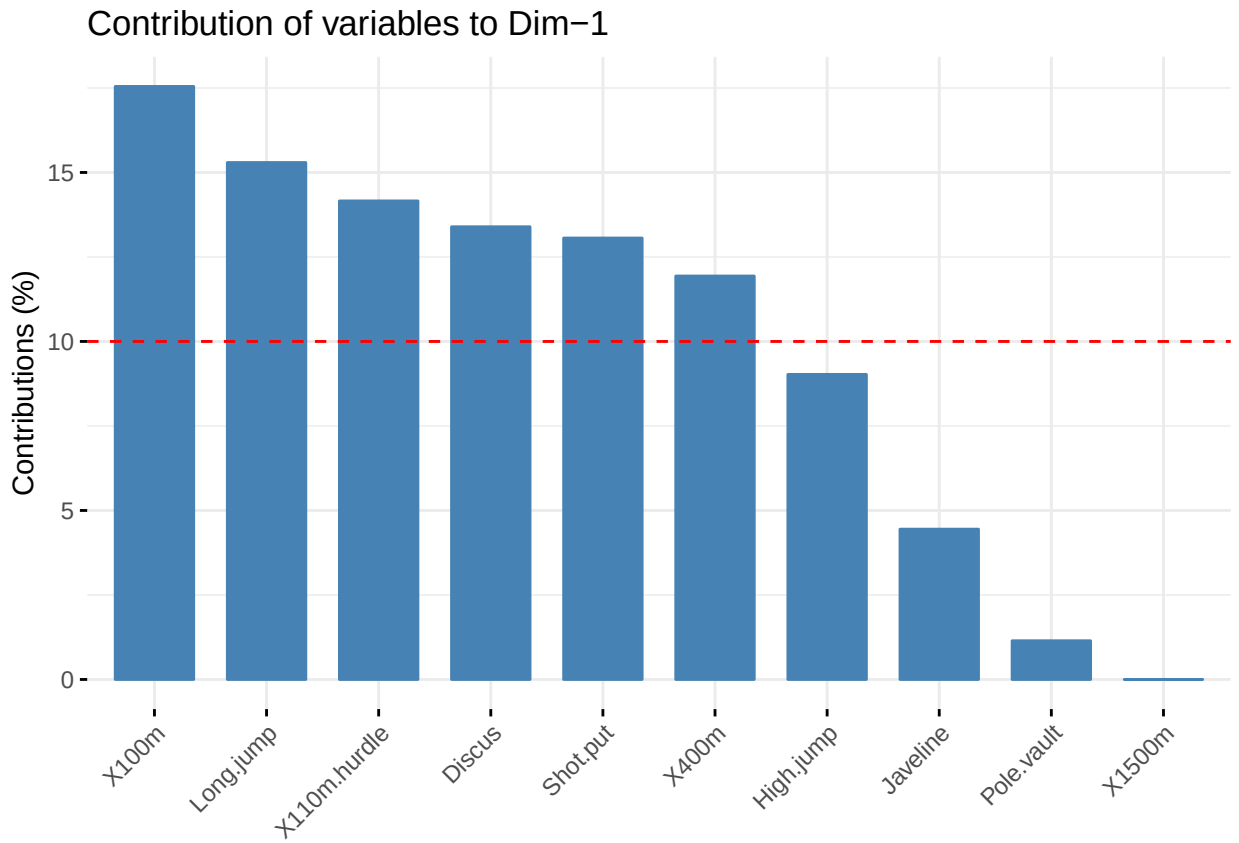
It's possible to use the function corrplot() [corrplot package] to highlight the most contributing variables for each dimension:

```
library("corrplot")

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

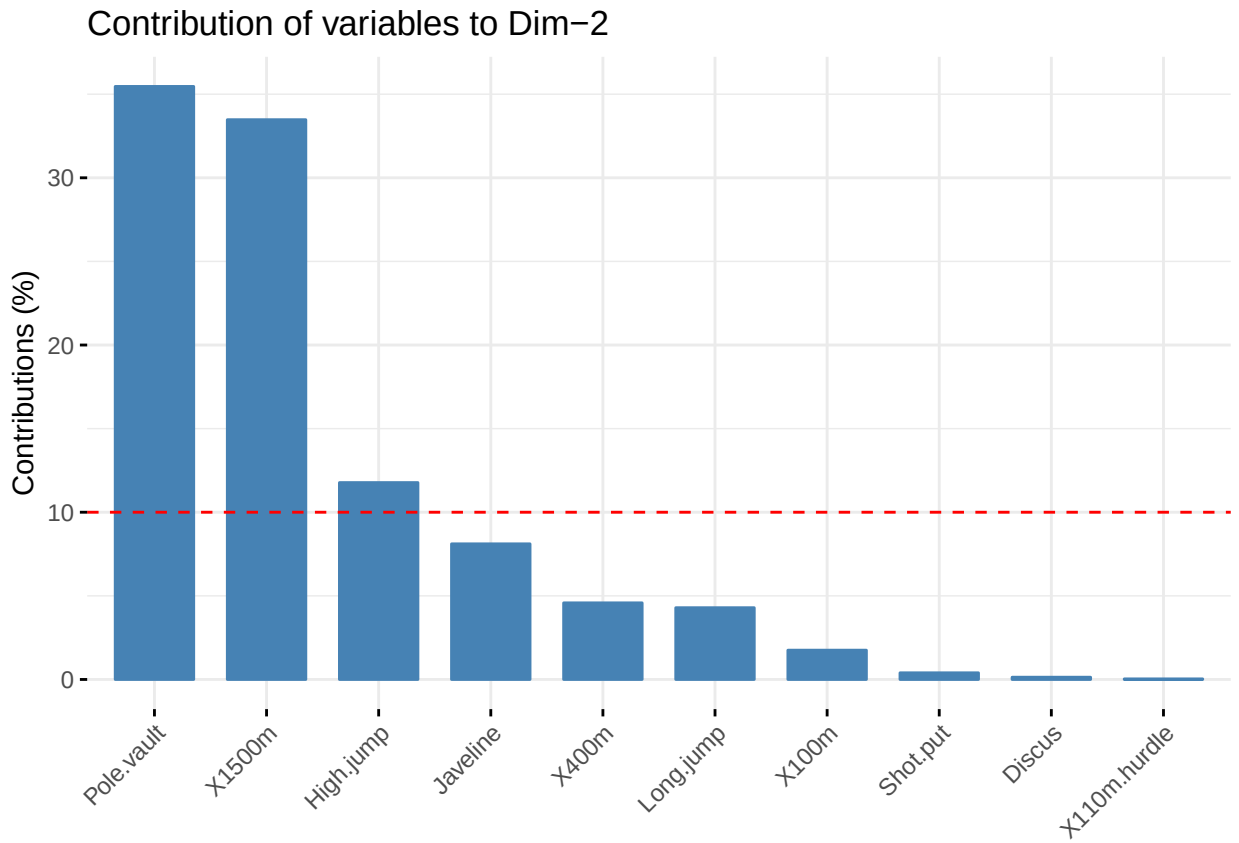


The function fviz_contrib() [factoextra package] can be used to draw a bar plot of variable contributions. If your data contains many variables, you can decide to show only the top contributing variables. The R code below shows the top 10 variables contributing to the principal components:

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

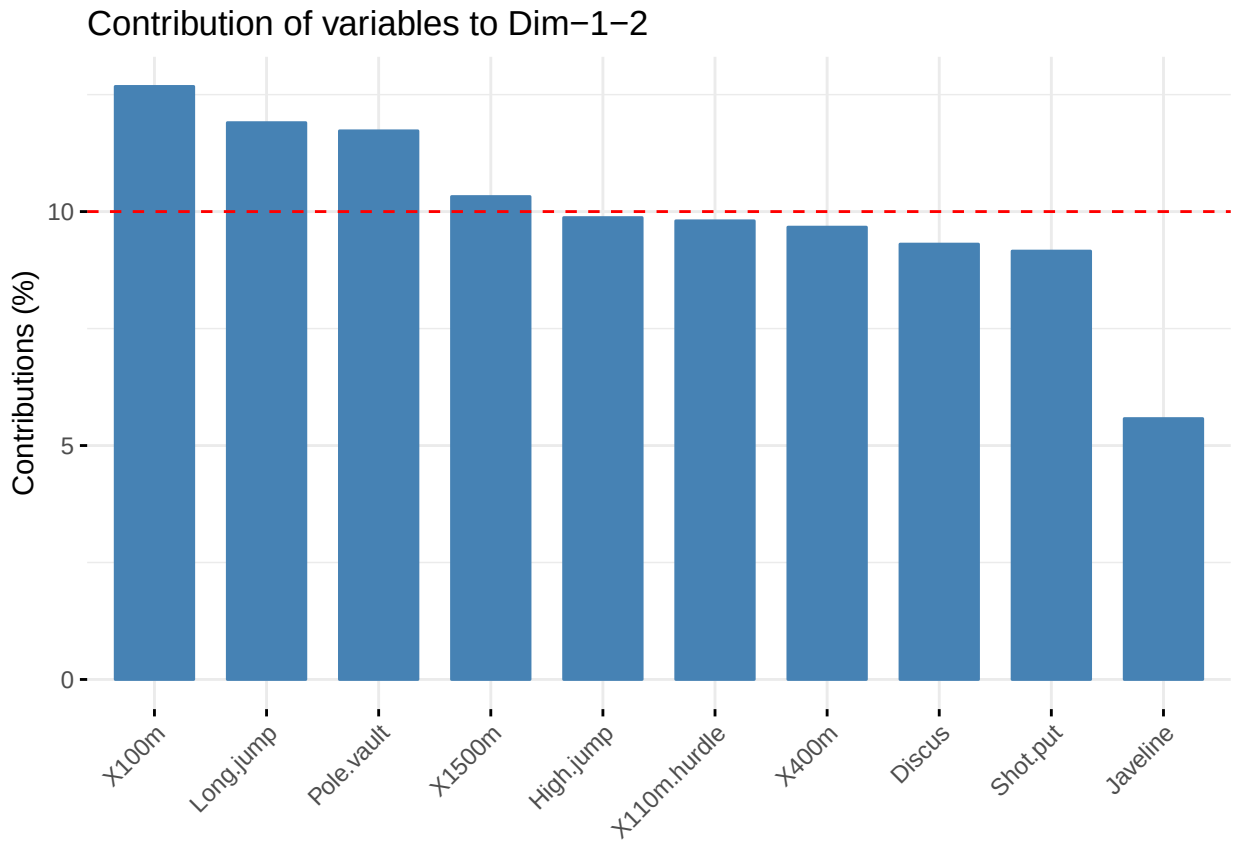



```r
# Contributions of variables to PC2
fviz_contrib(res.pca, choice = "var", axes = 2, top = 10)
```



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

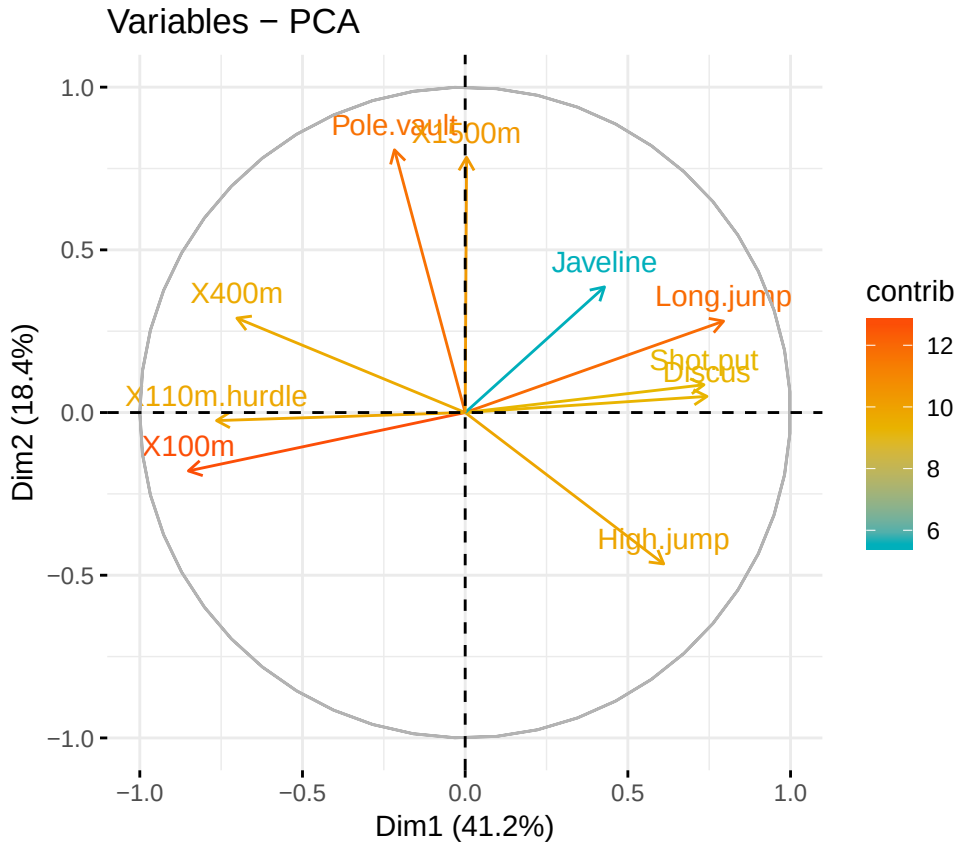
- C1 and C2 are the contributions of the variable on PC1 and PC2, respectively
- Eig1 and Eig2 are the eigenvalues of PC1 and PC2, respectively. Recall that eigenvalues measure the amount of variation retained by each PC.

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.

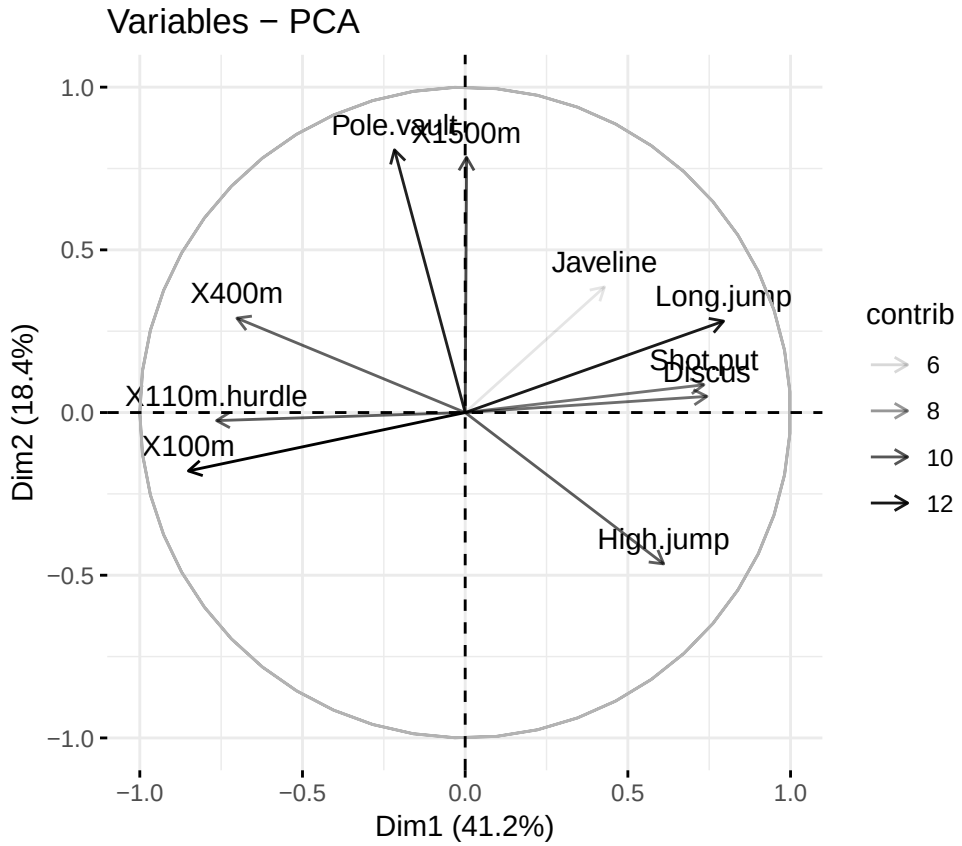
The most important (or, contributing) variables can be highlighted on the correlation plot as follow:

```
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")
```

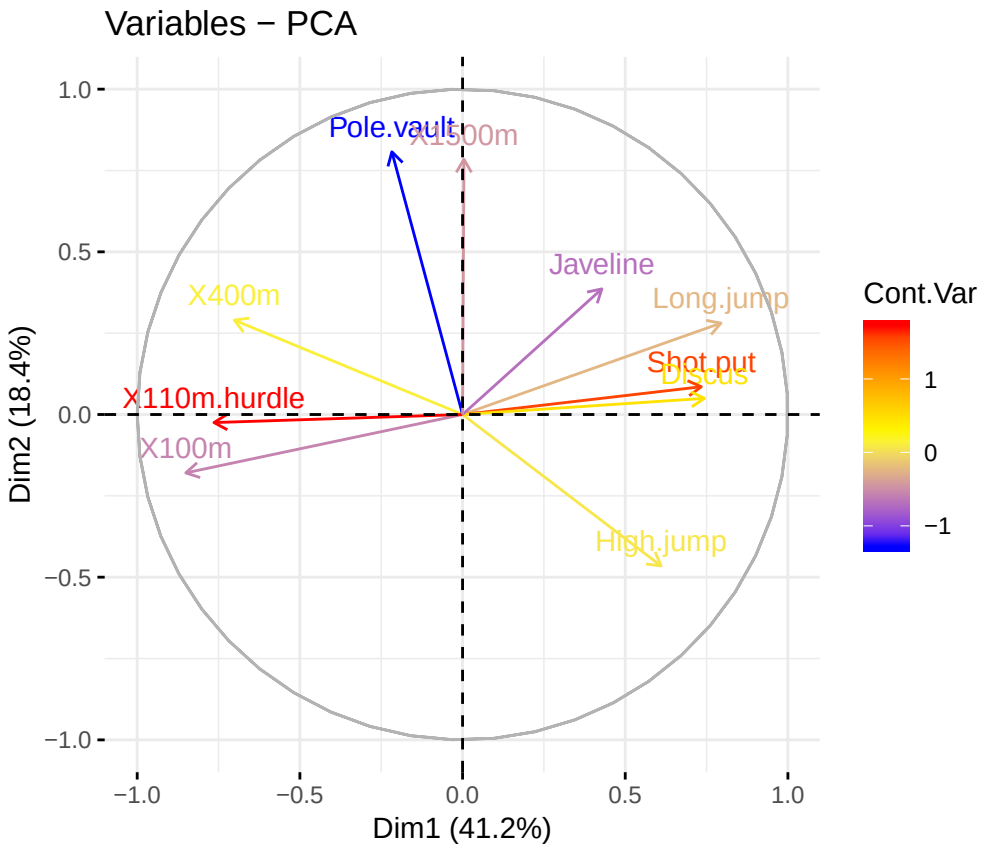


4.7 Color by a custom continuous variable

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")
```



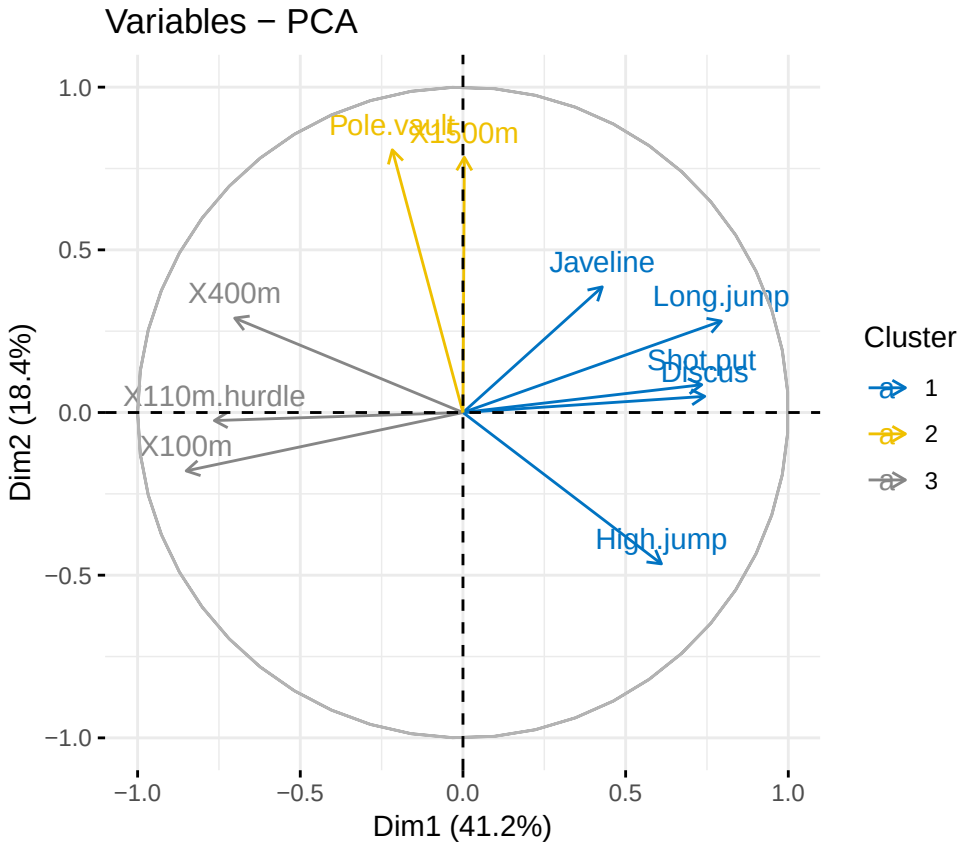
4.8 Color by groups

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")
```



4.9 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      0.7941806 6.059893e-06
:> Discus         0.7432090 4.842563e-05
:> Shot.put       0.7339127 6.723102e-05
:> High.jump      0.6100840 1.993677e-03
:> Javeline       0.4282266 4.149192e-02
:> X400m         -0.7016034 1.910387e-04
:> X110m.hurdle  -0.7641252 2.195812e-05
:> X100m         -0.8506257 2.727129e-07
```

```
# Description of dimension 2
res.desc$Dim.2
```

```
:> $quanti
```

```
:>            correlation      p.value
:> Pole.vault   0.8074511 3.205016e-06
:> X1500m       0.7844802 9.384747e-06
:> High.jump   -0.4652142 2.529390e-02
```

4.10 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     0.1955047  1.5890567  0.6424912  0.08389652  1.16829387
:> CLAY       0.8078795  2.4748137 -1.3873827  1.29838232 -0.82498206
:> BERNARD   -1.3591340  1.6480950  0.2005584 -1.96409420  0.08419345
:> YURKOV    -0.8889532 -0.4426067  2.5295843  0.71290837  0.40782264
:> ZSIVOCZKY -0.1081216 -2.0688377 -1.3342591 -0.10152796 -0.20145217
:> McMULLEN   0.1212195 -1.0139102 -0.8625170  1.34164291  1.62151286
```

```
# Quality of individuals
head(ind$cos2)
```

```
:>                 Dim.1      Dim.2       Dim.3       Dim.4        Dim.5
:> SEBRLE    0.007530179 0.49747323 0.081325232 0.001386688 0.2689026575
:> CLAY      0.048701249 0.45701660 0.143628117 0.125791741 0.0507850580
:> BERNARD   0.197199804 0.28996555 0.004294015 0.411819183 0.0007567259
:> YURKOV    0.096109800 0.02382571 0.778230322 0.061812637 0.0202279796
:> ZSIVOCZKY 0.001574385 0.57641944 0.239754152 0.001388216 0.0054654972
:> McMULLEN  0.002175437 0.15219499 0.110137872 0.266486530 0.3892621478
```

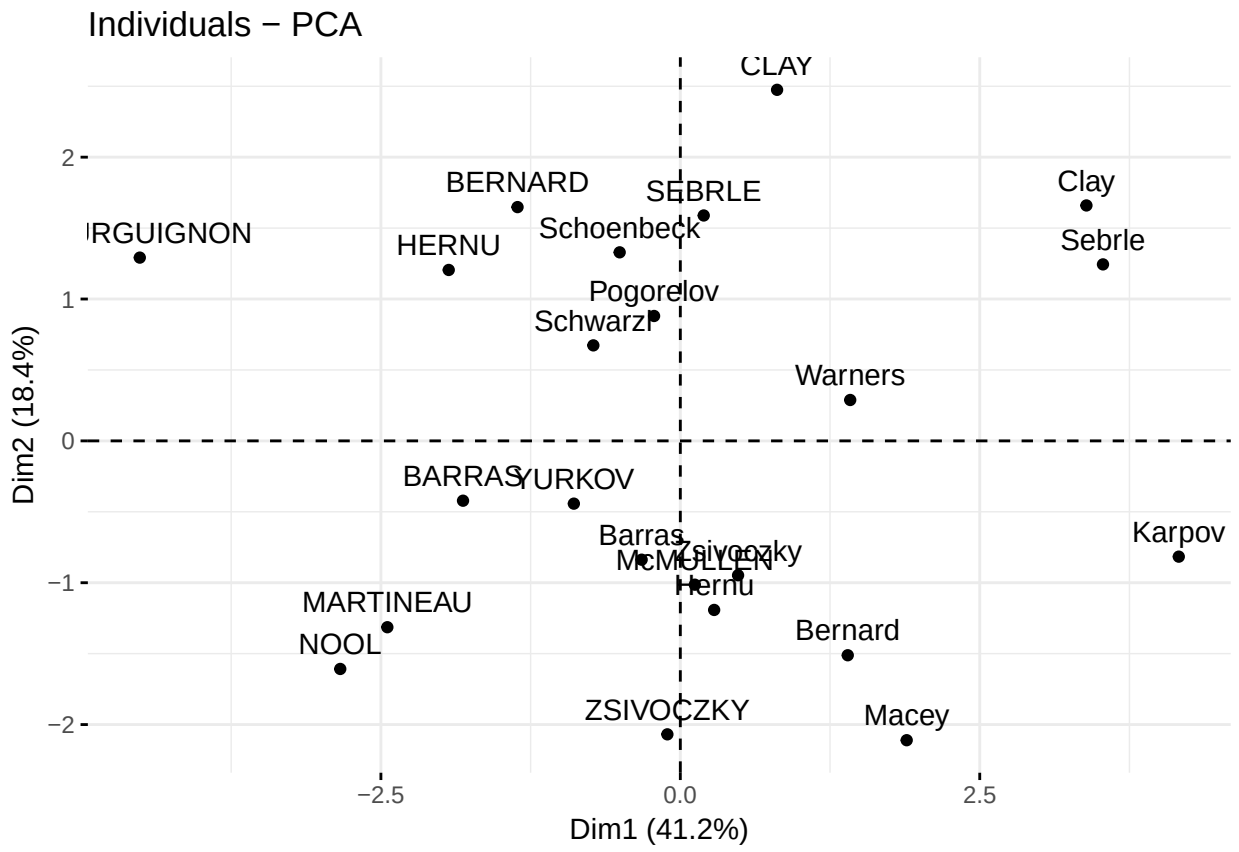
```
# Contributions of individuals
head(ind$contrib)
```

```
:>                Dim.1      Dim.2      Dim.3       Dim.4       Dim.5
:> SEBRLE    0.04029447  5.9714533  1.4483919  0.03734589  8.45894063
:> CLAY      0.68805664 14.4839248  6.7537381  8.94458283  4.21794385
:> BERNARD   1.94740183  6.4234107  0.1411345 20.46819433  0.04393073
:> YURKOV    0.83308415  0.4632733 22.4517396  2.69663605  1.03075263
:> ZSIVOCZKY 0.01232413 10.1217143  6.2464325  0.05469230  0.25151025
:> McMULLEN  0.01549089  2.4310854  2.6102794  9.55055888 16.29493304
```

4.11 Plots: quality and contribution

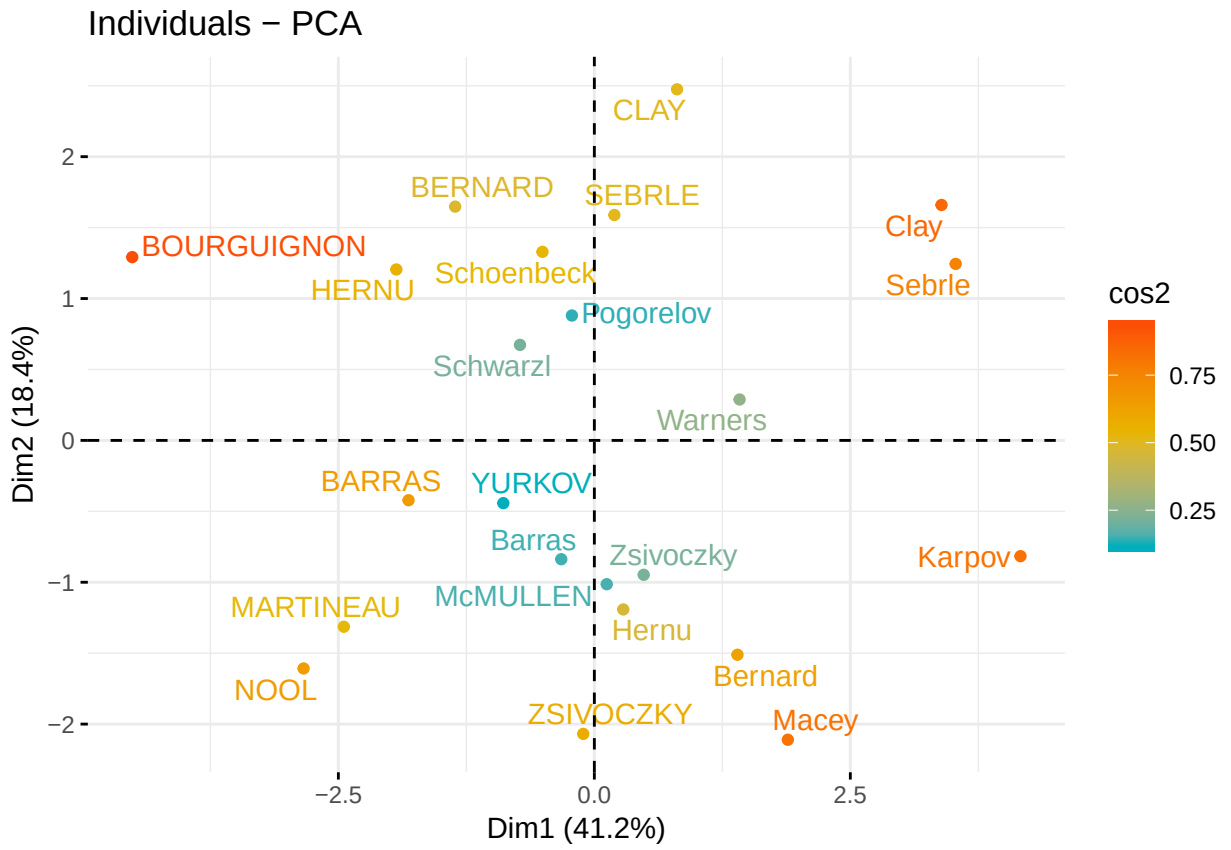
The fviz_pca_ind() is used to produce the graph of individuals. To create a simple plot, type this:

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



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

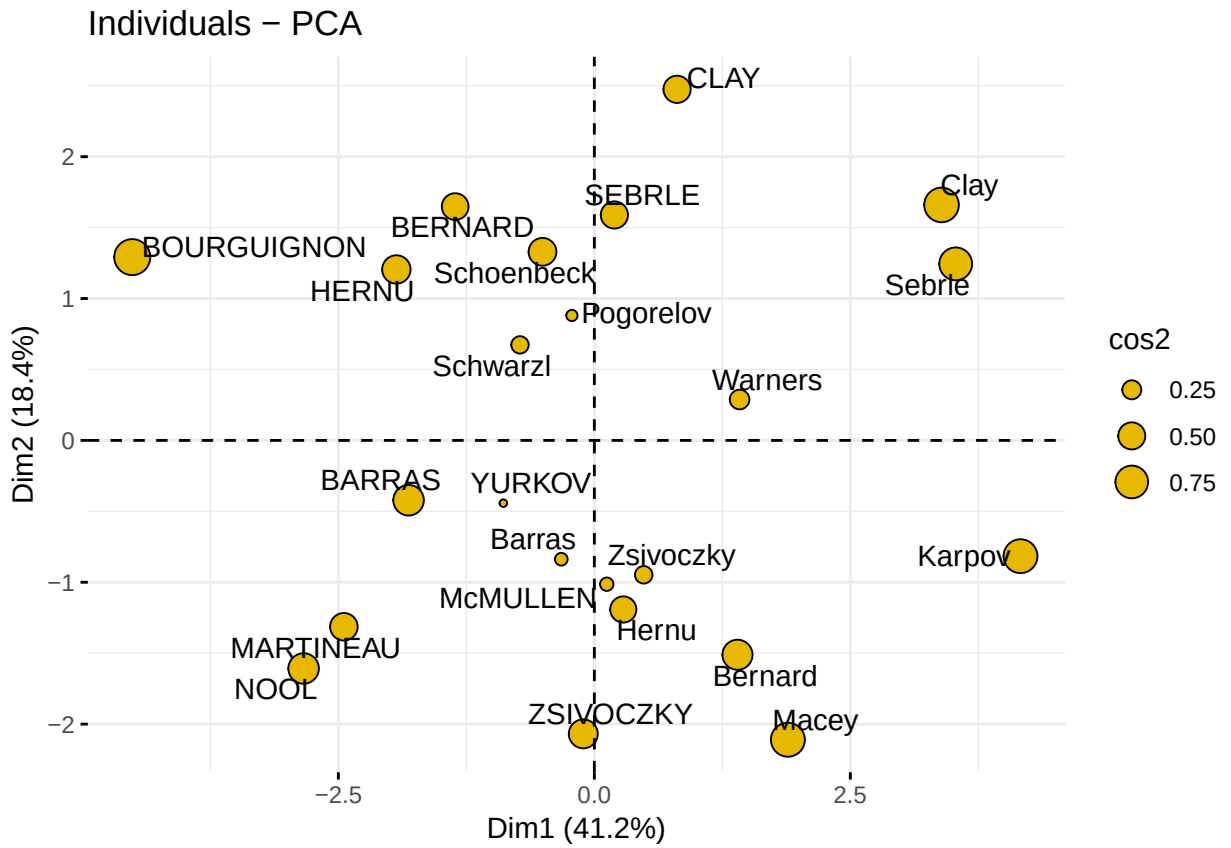
```
fviz_pca_ind(res.pca, col.ind = "cos2",
             gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"),
             repel = TRUE # Avoid text overlapping (slow if many points)
             )
```



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

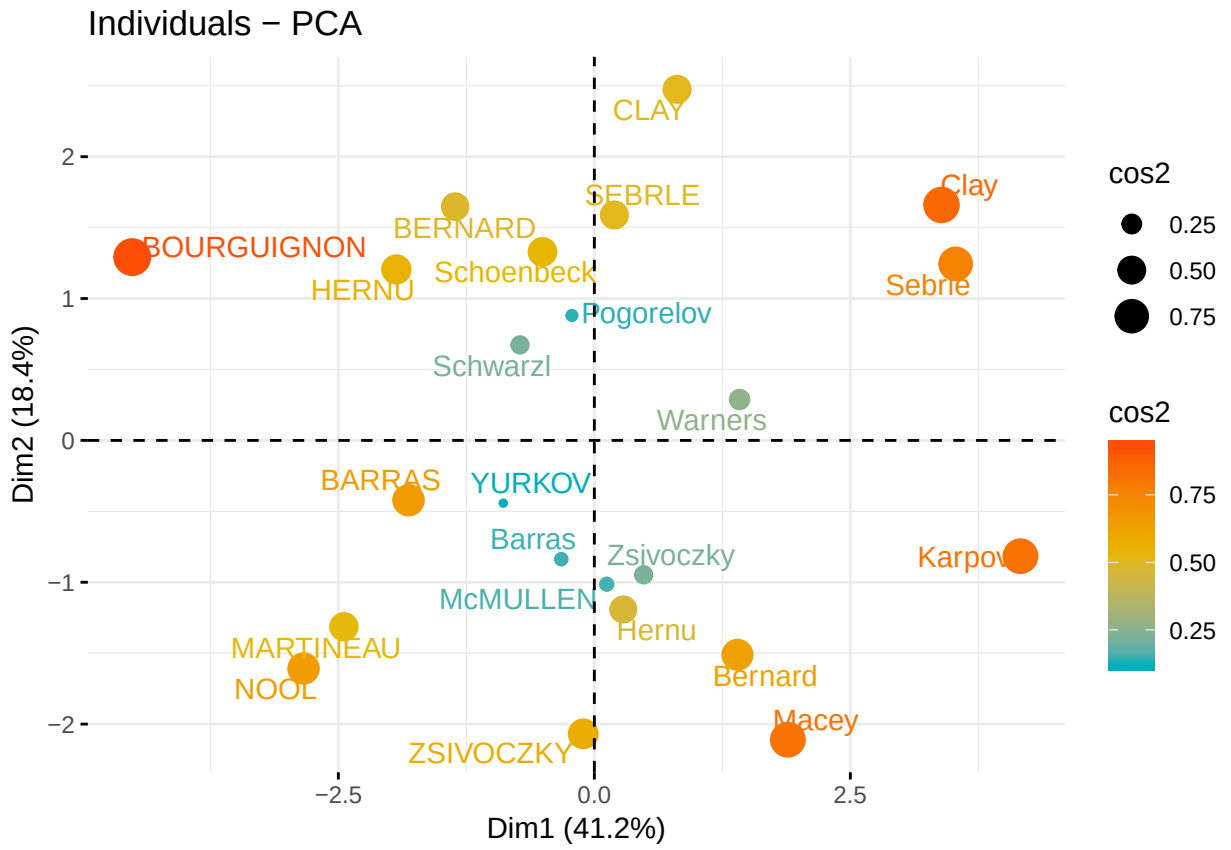
You can also change the point size according the cos2 of the corresponding individuals:

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



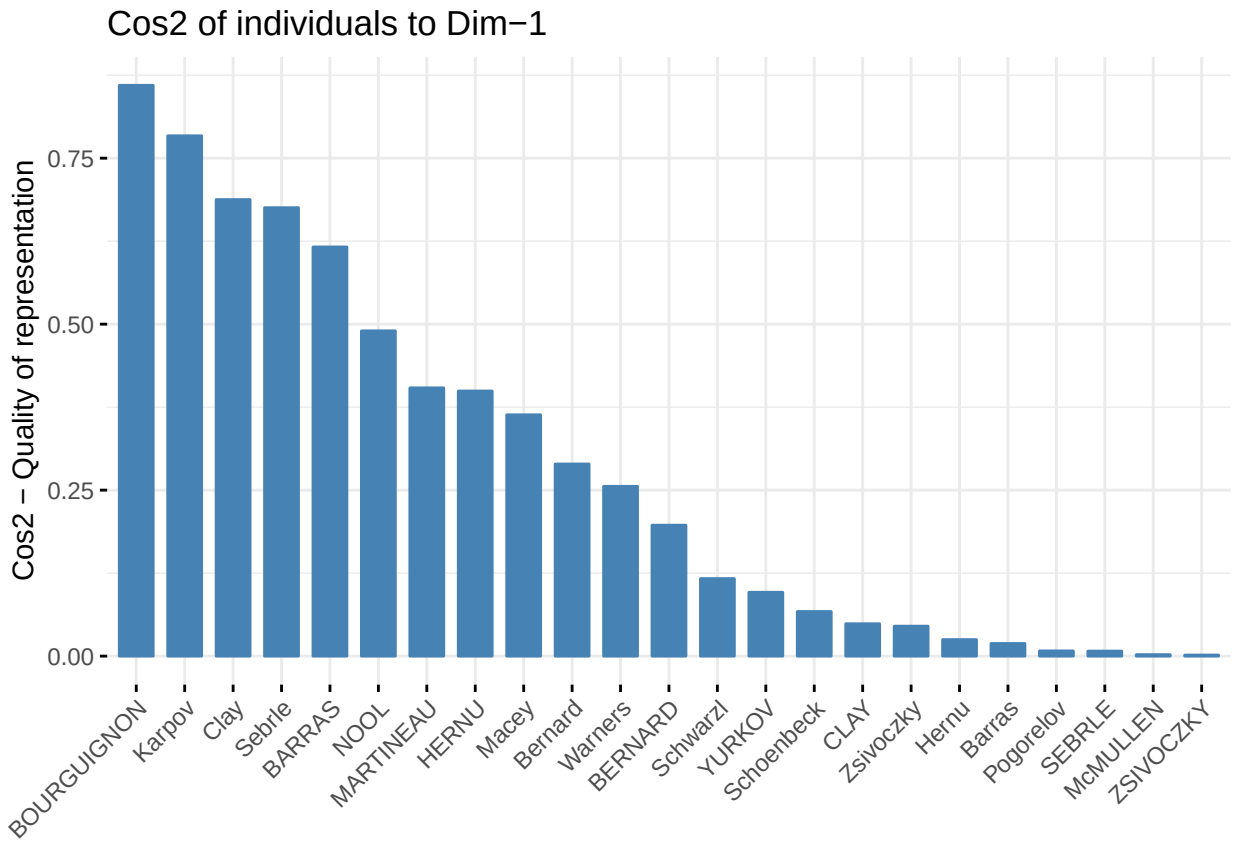
To change both point size and color by cos2, try this:

```
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)
             )
```



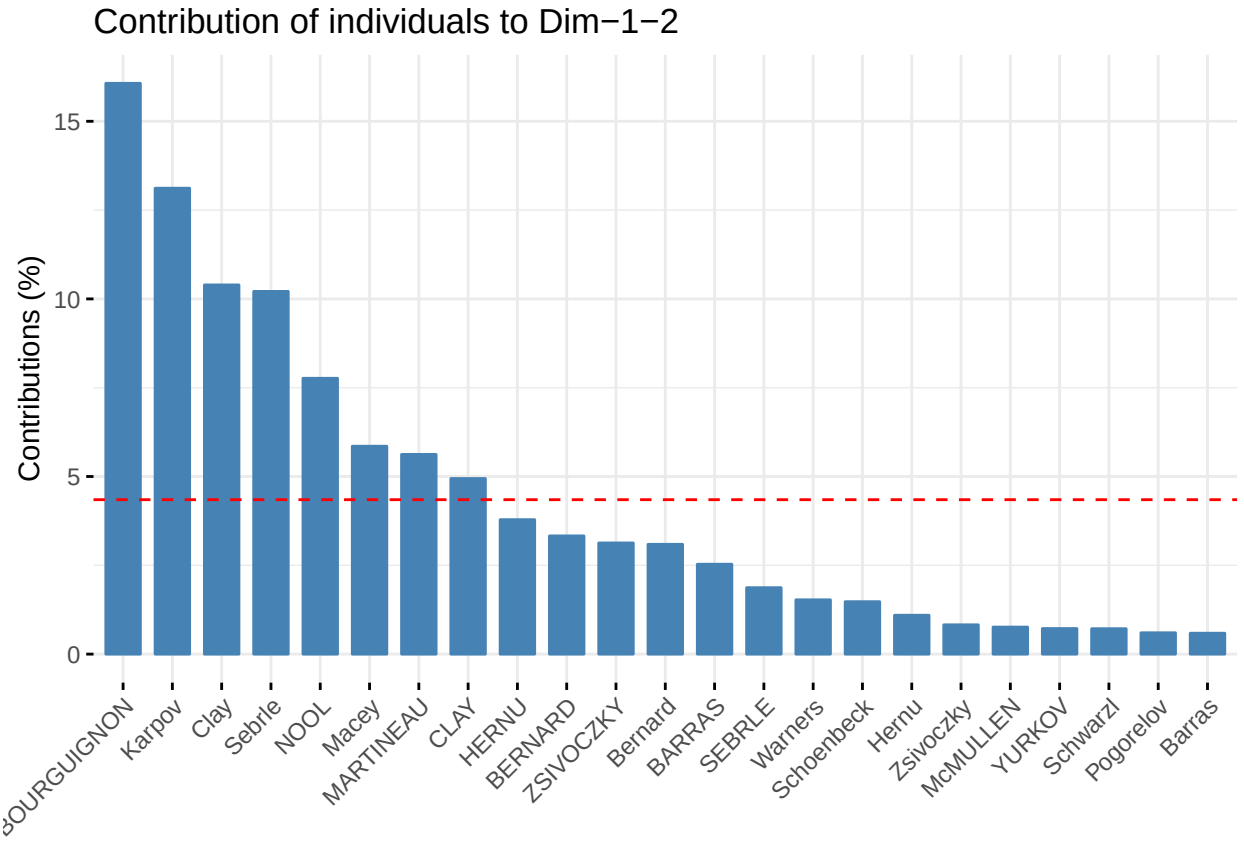
To create a bar plot of the quality of representation (cos2) of individuals on the factor map, you can use the function fviz_cos2() as previously described for variables:

```
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)
```

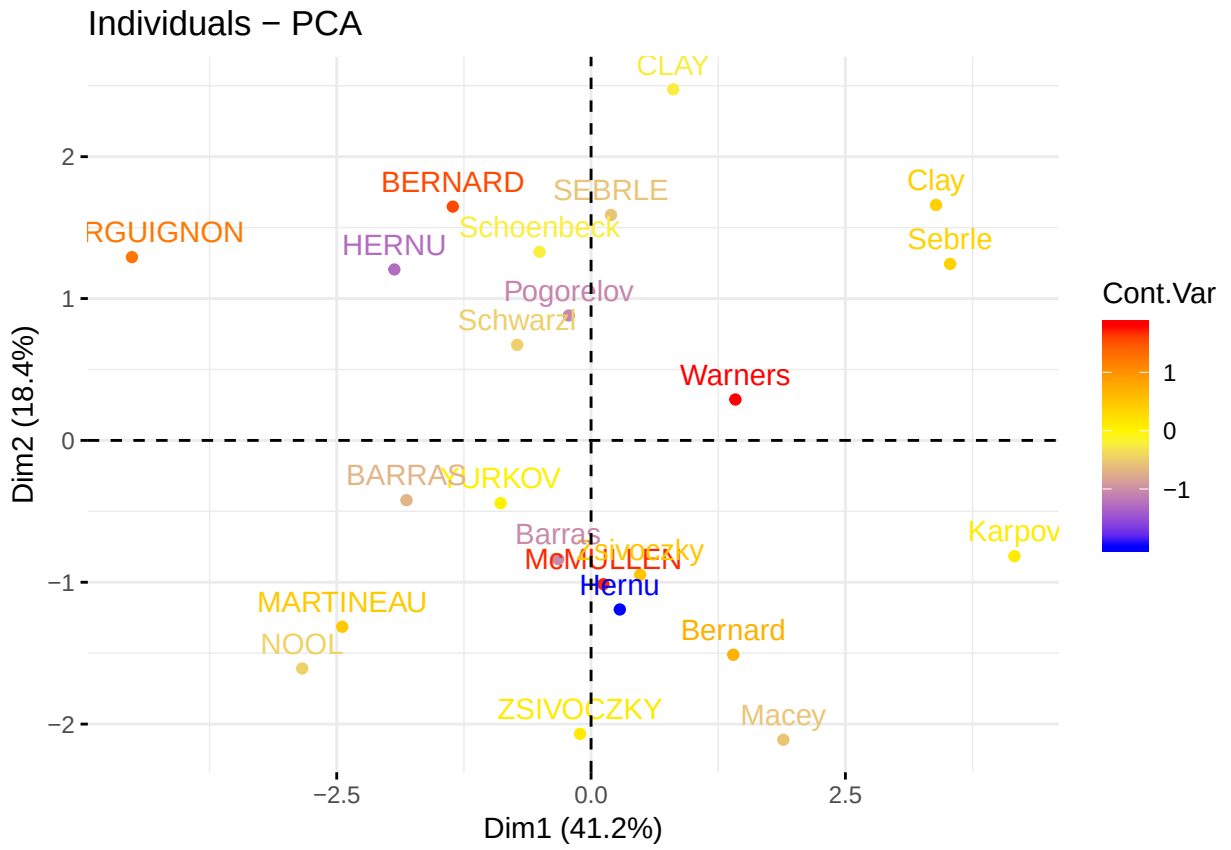


4.12 Color by a custom continuous variable

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")
```



4.13 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
```

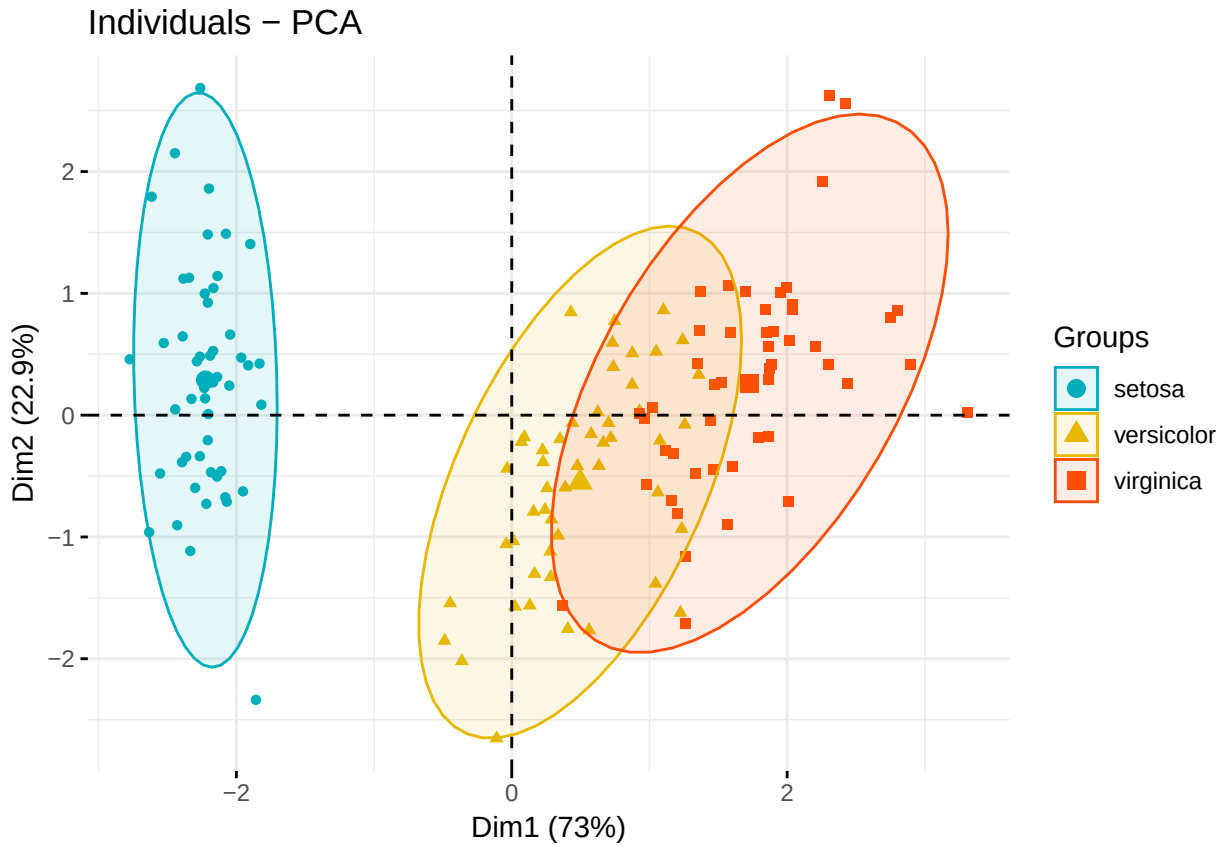
The column "Species" will be used as grouping variable. We start by computing principal component analysis as follow:

```
# 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.

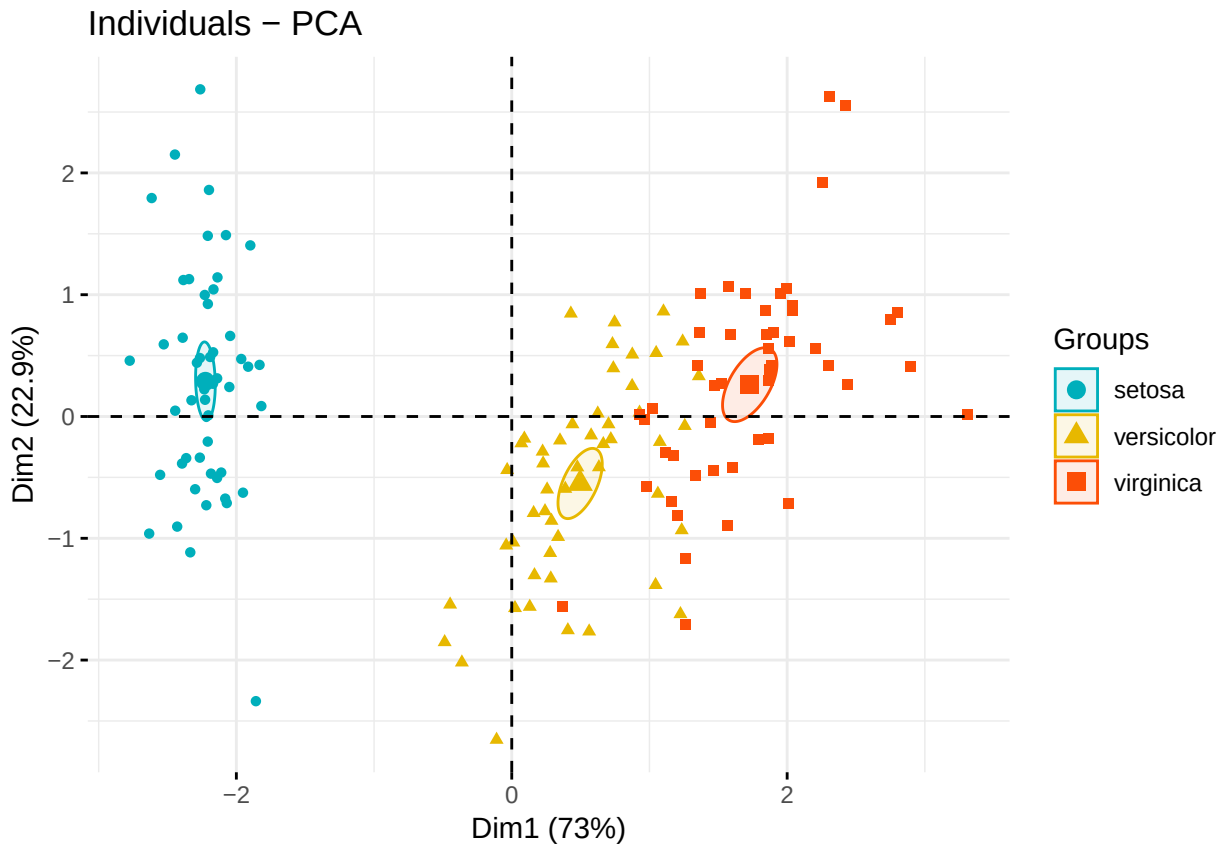
To add a concentration ellipse around each group, specify the argument addEllipses = TRUE. The argument palette can be used to change group colors.

```
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".

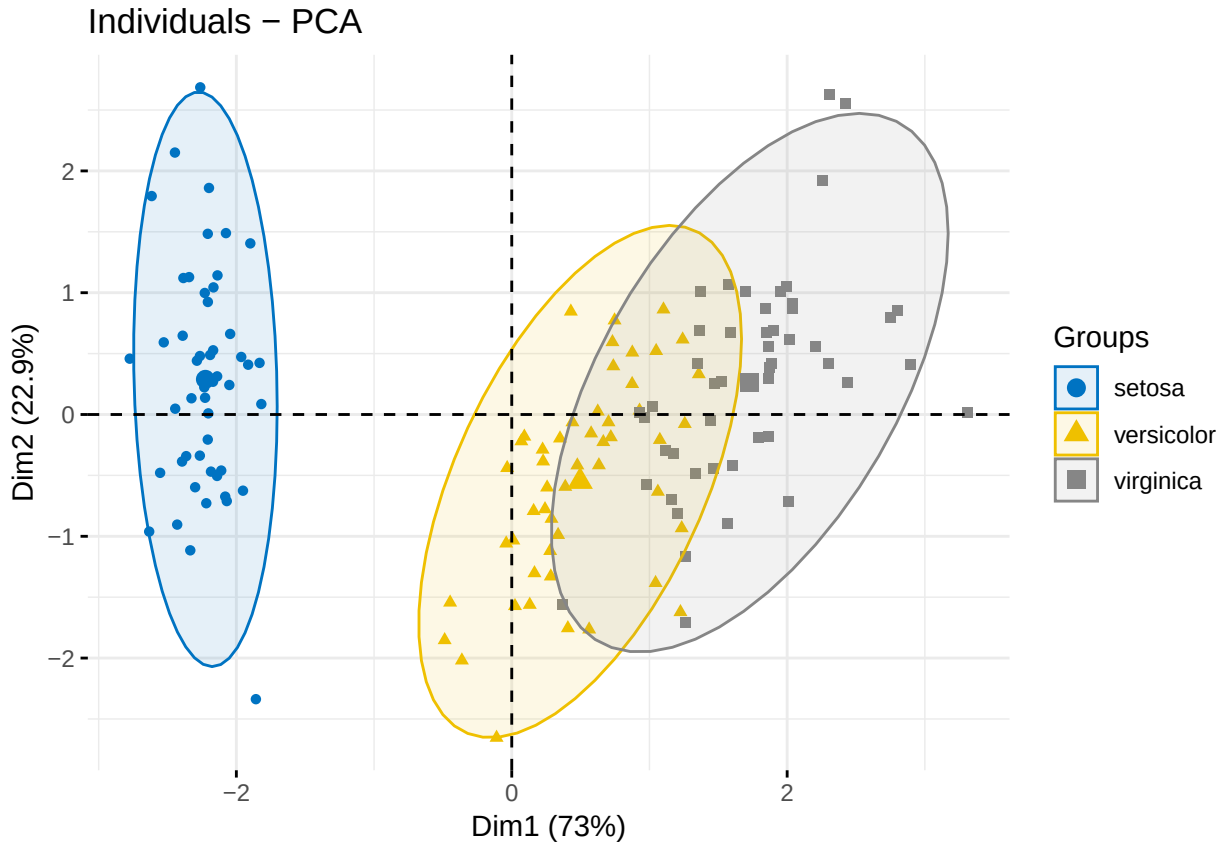
```
# Add confidence ellipses
fviz_pca_ind(iris.pca, geom.ind = "point", col.ind = iris$Species,
             palette = c("#00AFBB", "#E7B800", "#FC4E07"),
             addEllipses = TRUE, ellipse.type = "confidence",
             legend.title = "Groups"
             )
```



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"
             )
```



4.14 Graph customization

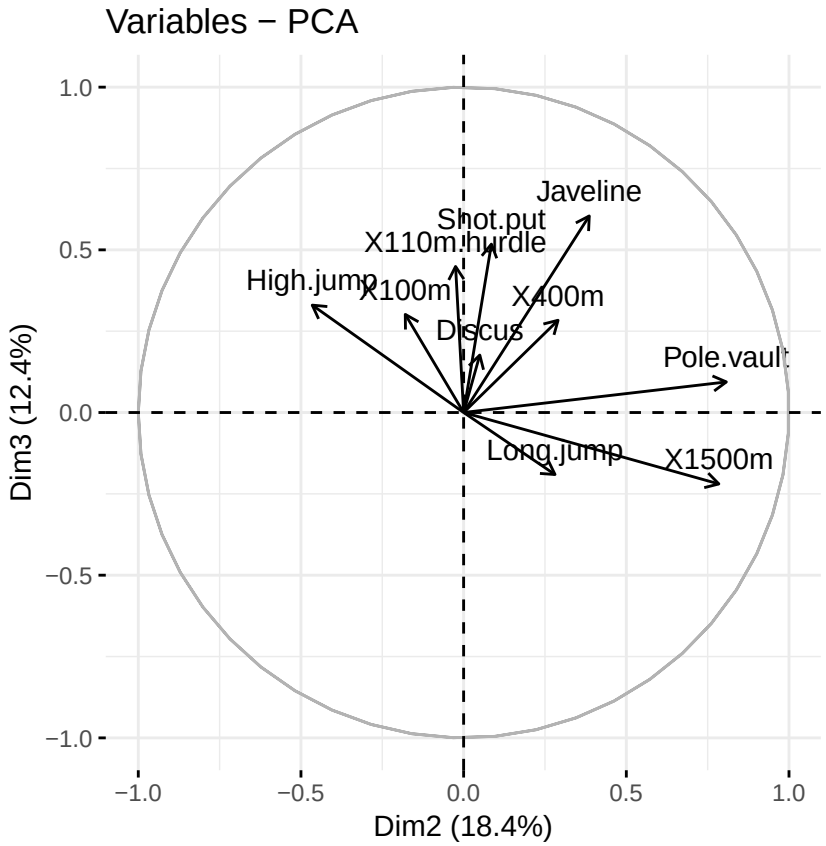
Note that, fviz_pca_ind() and fviz_pca_var() and related functions are wrapper around the core function fviz() [in factoextra]. fviz() is a wrapper around the function ggscatter() [in ggpubr]. Therefore, further arguments, to be passed to the function fviz() and ggscatter(), can be specified in fviz_pca_ind()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

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

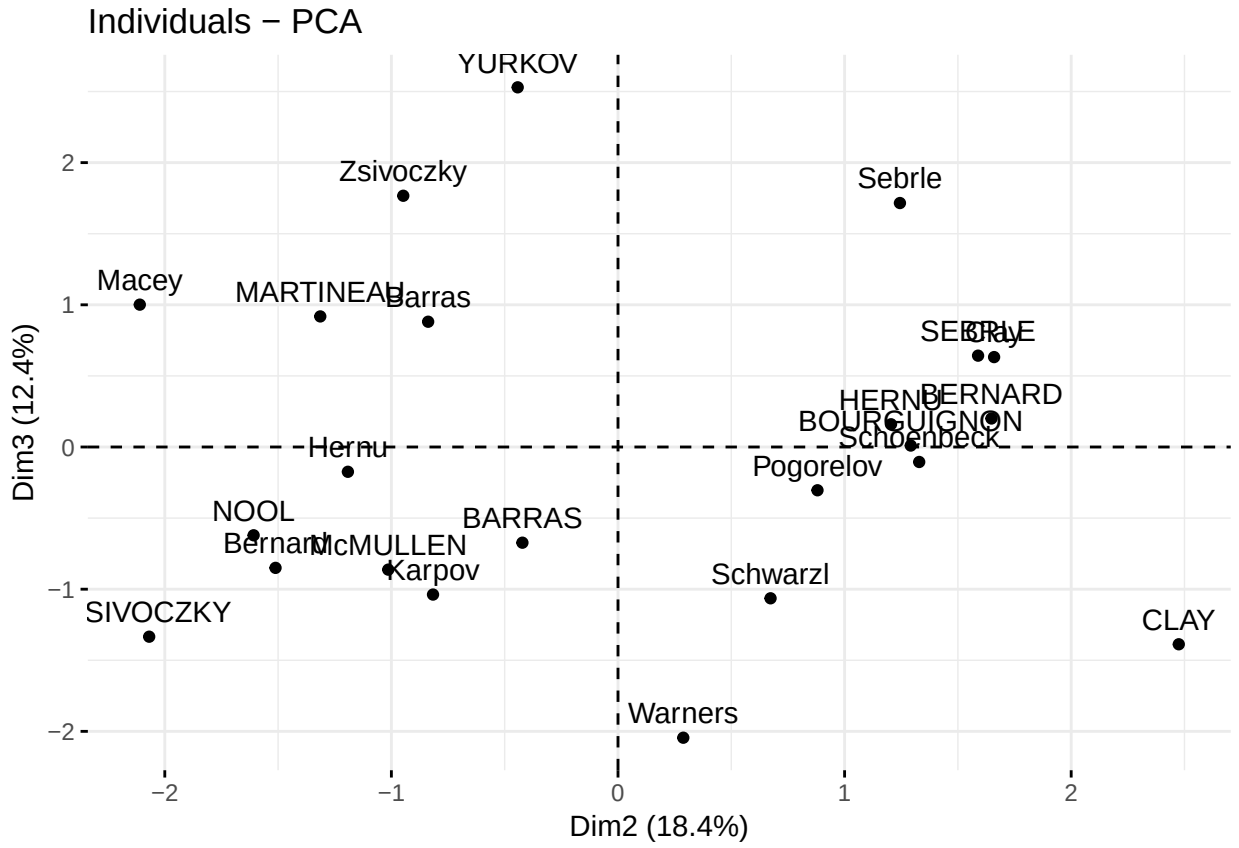
4.14.1 Dimensions

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))
```



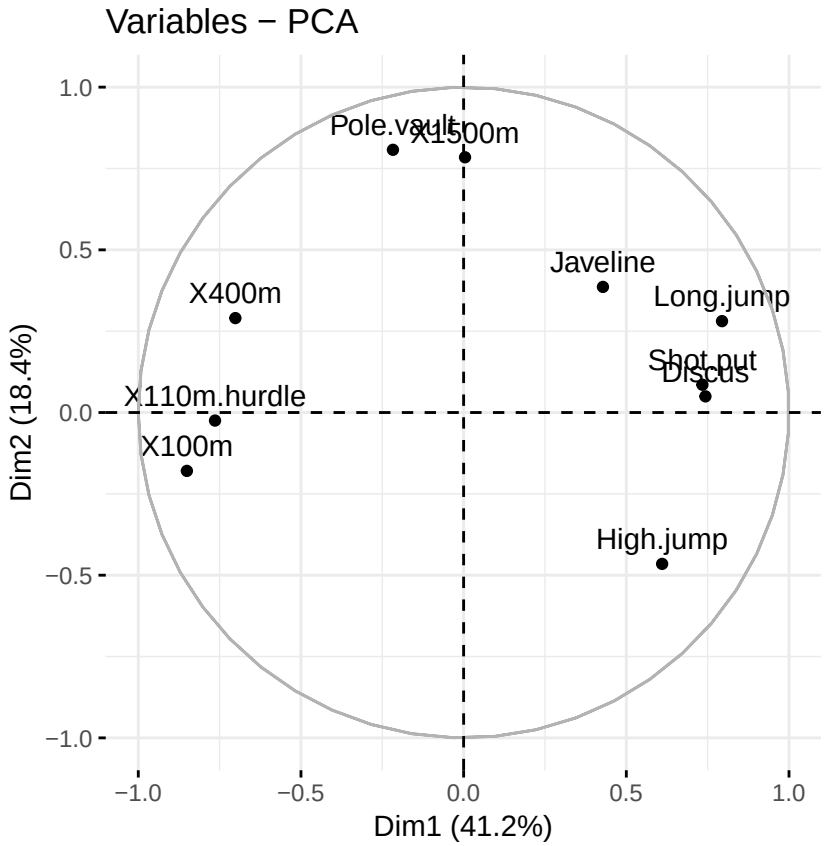
Plot elements: point, text, arrow The argument geom (for geometry) and derivatives are used to specify the geometry elements or graphical elements to be used for plotting.

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

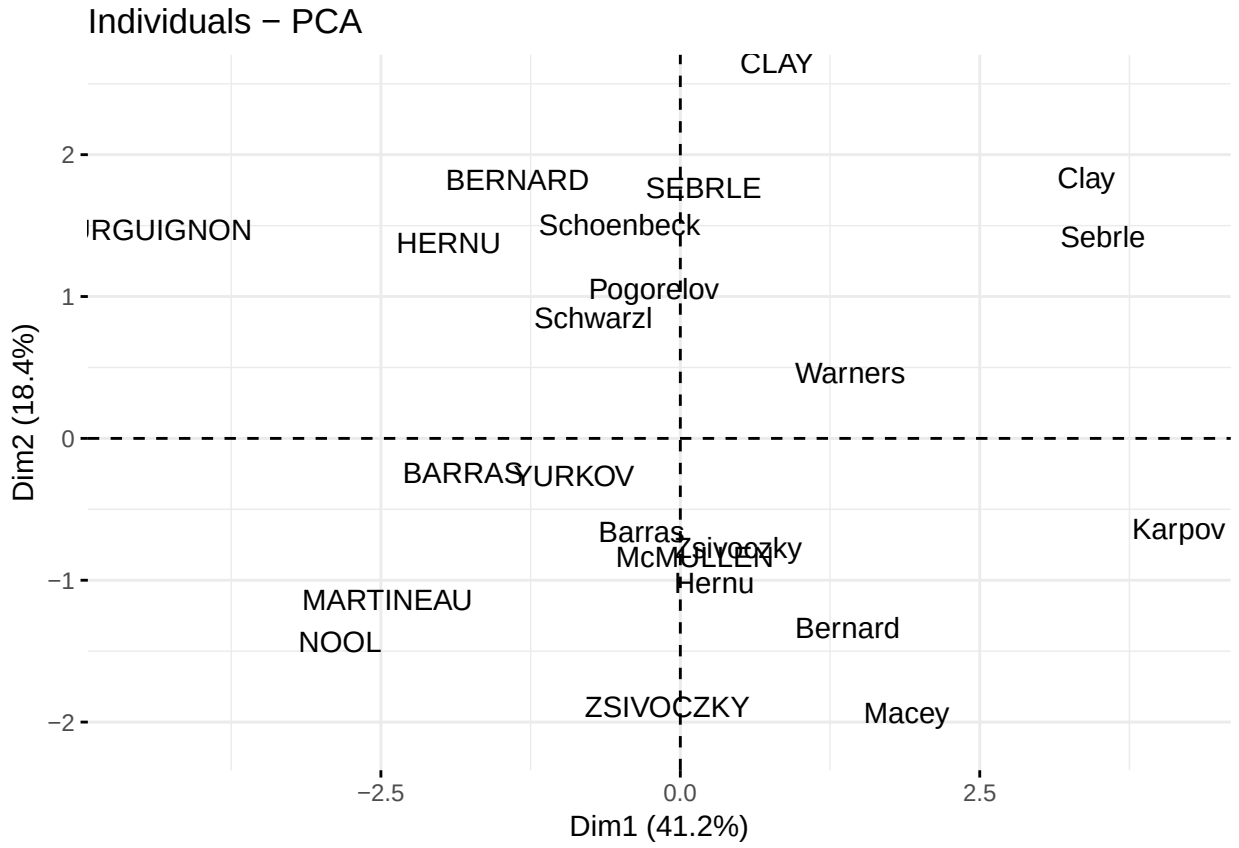
- Use geom.var = "point", to show only points;
- Use geom.var = "text" to show only text labels;
- Use geom.var = c("point", "text") to show both points and text labels
- Use geom.var = c("arrow", "text") to show arrows and labels (default).

For example, type this:

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

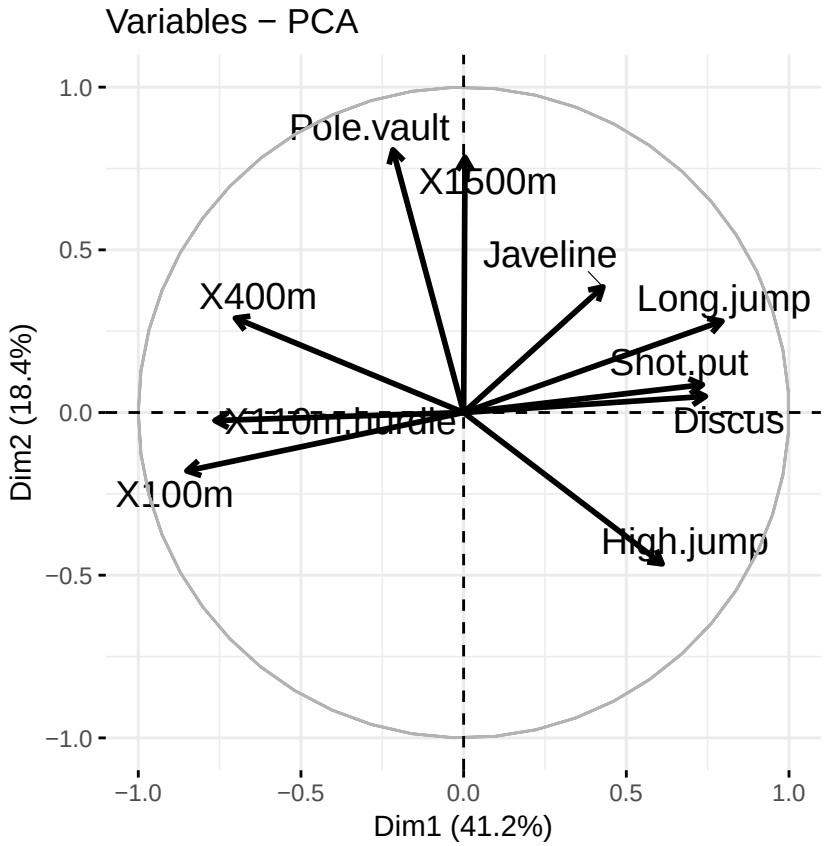



```
# Show individuals text labels only
fviz_pca_ind(res.pca, geom.ind =  "text")
```

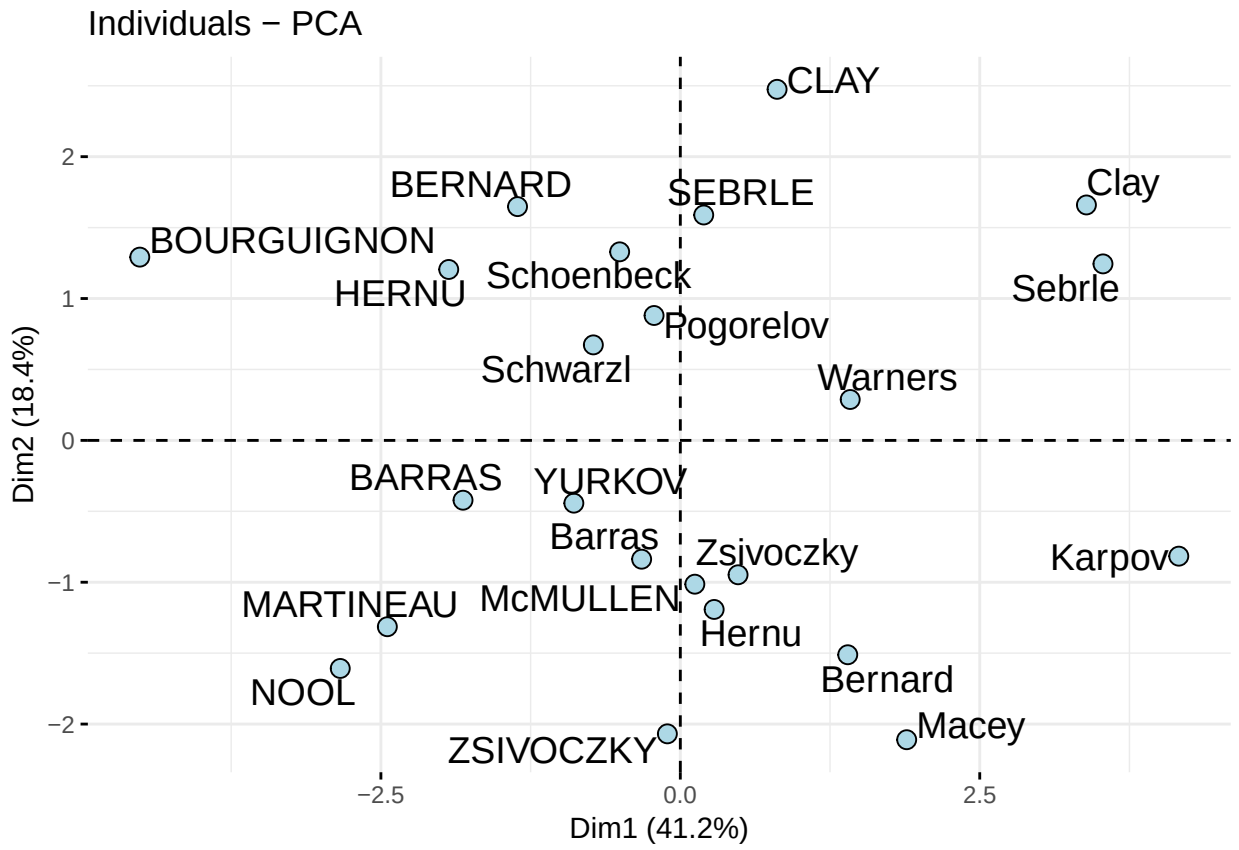


4.15 Size and shape of plot elements

```
# Change the size of arrows an labels
fviz_pca_var(res.pca, arrowsize = 1, labelsize = 5,
             repel = TRUE)
```

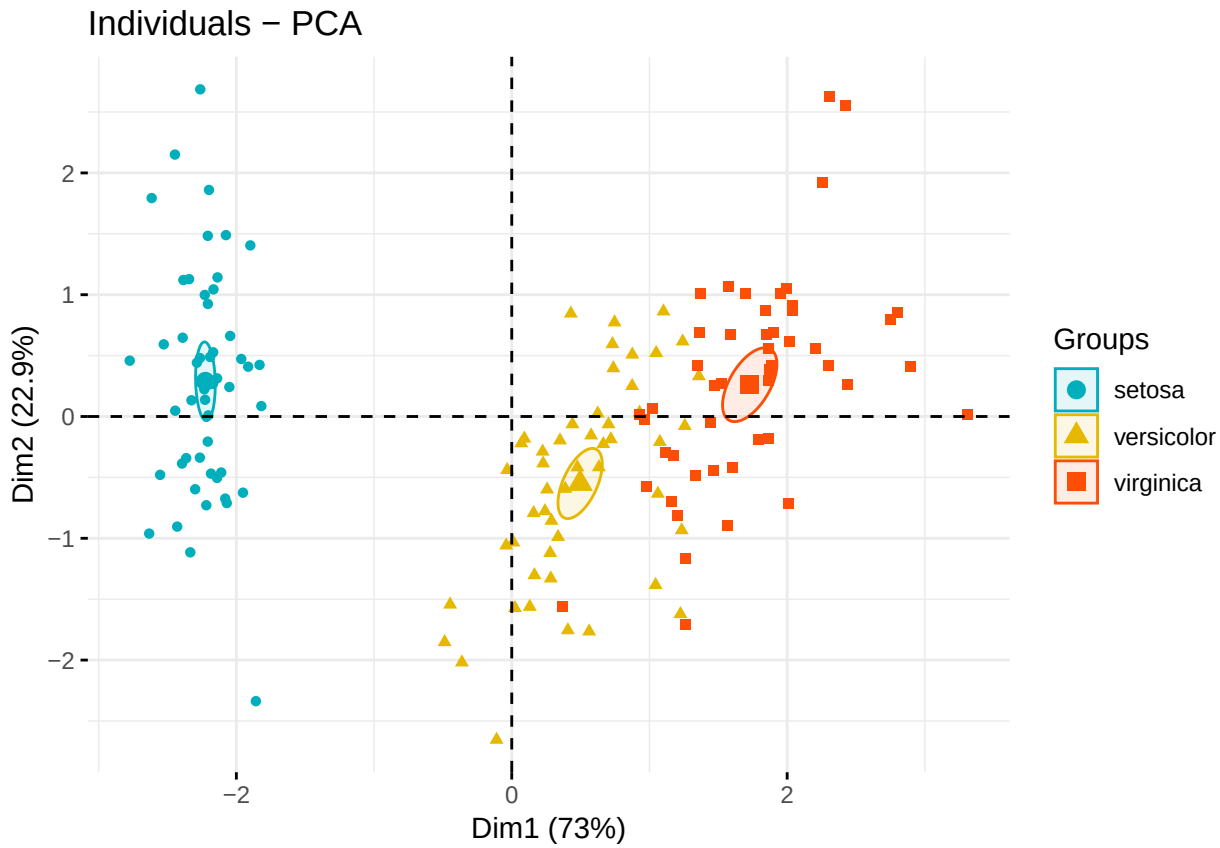



```
# Change points size, shape and fill color
# Change labelsize
fviz_pca_ind(res.pca,
             pointsize = 3, pointshape = 21, fill = "lightblue",
             labelsize = 5, repel = TRUE)
```

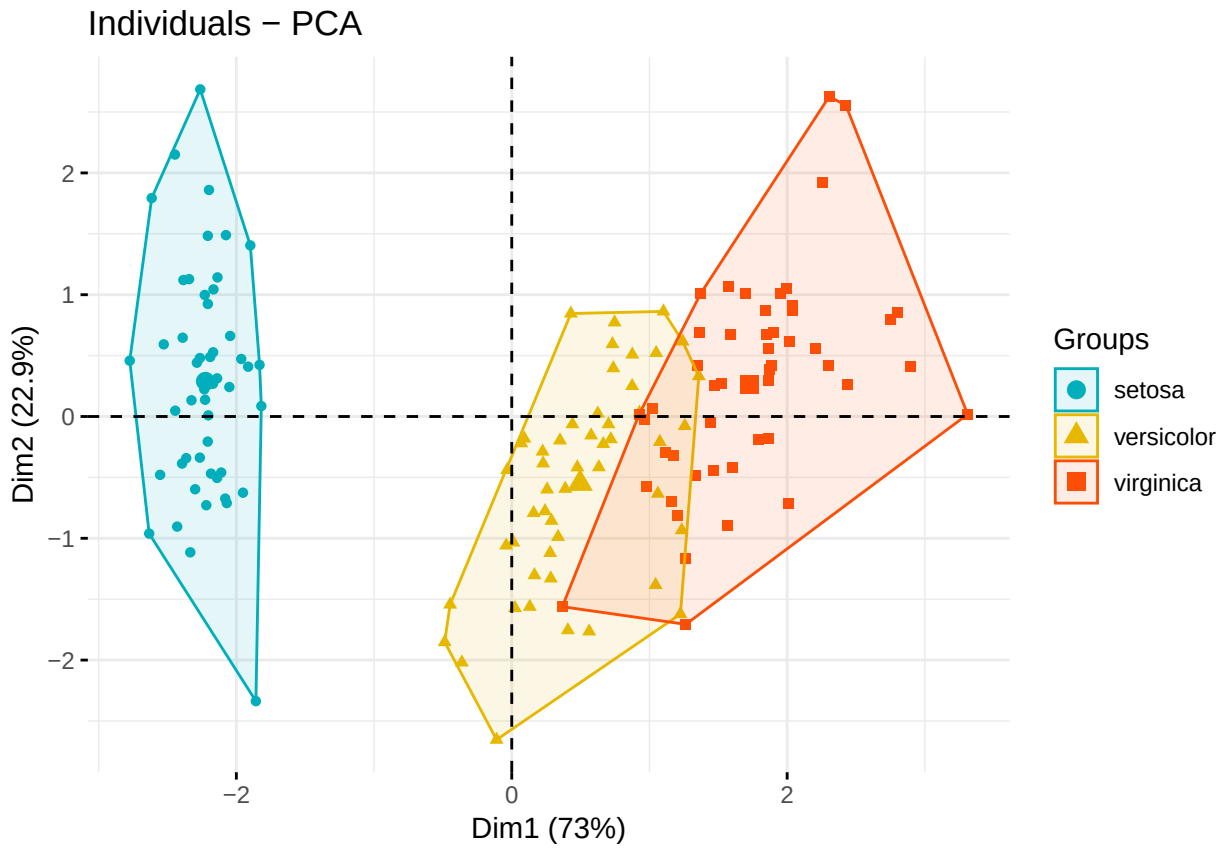


4.16 Ellipses

```
# 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"
             )
```

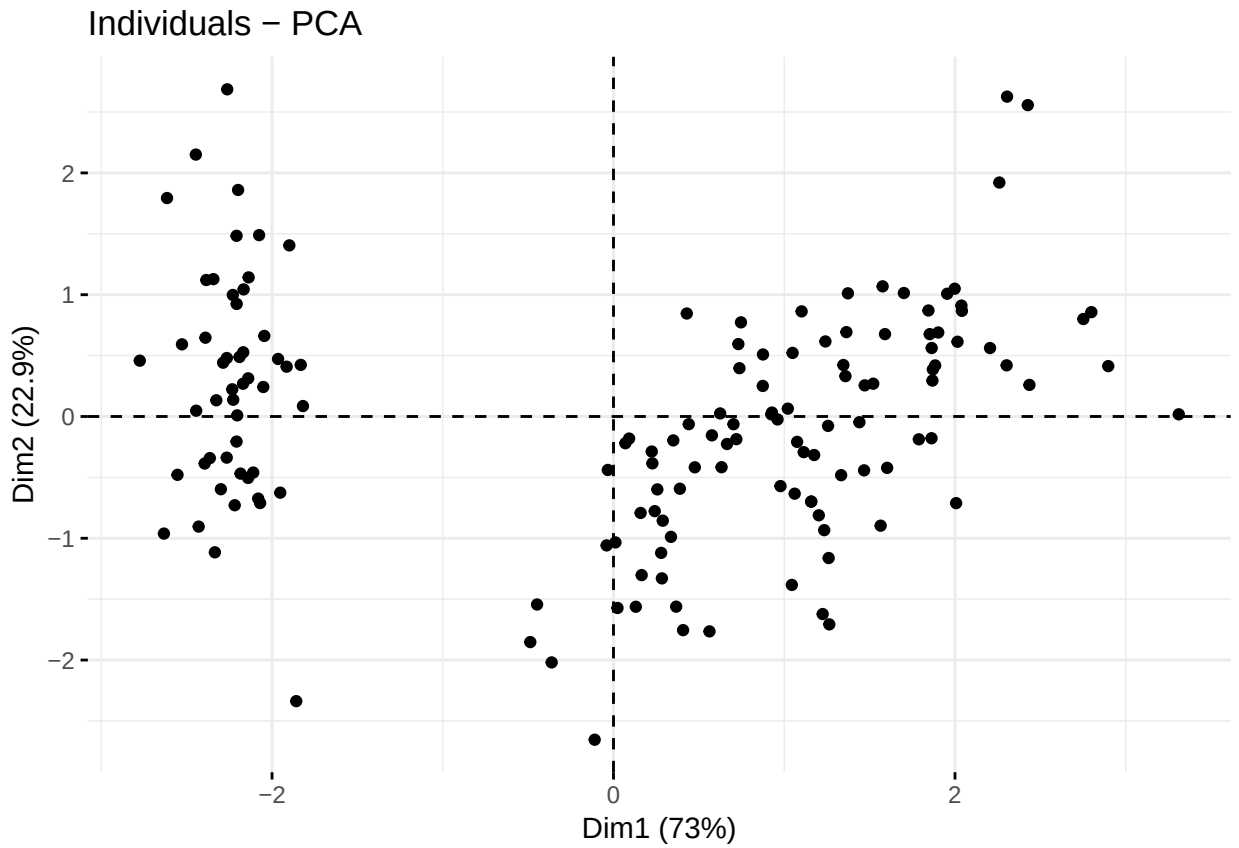


4.17 Group mean points

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

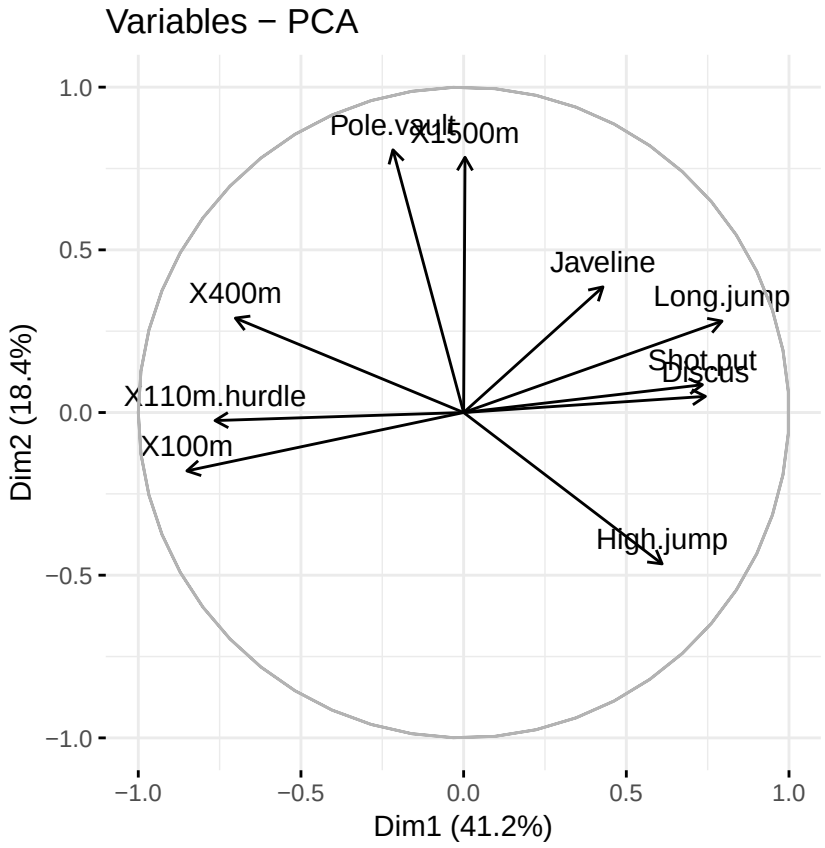
To remove the mean points, use the argument mean.point = FALSE.

```
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)
```



4.18 Axis lines

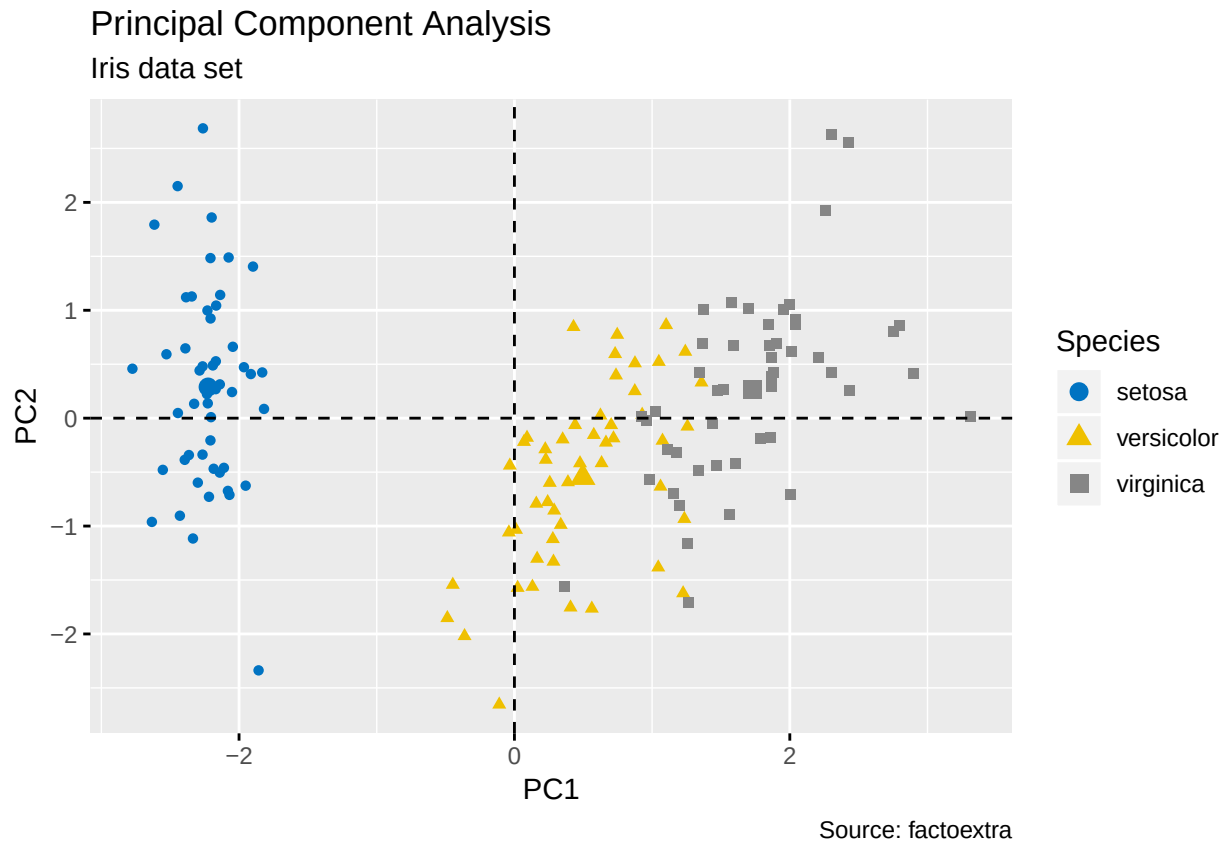
```
fviz_pca_var(res.pca, axes.linetype = "blank")
```



4.19 Graphical parameters

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

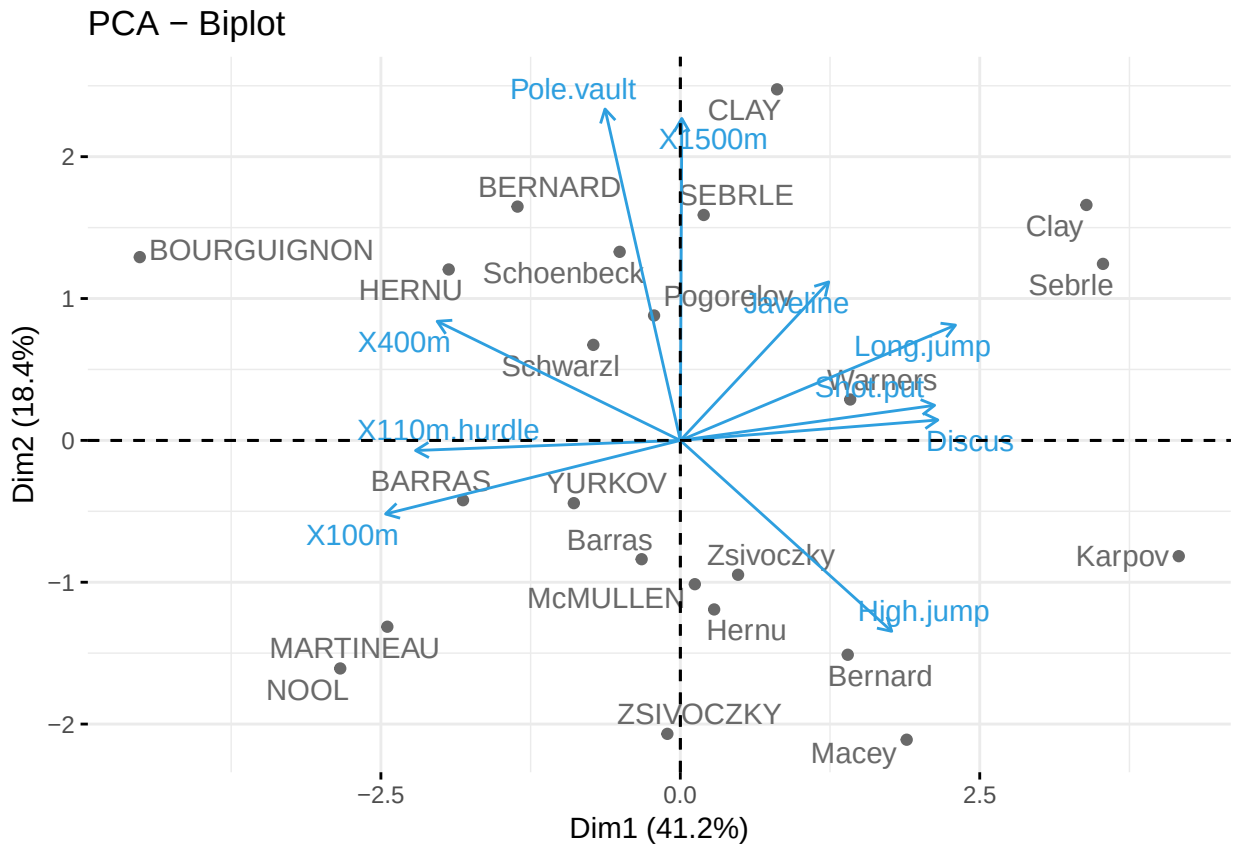
```
ind.p <- fviz_pca_ind(iris.pca, geom = "point", col.ind = iris$Species)
ggpubr::ggpar(ind.p,
              title = "Principal Component Analysis",
              subtitle = "Iris data set",
              caption = "Source: factoextra",
              xlab = "PC1", ylab = "PC2",
              legend.title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



4.20 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
                )
```

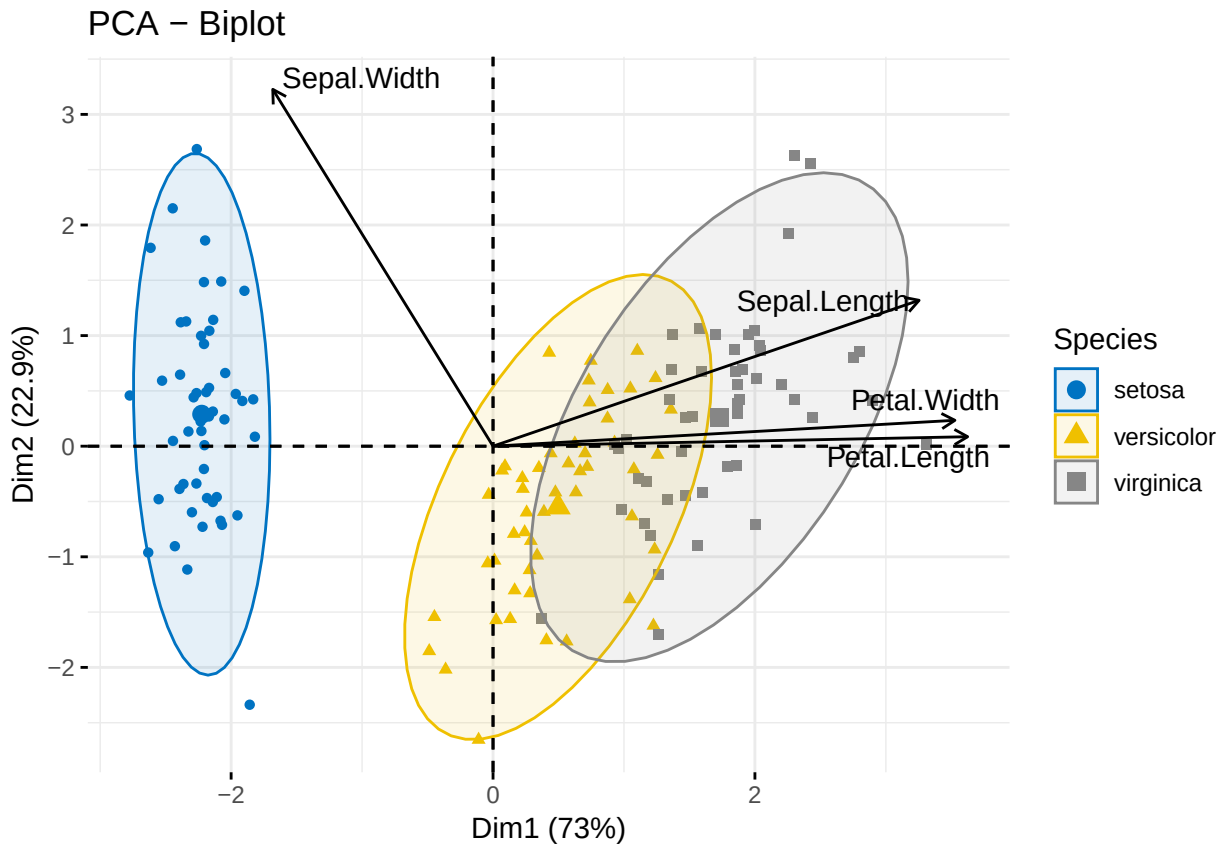


Note that, the biplot might be only useful when there is a low number of variables and individuals in the data set; otherwise the final plot would be unreadable.

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.

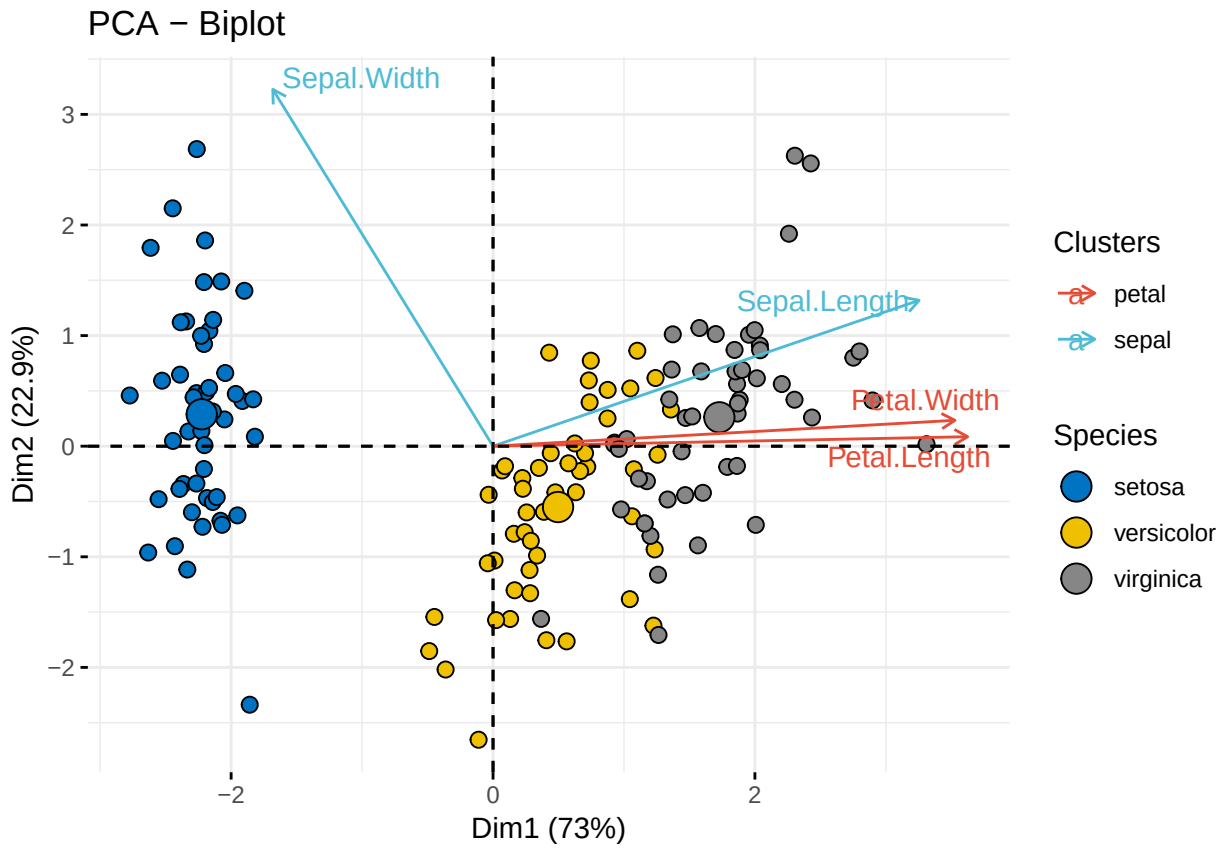
```
fviz_pca_biplot(iris.pca,
                col.ind = iris$Species, palette = "jco",
                addEllipses = TRUE, label = "var",
                col.var = "black", repel = TRUE,
                legend.title = "Species")
```



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.

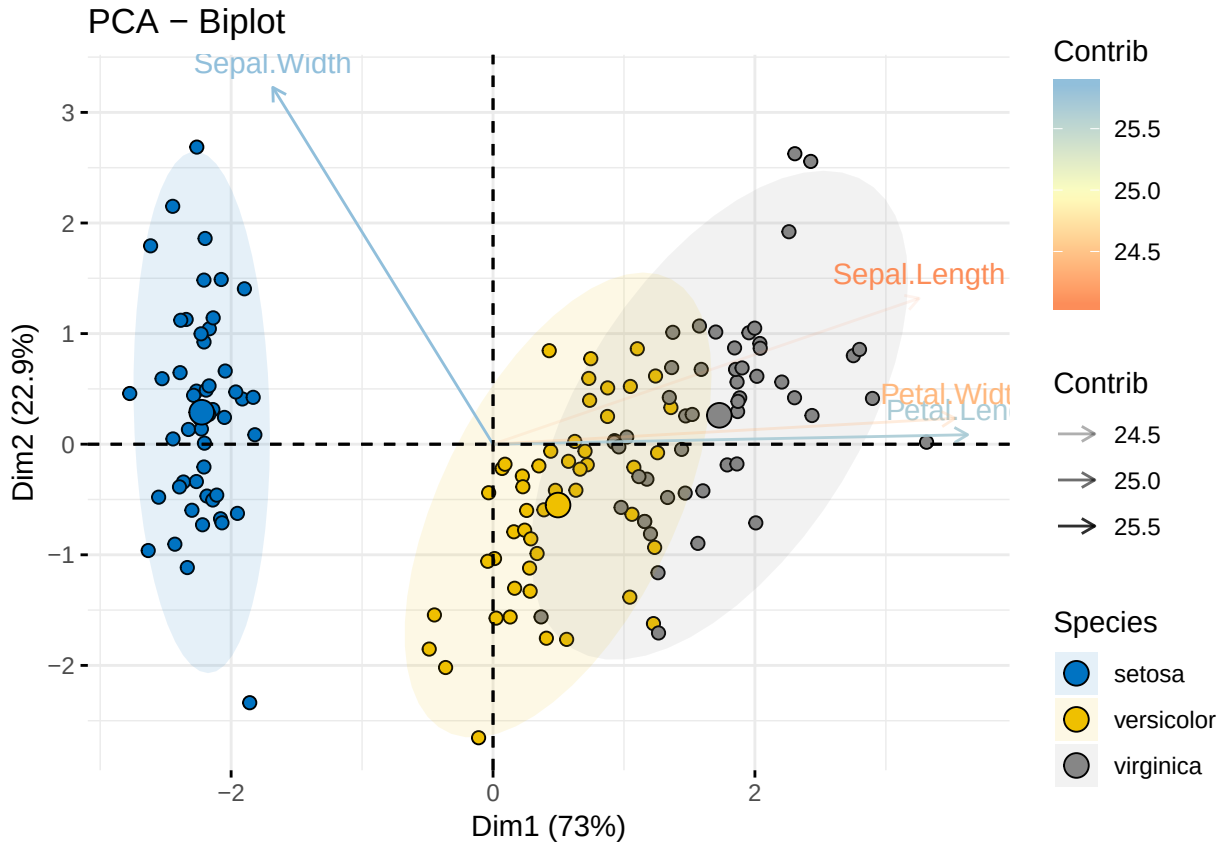
To customize individuals and variable colors, we use the helper functions fill_palette() and color_palette() [in ggpubr package].

```
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")
                )
```



4.21 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)
```

4.22 Quantitative variables

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

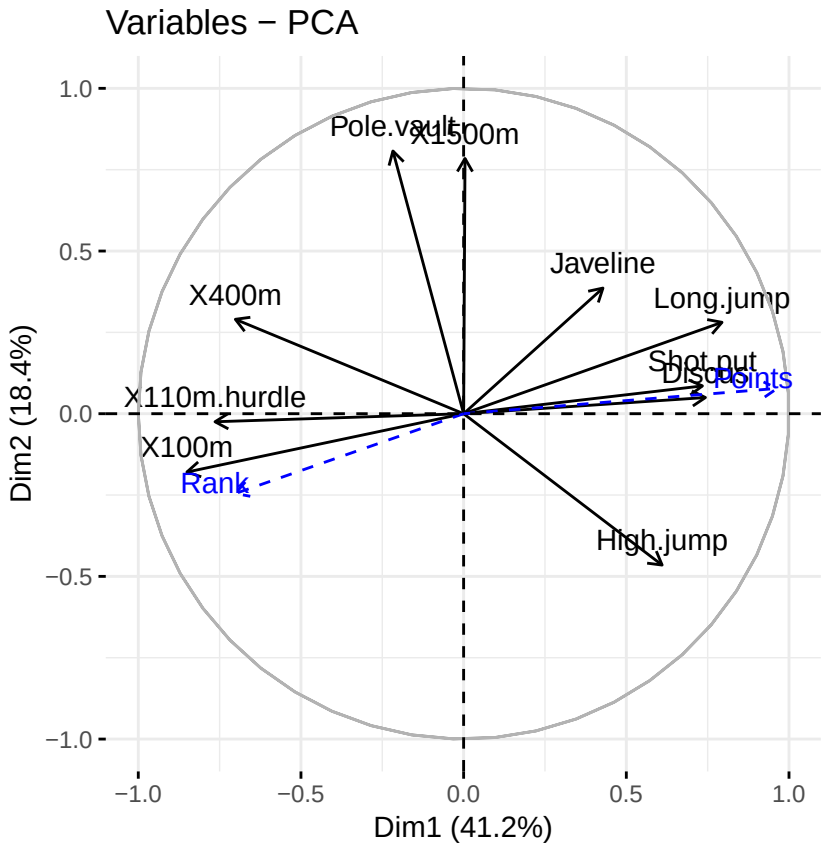
```
res.pca$quanti.sup
```

```
:> $coord
:>             Dim.1       Dim.2      Dim.3       Dim.4       Dim.5
:> Rank   -0.7014777 -0.24519443 -0.1834294  0.05575186 -0.07382647
```

```
:> Points  0.9637075  0.07768262  0.1580225 -0.16623092 -0.03114711
:>
:> $cor
:>             Dim.1       Dim.2      Dim.3       Dim.4       Dim.5
:> Rank   -0.7014777 -0.24519443 -0.1834294  0.05575186 -0.07382647
:> Points  0.9637075  0.07768262  0.1580225 -0.16623092 -0.03114711
:>
:> $cos2
:>            Dim.1       Dim.2      Dim.3      Dim.4        Dim.5
:> Rank   0.4920710 0.060120310 0.03364635 0.00310827 0.0054503477
:> Points 0.9287322 0.006034589 0.02497110 0.02763272 0.0009701427
```

Visualize all variables (active and supplementary ones):

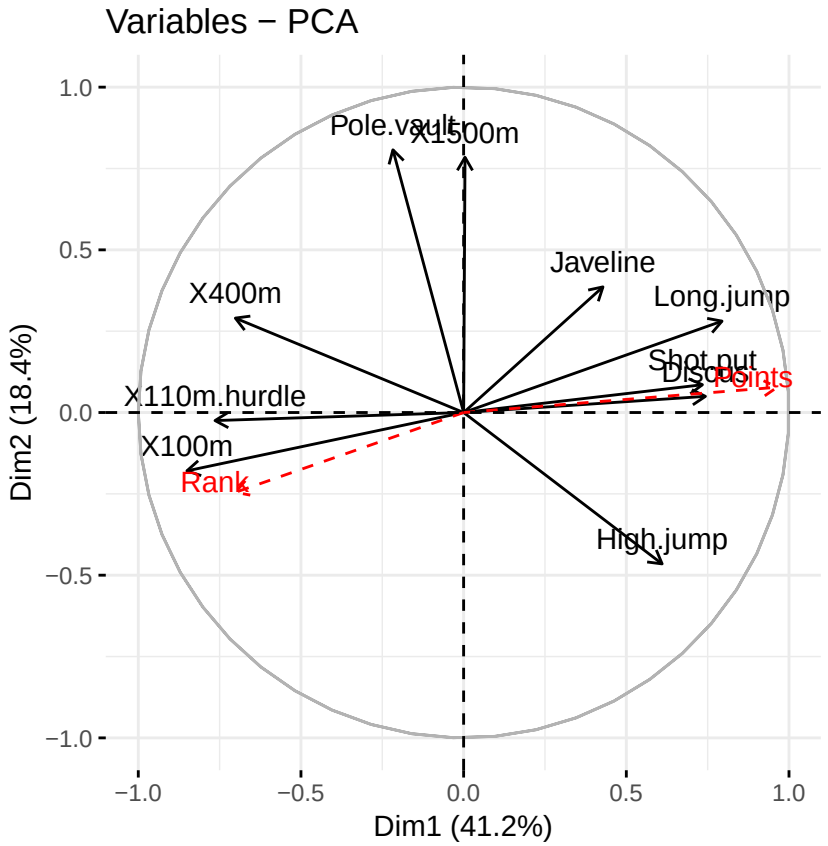
```
fviz_pca_var(res.pca)
```



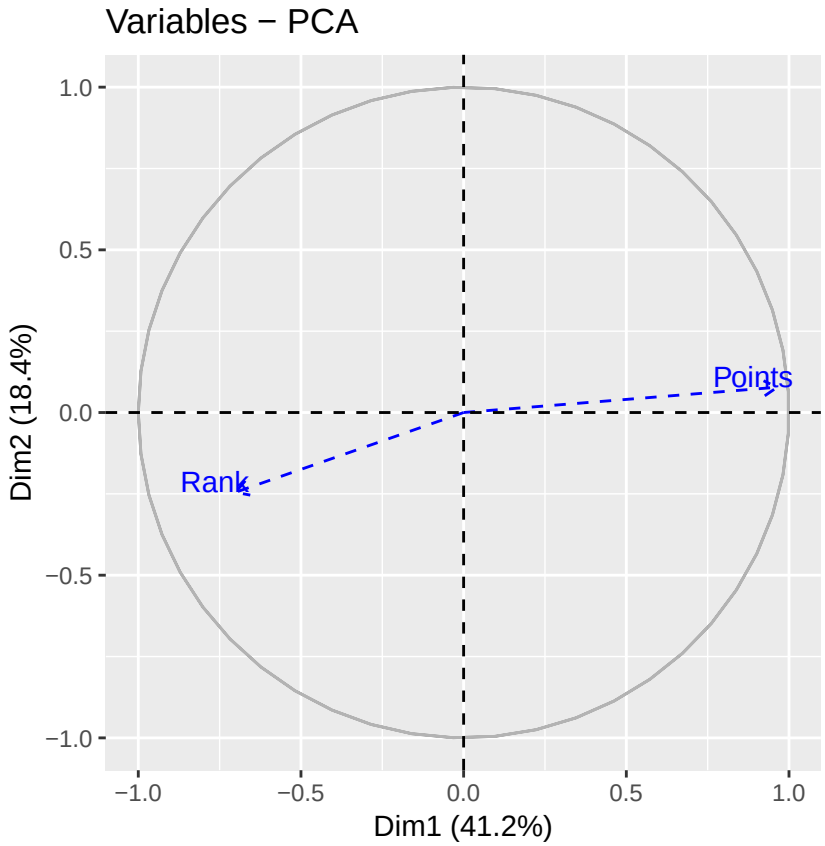
Note that, by default, supplementary quantitative variables are shown in blue color and dashed lines.

Further arguments to customize the plot:

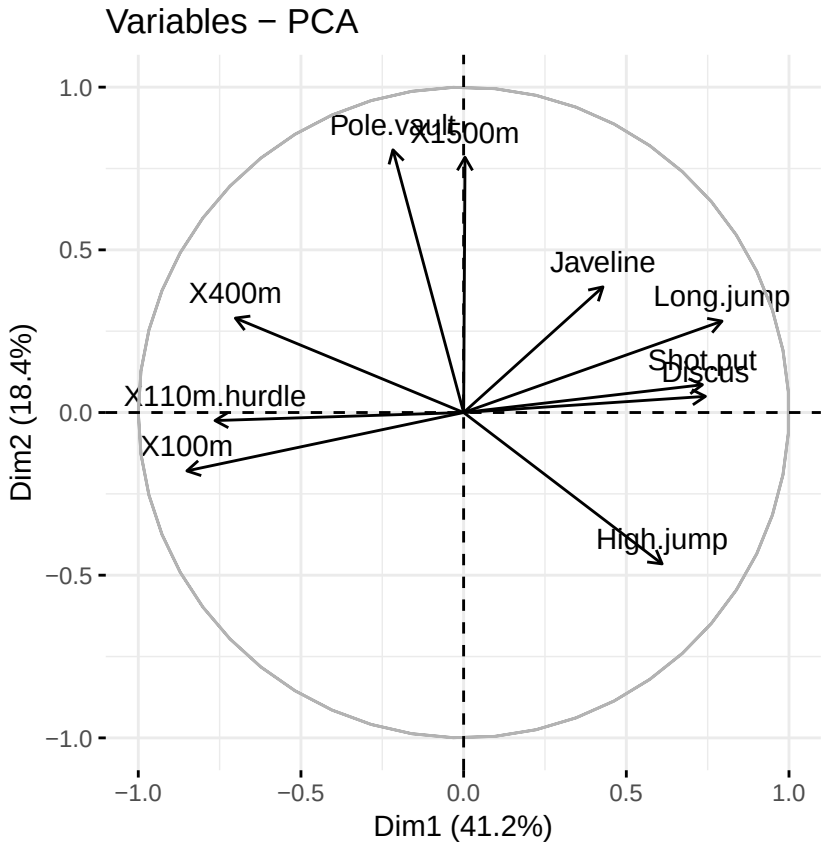
```
# Change color of variables
fviz_pca_var(res.pca,
             col.var = "black",      # Active variables
             col.quanti.sup = "red" # Suppl. quantitative variables
             )
```




```
# Hide active variables on the plot,
# show only supplementary variables
fviz_pca_var(res.pca, invisible = "var")
```

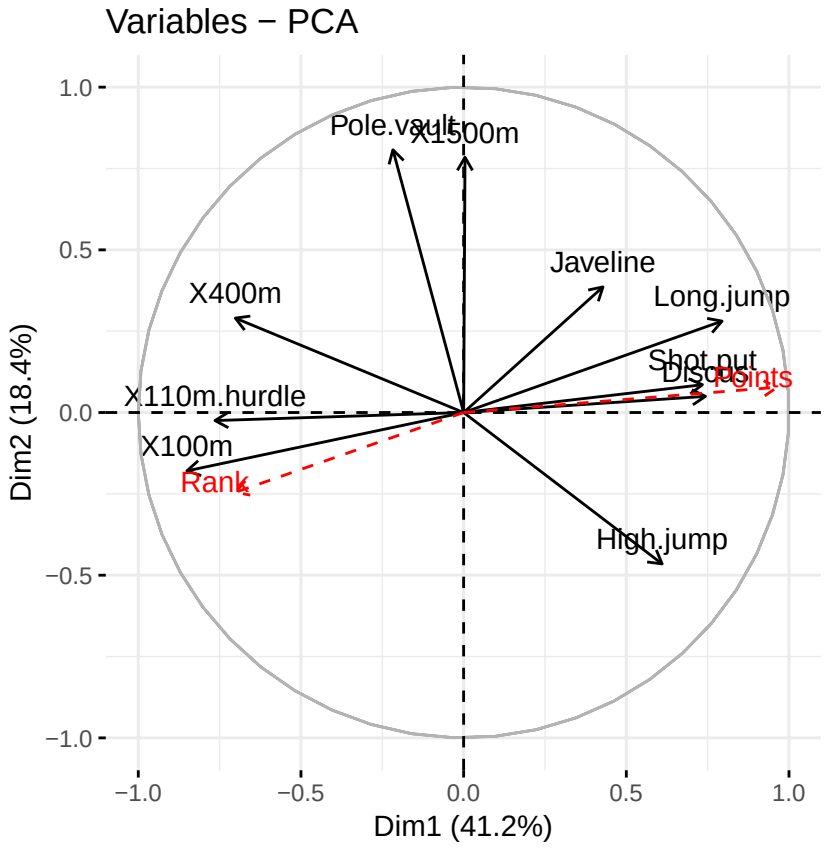



```
# Hide supplementary variables
fviz_pca_var(res.pca, invisible = "quanti.sup")
```



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")
```



4.23 Individuals

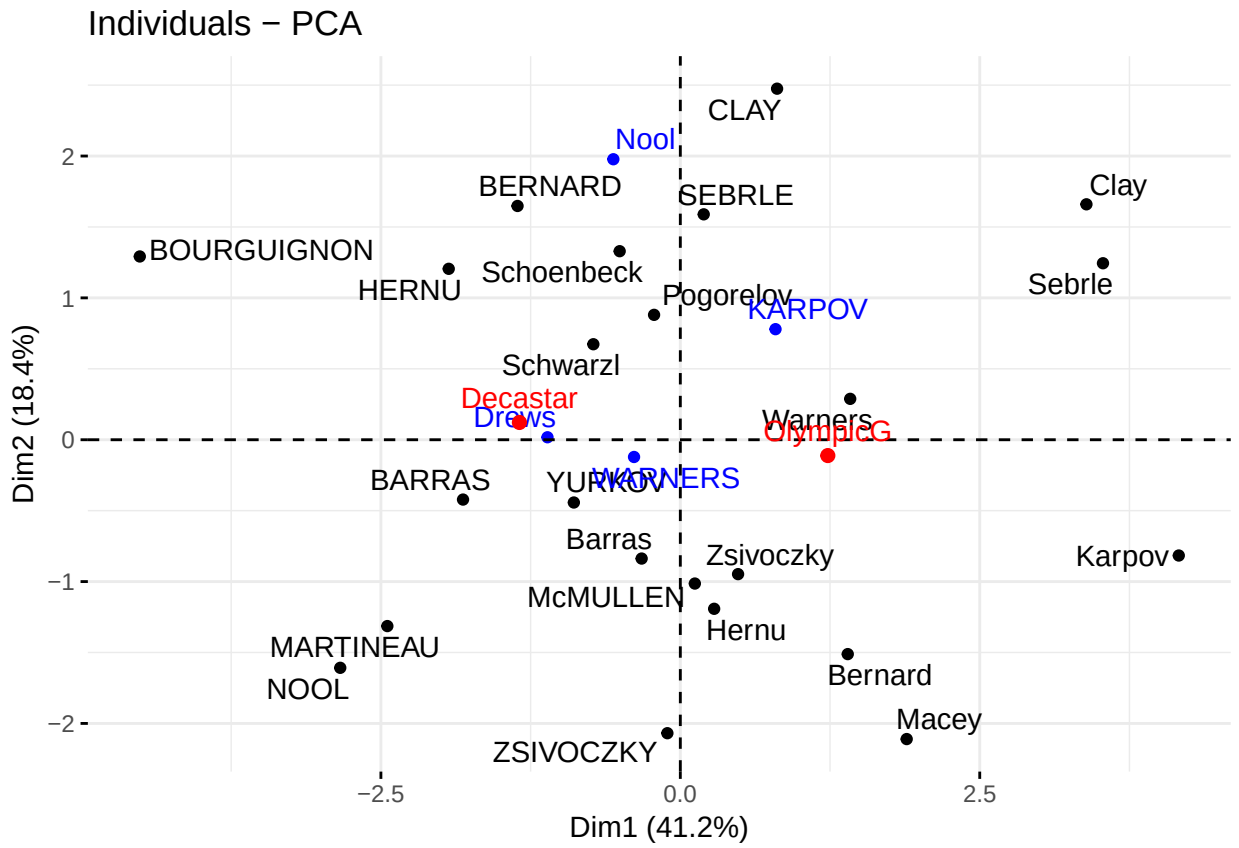
Predicted results for the supplementary individuals (ind.sup):

```
res.pca$ind.sup
```

```
:> $coord
:>              Dim.1       Dim.2      Dim.3      Dim.4       Dim.5
:> KARPOV   0.7947206  0.77951227 -1.6330203  1.7242283 -0.75070396
:> WARNERS -0.3864645 -0.12159237 -1.7387332 -0.7063341 -0.03230011
:> Nool    -0.5591306  1.97748871 -0.4830358 -2.2784526 -0.25461493
:> Drews   -1.1092038  0.01741477 -3.0488182 -1.5343468 -0.32642192
:>
:> $cos2
:>              Dim.1        Dim.2      Dim.3      Dim.4        Dim.5
:> KARPOV  0.05104677 4.911173e-02 0.21553730 0.24028620 0.0455487744
:> WARNERS 0.02422707 2.398250e-03 0.49039677 0.08092862 0.0001692349
:> Nool    0.02897149 3.623868e-01 0.02162236 0.48108780 0.0060077529
:> Drews   0.09207094 2.269527e-05 0.69560547 0.17617609 0.0079736753
:>
:> $dist
:>   KARPOV  WARNERS     Nool    Drews
:> 3.517470 2.482899 3.284943 3.655527
```

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.

4.24 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:

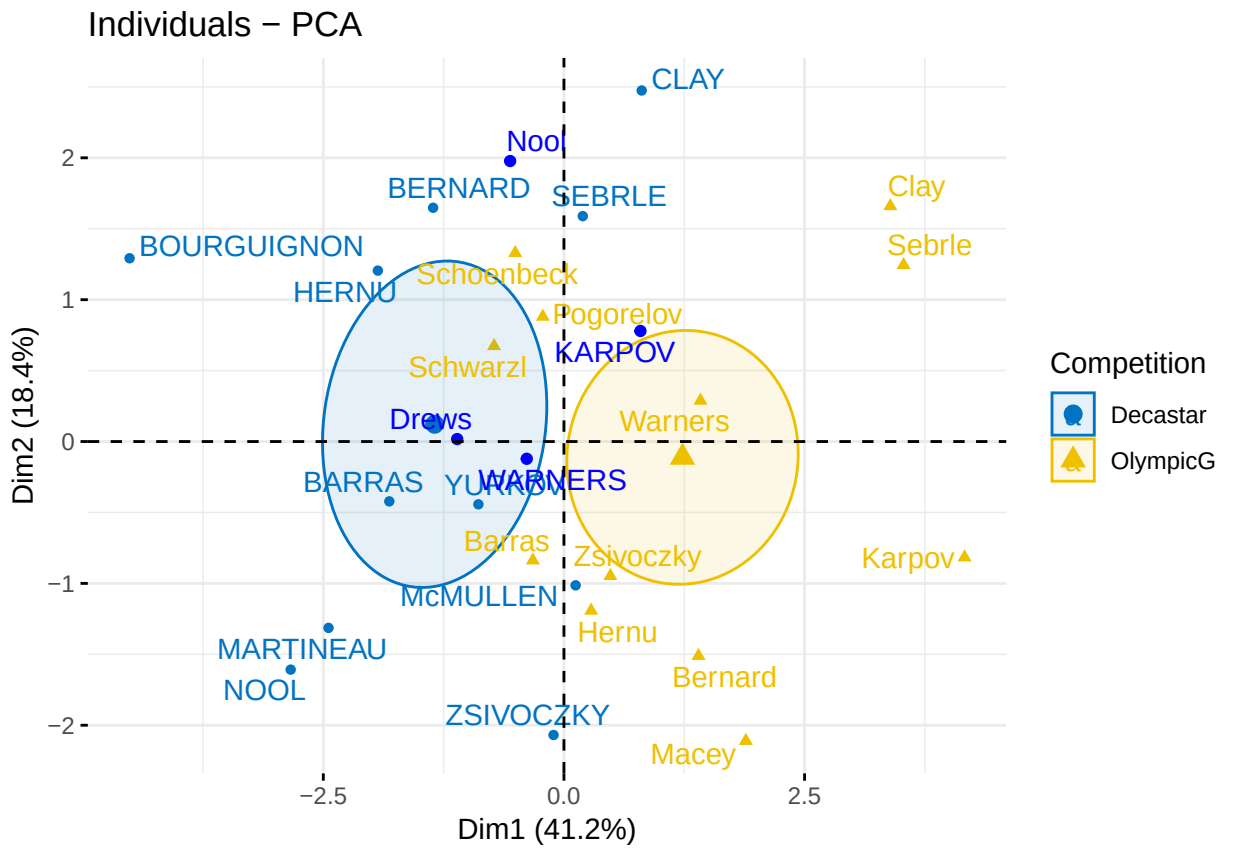
```
res.pca$quali
```

```
:> $coord
:>              Dim.1      Dim.2       Dim.3      Dim.4      Dim.5
:> Decastar -1.343451  0.1218097 -0.03789524  0.1808357  0.1343364
:> OlympicG  1.231497 -0.1116589  0.03473730 -0.1657661 -0.1231417
:>
:> $cos2
:>              Dim.1       Dim.2        Dim.3      Dim.4       Dim.5
```

```
:> Decastar 0.9051233 0.007440939 0.0007201669 0.01639956 0.009050062
:> OlympicG 0.9051233 0.007440939 0.0007201669 0.01639956 0.009050062
:>
:> $v.test
:>              Dim.1      Dim.2      Dim.3      Dim.4      Dim.5
:> Decastar -2.970766  0.4034256 -0.1528767  0.8971036  0.7202457
:> OlympicG  2.970766 -0.4034256  0.1528767 -0.8971036 -0.7202457
:>
:> $dist
:> Decastar OlympicG
:> 1.412108 1.294433
:>
:> $eta2
:>                 Dim.1      Dim.2       Dim.3      Dim.4      Dim.5
:> Competition 0.4011568 0.00739783 0.001062332 0.03658159 0.02357972
```

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)
```

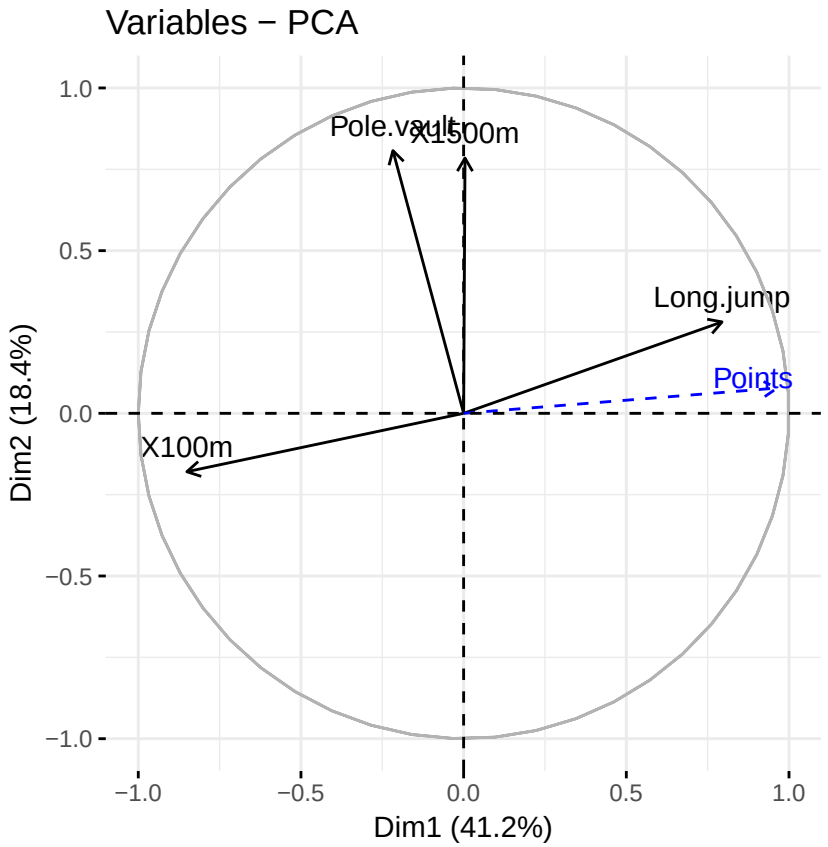


Recall that, to remove the mean points of groups, specify the argument mean.point = FALSE.

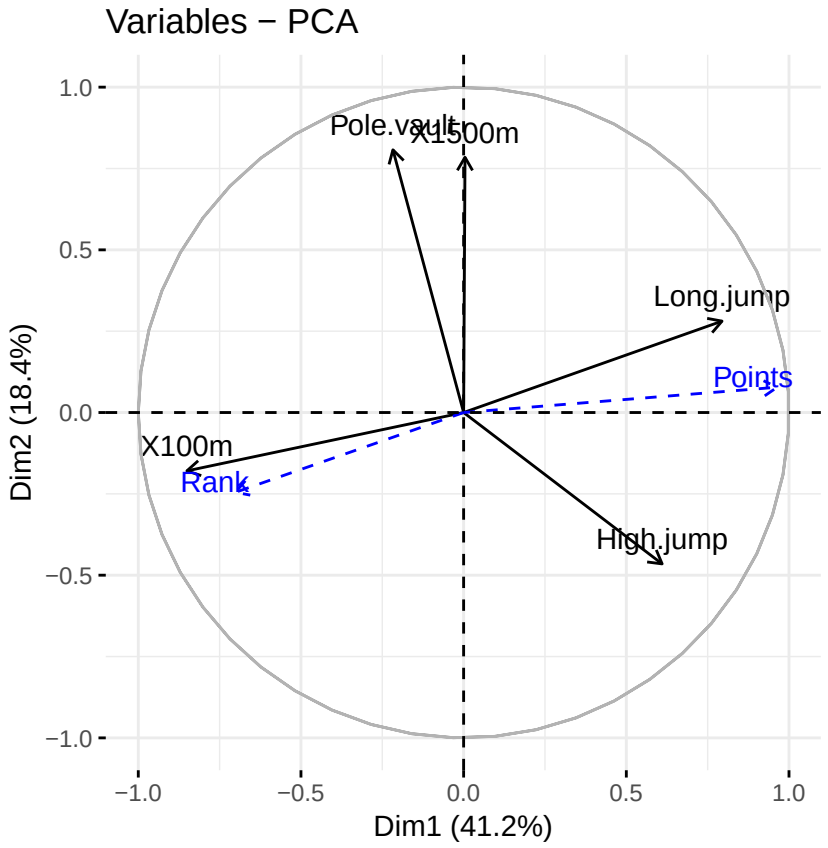
4.25 Filtering results

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

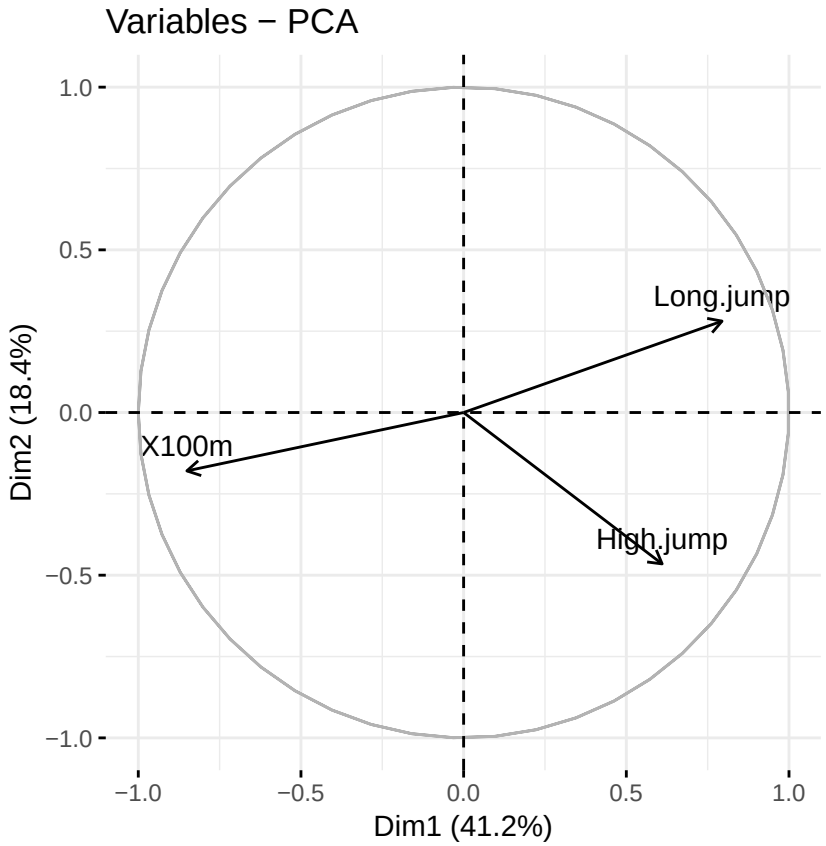
```
# Visualize variable with cos2 >= 0.6
fviz_pca_var(res.pca, select.var = list(cos2 = 0.6))
```



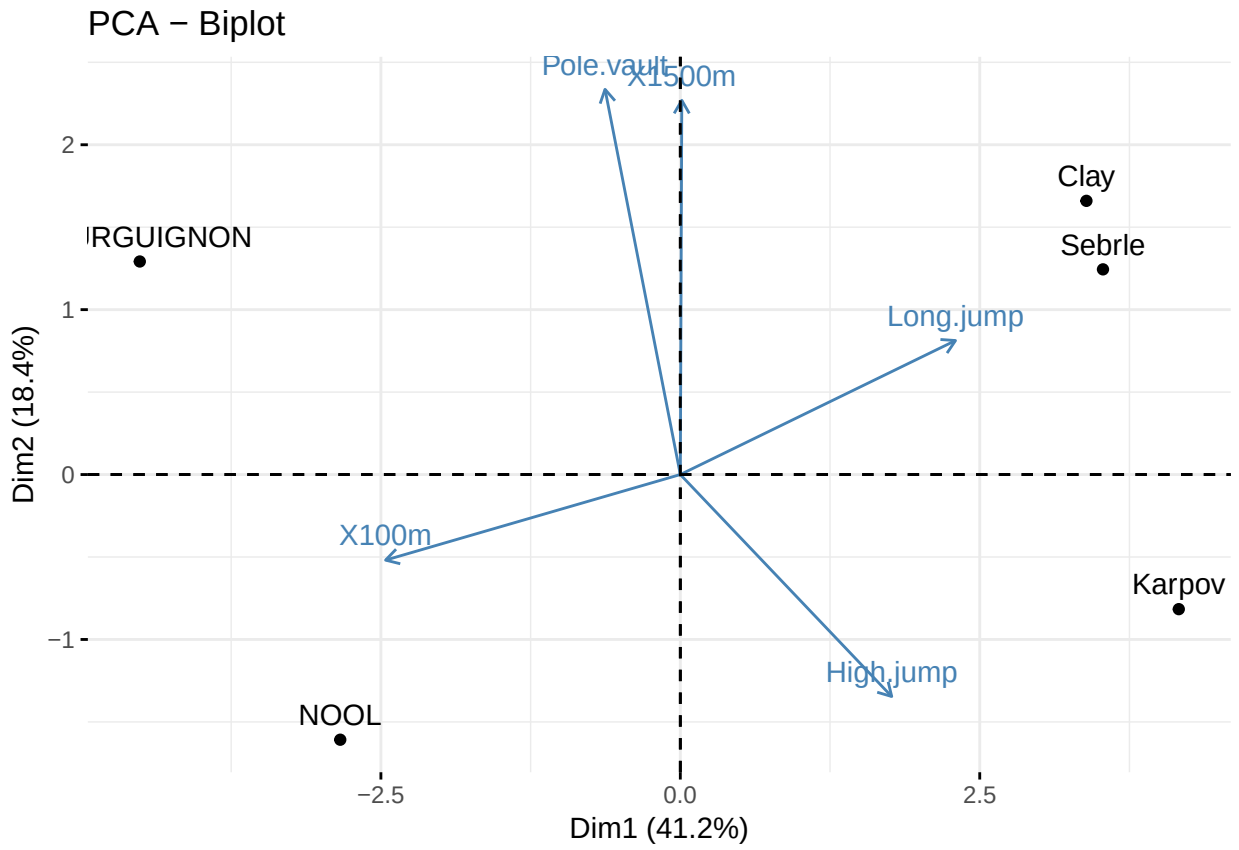

```
# Top 5 active variables with the highest cos2
fviz_pca_var(res.pca, select.var= list(cos2 = 5))
```




```
# 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())
```



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.

4.26 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)
ggexport(plotlist = list(scree.plot, ind.plot, var.plot),
         filename = file.path(data_out_dir, "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"))
```

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"
```

4.27 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 = ";")
```

4.28 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]

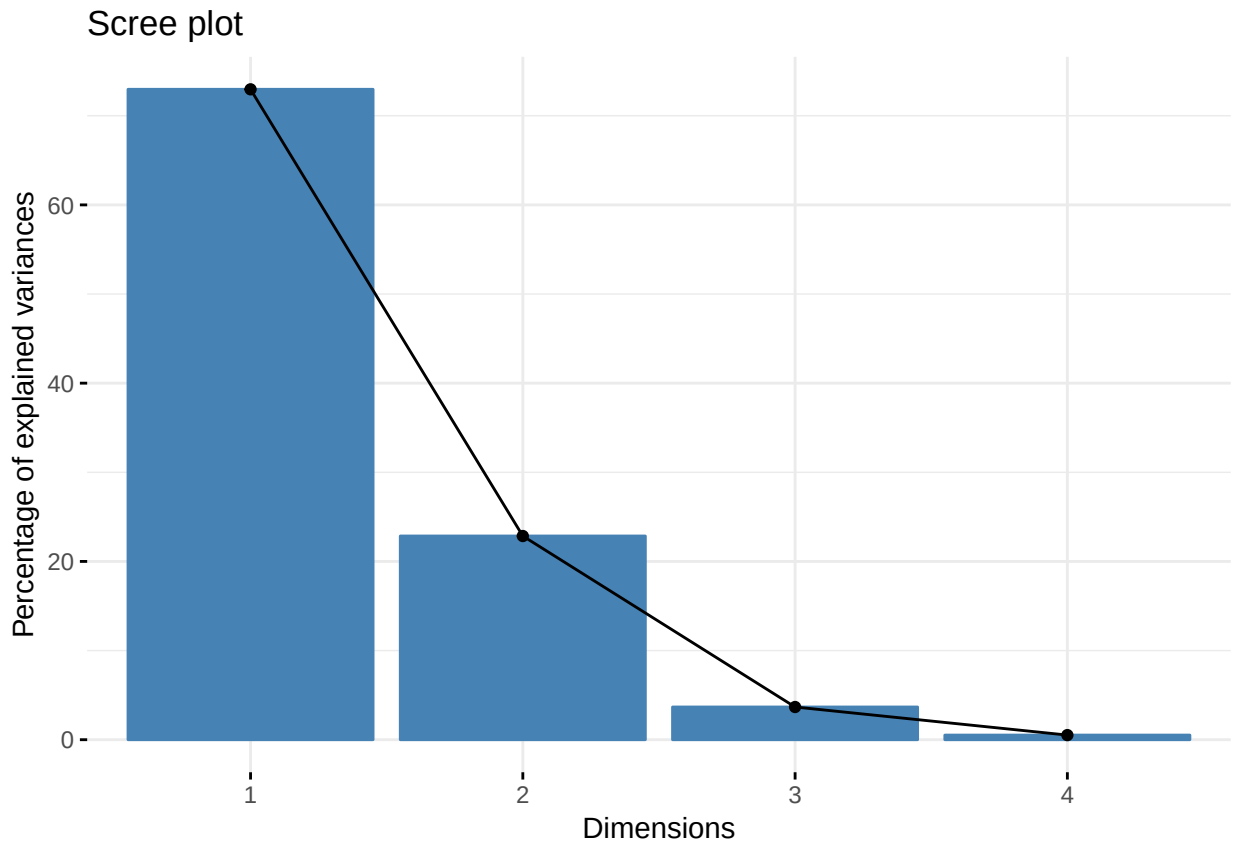
```
library(ade4)
res.pca <- dudi.pca(iris[, -5], scannf = FALSE, nf = 5)
```

4. Using epPCA() [ExPosition]

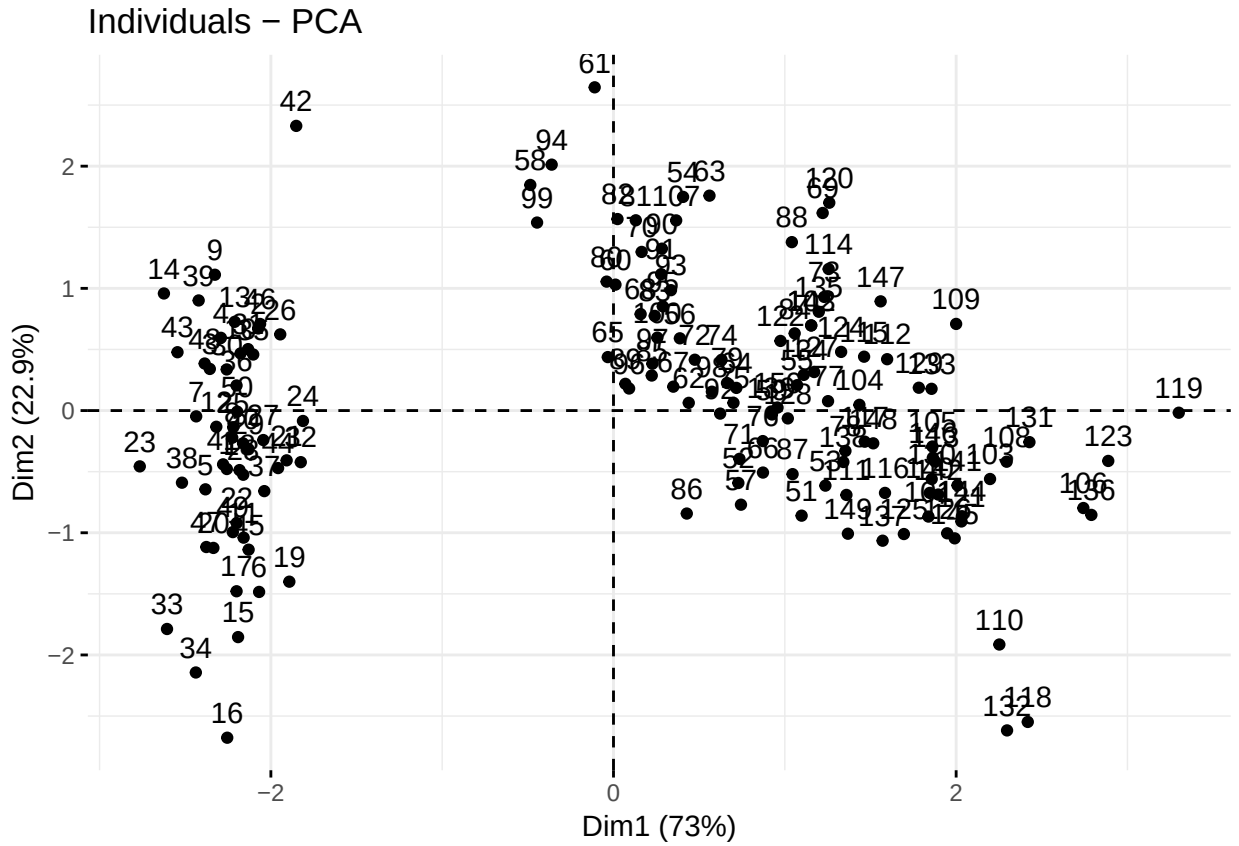
```
library(ExPosition)
res.pca <- epPCA(iris[, -5], graph = FALSE)
```

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:

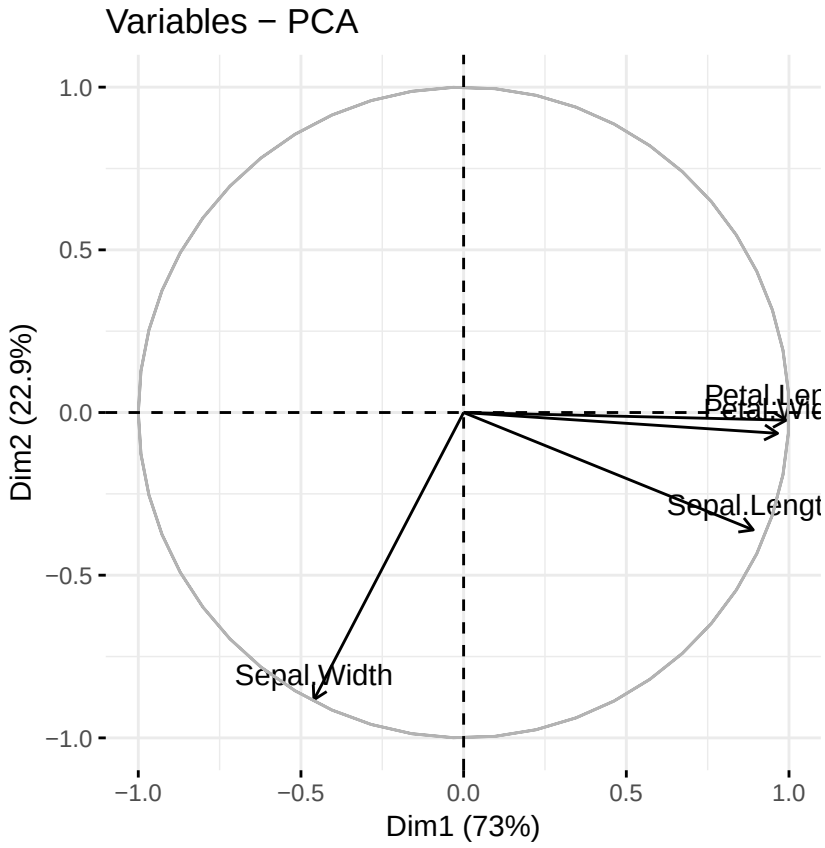
```
fviz_eig(res.pca)     # Scree plot
```




```
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.Ler
Petal.Vie
Sepal.Lengt
Sepal.Width
−1.0
−0.5
0.0
0.5
1.0
Dim1 (73%)

Chapter 5

Biplot of the Iris data set

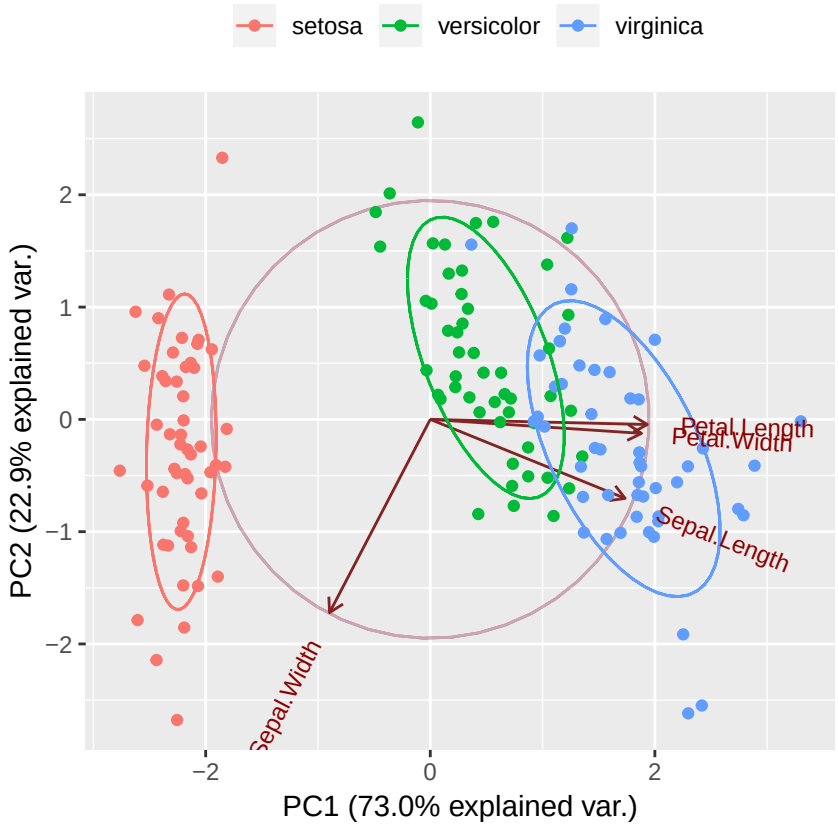
```
# devtools::install_github("vqv/ggbiplot")
library(ggbiplot)

iris.pca <- prcomp(iris[, 1:4], center = TRUE, scale = TRUE)
print(iris.pca)
```

```
:> Standard deviations (1, .., p=4):
:> [1] 1.7083611 0.9560494 0.3830886 0.1439265
:>
:> Rotation (n x k) = (4 x 4):
:>                     PC1         PC2        PC3        PC4
:> Sepal.Length  0.5210659 -0.37741762  0.7195664  0.2612863
:> Sepal.Width  -0.2693474 -0.92329566 -0.2443818 -0.1235096
:> Petal.Length  0.5804131 -0.02449161 -0.1421264 -0.8014492
:> Petal.Width   0.5648565 -0.06694199 -0.6342727  0.5235971
```

```
summary(iris.pca)
```

```
:> Importance of components:
:>                           PC1    PC2     PC3     PC4
:> Standard deviation     1.7084 0.9560 0.38309 0.14393
:> Proportion of Variance 0.7296 0.2285 0.03669 0.00518
:> Cumulative Proportion  0.7296 0.9581 0.99482 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.

Chapter 6

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] 1.7083611 0.9560494 0.3830886 0.1439265
:>
:> Rotation (n x k) = (4 x 4):
:>                     PC1         PC2        PC3        PC4
:> Sepal.Length  0.5210659 -0.37741762  0.7195664  0.2612863
:> Sepal.Width  -0.2693474 -0.92329566 -0.2443818 -0.1235096
:> Petal.Length  0.5804131 -0.02449161 -0.1421264 -0.8014492
:> Petal.Width   0.5648565 -0.06694199 -0.6342727  0.5235971
```

```
# 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,] -2.257141 -0.4784238  0.12727962  0.024087508
:> [2,] -2.074013  0.6718827  0.23382552  0.102662845
:> [3,] -2.356335  0.3407664 -0.04405390  0.028282305
:> [4,] -2.291707  0.5953999 -0.09098530 -0.065735340
:> [5,] -2.381863 -0.6446757 -0.01568565 -0.035802870
:> [6,] -2.068701 -1.4842053 -0.02687825  0.006586116
```

```
# 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] 2.91849782 0.91403047 0.14675688 0.02071484
```

```
# # 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 2.9185 0.00000 0.00000 0.00000
:> PC2 0.0000 0.91403 0.00000 0.00000
:> PC3 0.0000 0.00000 0.14676 0.00000
:> PC4 0.0000 0.00000 0.00000 0.02071
```

Chapter 7

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] 2.91849782 0.91403047 0.14675688 0.02071484
:>
:> $vectors
:>            [,1]        [,2]       [,3]       [,4]
:> [1,]  0.5210659 -0.37741762  0.7195664  0.2612863
:> [2,] -0.2693474 -0.92329566 -0.2443818 -0.1235096
:> [3,]  0.5804131 -0.02449161 -0.1421264 -0.8014492
:> [4,]  0.5648565 -0.06694199 -0.6342727  0.5235971
```

```
# 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,] -2.257141 -0.4784238  0.12727962  0.024087508
:> [2,] -2.074013  0.6718827  0.23382552  0.102662845
:> [3,] -2.356335  0.3407664 -0.04405390  0.028282305
:> [4,] -2.291707  0.5953999 -0.09098530 -0.065735340
:> [5,] -2.381863 -0.6446757 -0.01568565 -0.035802870
:> [6,] -2.068701 -1.4842053 -0.02687825  0.006586116
```

```
# Compute covariance matrix for new dataset
round(cov(df.new.t), 5)
```

```
:>        PC1     PC2     PC3     PC4
:> PC1 2.9185 0.00000 0.00000 0.00000
:> PC2 0.0000 0.91403 0.00000 0.00000
:> PC3 0.0000 0.00000 0.14676 0.00000
:> PC4 0.0000 0.00000 0.00000 0.02071
```

Chapter 8

Diagnostic Plots

I have tried to use fortify function in ggplot2 which can access different statistics related to linear model. The basic diagnostic plot which we often get using plot function in the fitted model using lm command.

The function `diagPlots` gives an list of six different plots which can be arranged in a grid using `grid` and `gridExtra` packages.

```
library(ggplot2)

diagPlot<-function(model){
    p1<-ggplot(model, aes(.fitted, .resid))+geom_point()
    p1<-p1+stat_smooth(method="loess")+geom_hline(yintercept=0, col="red", linetype="dashed")
    p1<-p1+xlab("Fitted values")+ylab("Residuals")
    p1<-p1+ggtitle("Residual vs Fitted Plot")+theme_bw()

    p2<-ggplot(model, aes(qqnorm(.stdresid)[[1]], .stdresid))+geom_point(na.rm = TRUE)
    p2<-p2+geom_abline(aes(qqline(.stdresid)))+xlab("Theoretical Quantiles")+ylab("Standardized Residual
    p2<-p2+ggtitle("Normal Q-Q")+theme_bw()

    p3<-ggplot(model, aes(.fitted, sqrt(abs(.stdresid))))+geom_point(na.rm=TRUE)
    p3<-p3+stat_smooth(method="loess", na.rm = TRUE)+xlab("Fitted Value")
    p3<-p3+ylab(expression(sqrt("|Standardized residuals|")))
    p3<-p3+ggtitle("Scale-Location")+theme_bw()

    p4<-ggplot(model, aes(seq_along(.cooksd), .cooksd))+geom_bar(stat="identity", position="identity")
    p4<-p4+xlab("Obs. Number")+ylab("Cook's distance")
    p4<-p4+ggtitle("Cook's distance")+theme_bw()

    p5<-ggplot(model, aes(.hat, .stdresid))+geom_point(aes(size=.cooksd), na.rm=TRUE)
    p5<-p5+stat_smooth(method="loess", na.rm=TRUE)
    p5<-p5+xlab("Leverage")+ylab("Standardized Residuals")
    p5<-p5+ggtitle("Residual vs Leverage Plot")
    p5<-p5+scale_size_continuous("Cook's Distance", range=c(1,5))
    p5<-p5+theme_bw()+theme(legend.position="bottom")

    p6<-ggplot(model, aes(.hat, .cooksd))+geom_point(na.rm=TRUE)+stat_smooth(method="loess", na.rm=TRUE)
    p6<-p6+xlab("Leverage hii")+ylab("Cook's Distance")
    p6<-p6+ggtitle("Cook's dist vs Leverage hii/(1-hii)")
    p6<-p6+geom_abline(slope=seq(0,3,0.5), color="gray", linetype="dashed")
```

```
    p6<-p6+theme_bw()

    return(list(rvfPlot=p1, qqPlot=p2, sclLocPlot=p3, cdPlot=p4, rvlevPlot=p5, cvlPlot=p6))
}
```

Using the mtcars datasets, a linear model is fitted with mpg as response and cyl, disp, hp, drat and wt has predictor variable

```
lm.model <- lm(mpg ~ cyl+disp+hp+drat+wt, data=mtcars)
diagPlts <- diagPlot(lm.model)
```

To display the plots in a grid, some packages mentioned above should be installed.

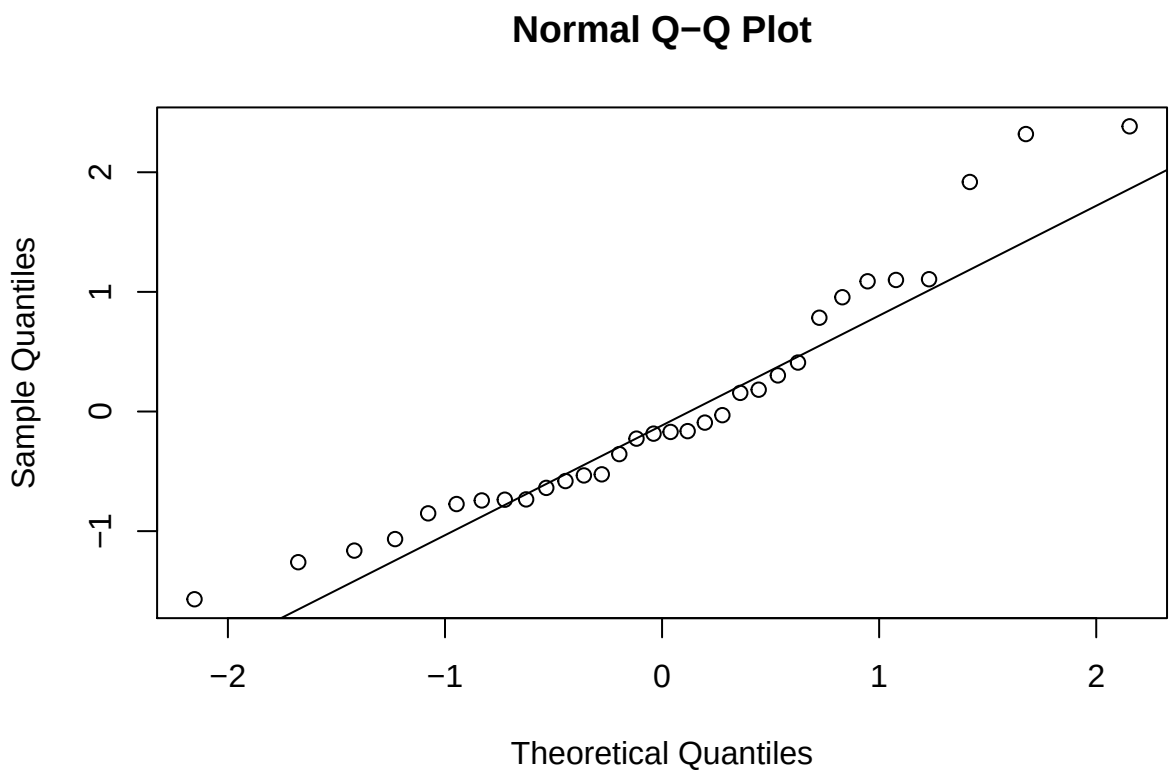
```
lbry <- c("grid", "gridExtra")
lapply(lbry, require, character.only=TRUE, warn.conflicts = FALSE, quietly = TRUE)
```

```
:> [[1]]
:> [1] TRUE
:>
:> [[2]]
:> [1] TRUE
```

Thus the plot obtained is,

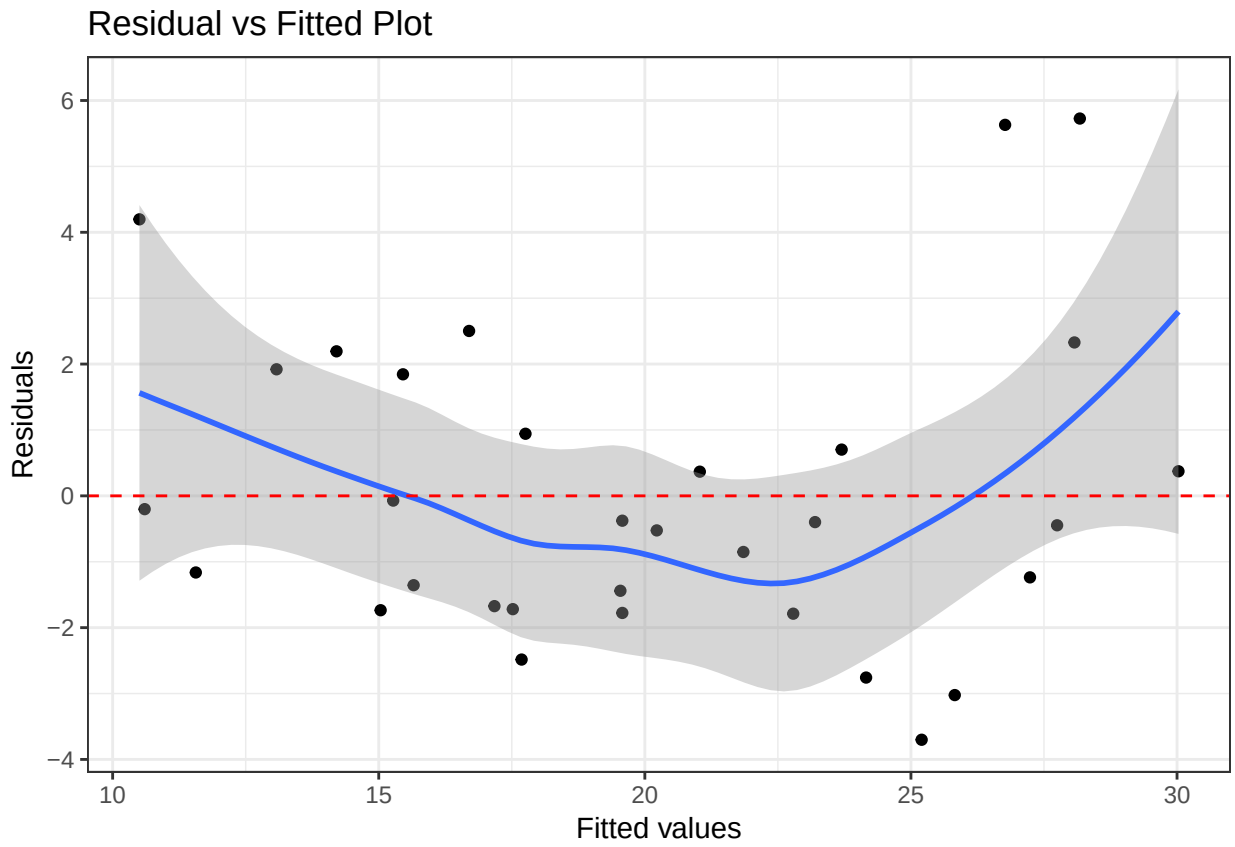
```
do.call(grid.arrange, c(diagPlts, main="Diagnostic Plots", ncol=3))
```

```
:> Error: Aesthetics must be either length 1 or the same as the data (32): x
```



8.1 Residual vs Fitted plot

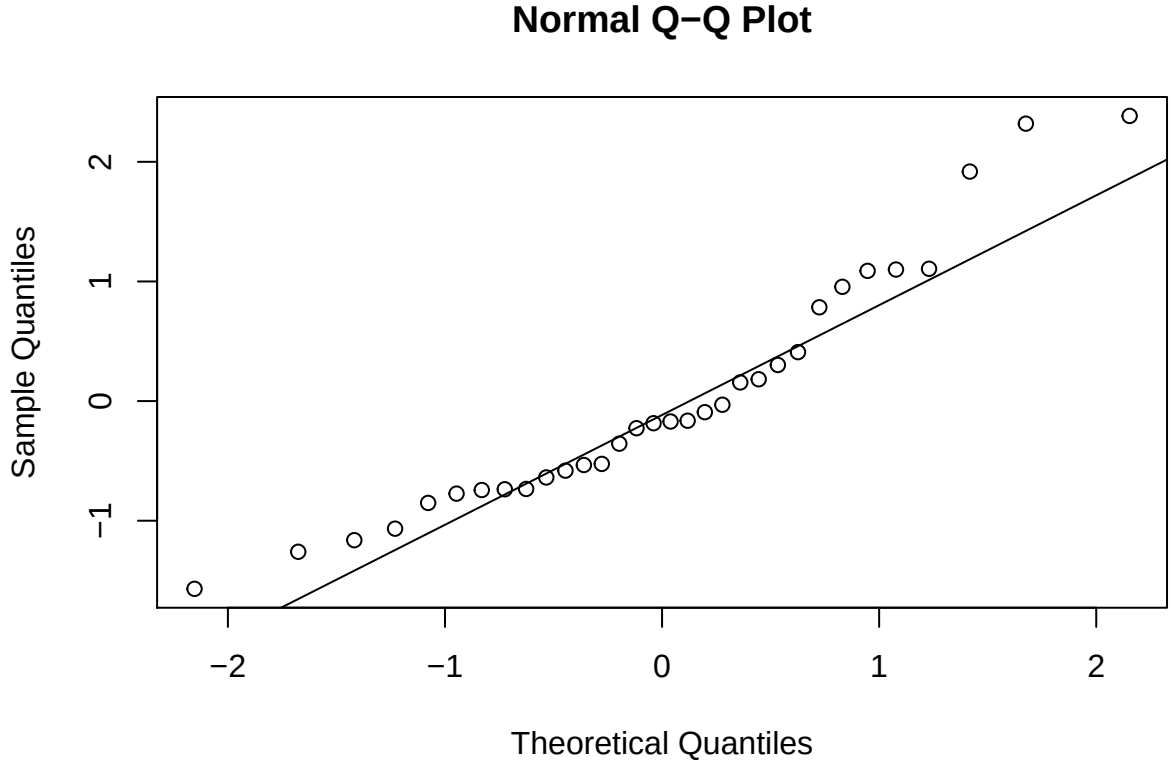
```
model <- lm.model
    p1<-ggplot(model, aes(.fitted, .resid)) +
        geom_point() +
        stat_smooth(method="loess") +
        geom_hline(yintercept=0, col="red", linetype="dashed") +
        xlab("Fitted values") +
        ylab("Residuals") +
        ggtitle("Residual vs Fitted Plot") +
        theme_bw()
    p1
```



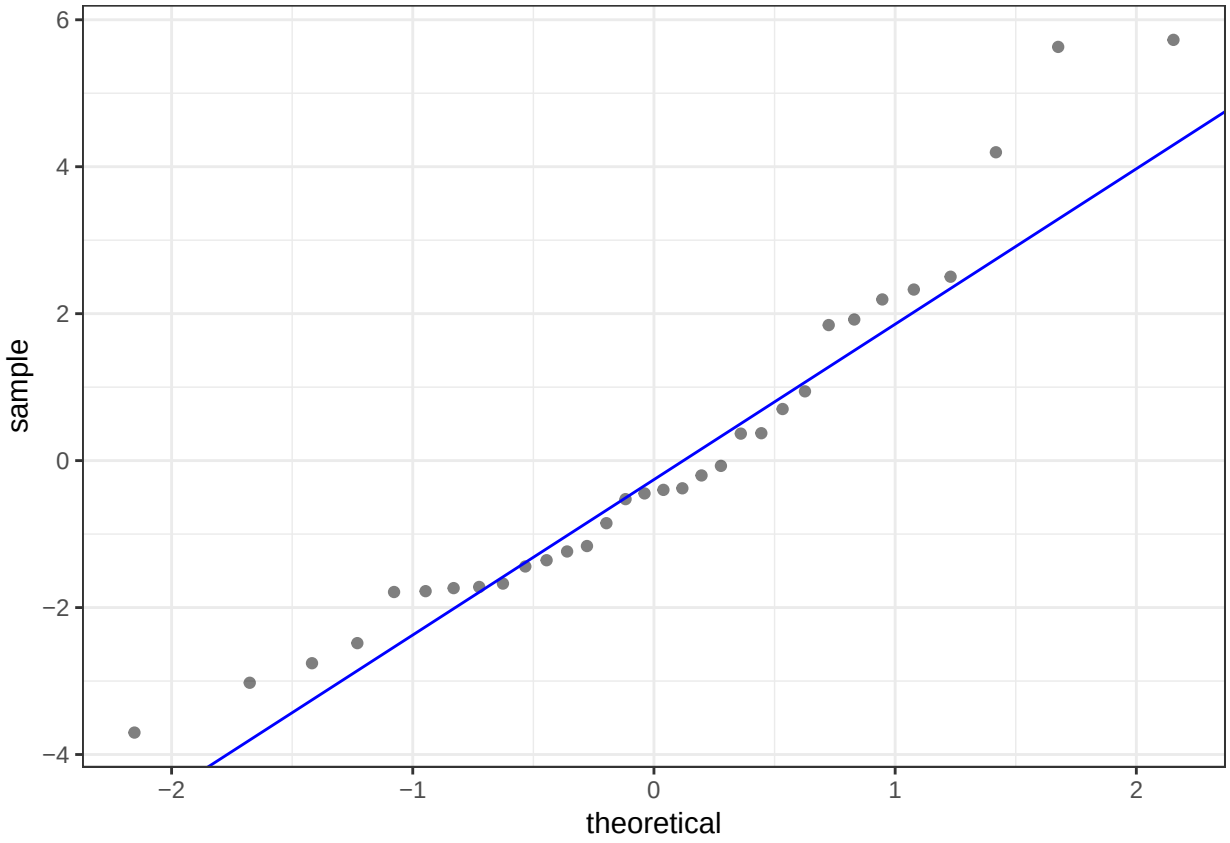
8.2 Nomral QQ

```
    p2 <- ggplot(model, aes(qqnorm(.stdresid)[[1]], .stdresid)) +
    geom_point(na.rm = TRUE) +
    geom_abline(aes(qqline(.stdresid)))
    # xlab("Theoretical Quantiles") +
    # ylab("Standardized Residuals") +
    # ggtitle("Normal Q-Q") +
    # theme_bw()

p2
```

```
:> Error: Aesthetics must be either length 1 or the same as the data (32): x
```



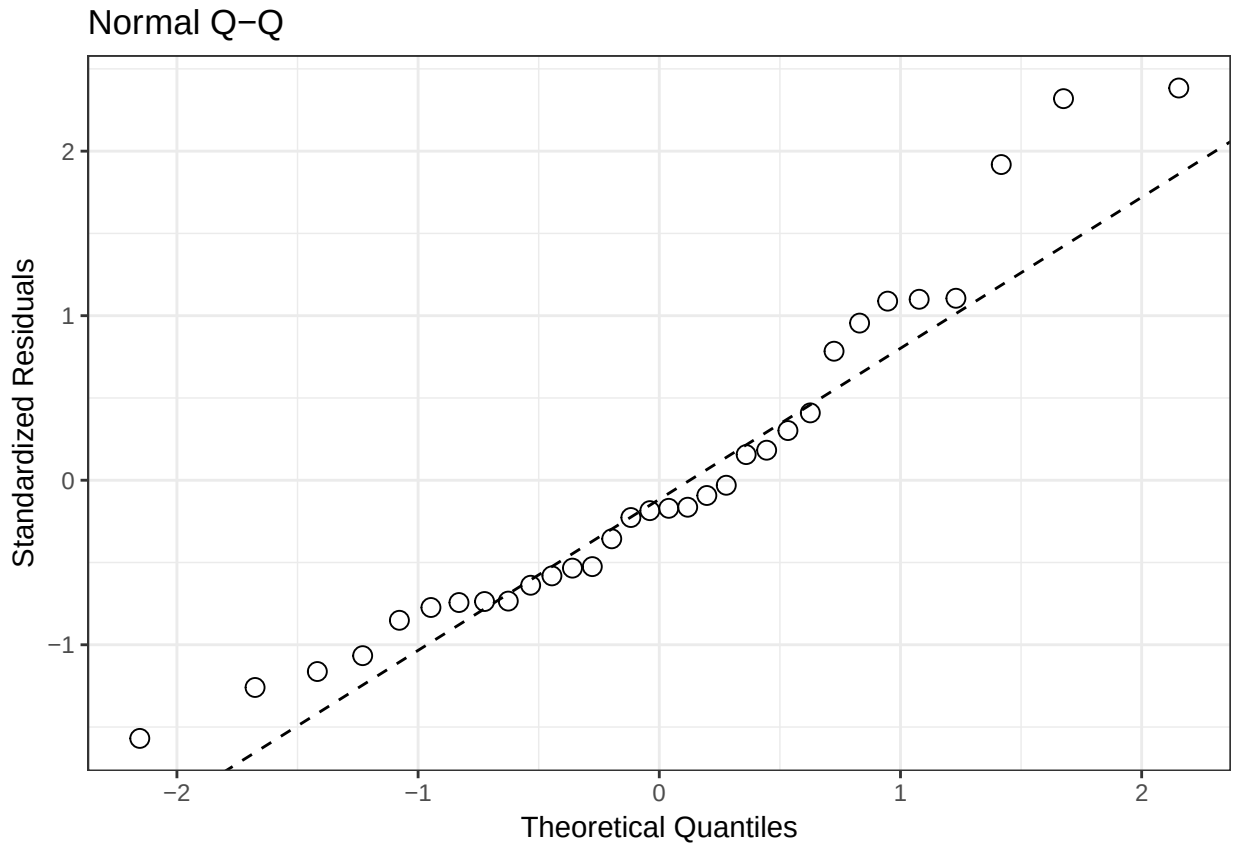
```
    y <- quantile(model$resid[!is.na(model$resid)], c(0.25, 0.75))
    x <- qnorm(c(0.25, 0.75))
    slope <- diff(y)/diff(x)
    int <- y[1L] - slope * x[1L]
    p <- ggplot(model, aes(sample = .resid)) +
        stat_qq(alpha = 0.5) +
        geom_abline(slope = slope, intercept = int, color="blue") +
        theme_bw()

    p
```



The standard Q-Q diagnostic for linear models plots quantiles of the standardized residuals vs. theoretical quantiles of N(0,1). ?'s ggQQ function plots the residuals. The snippet below amends that and adds a few cosmetic changes to make the plot more like what one would get from plot(lm(…)).

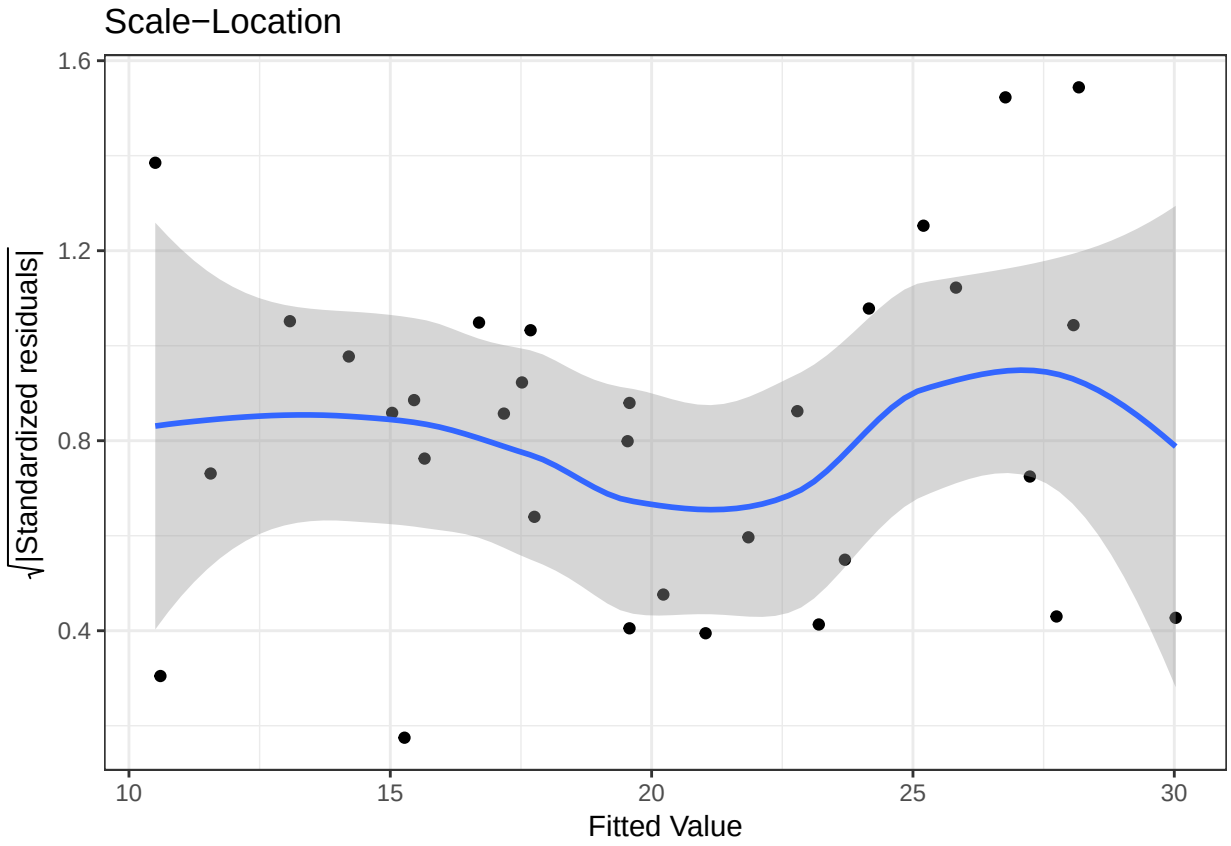
```
# https://stackoverflow.com/a/19990107/5270873
ggQQ = function(lm) {
  # extract standardized residuals from the fit
  d <- data.frame(std.resid = rstandard(lm))
  # calculate 1Q/4Q line
  y <- quantile(d$std.resid[!is.na(d$std.resid)], c(0.25, 0.75))
  x <- qnorm(c(0.25, 0.75))
  slope <- diff(y)/diff(x)
  int <- y[1L] - slope * x[1L]

  p <- ggplot(data=d, aes(sample=std.resid)) +
    stat_qq(shape=1, size=3) +           # open circles
```

```
    labs(title="Normal Q-Q",             # plot title
         x="Theoretical Quantiles",      # x-axis label
         y="Standardized Residuals") +   # y-axis label
    geom_abline(slope = slope, intercept = int, linetype="dashed")  + # dashed reference line
      theme_bw()
  return(p)
}

ggQQ(model)
```



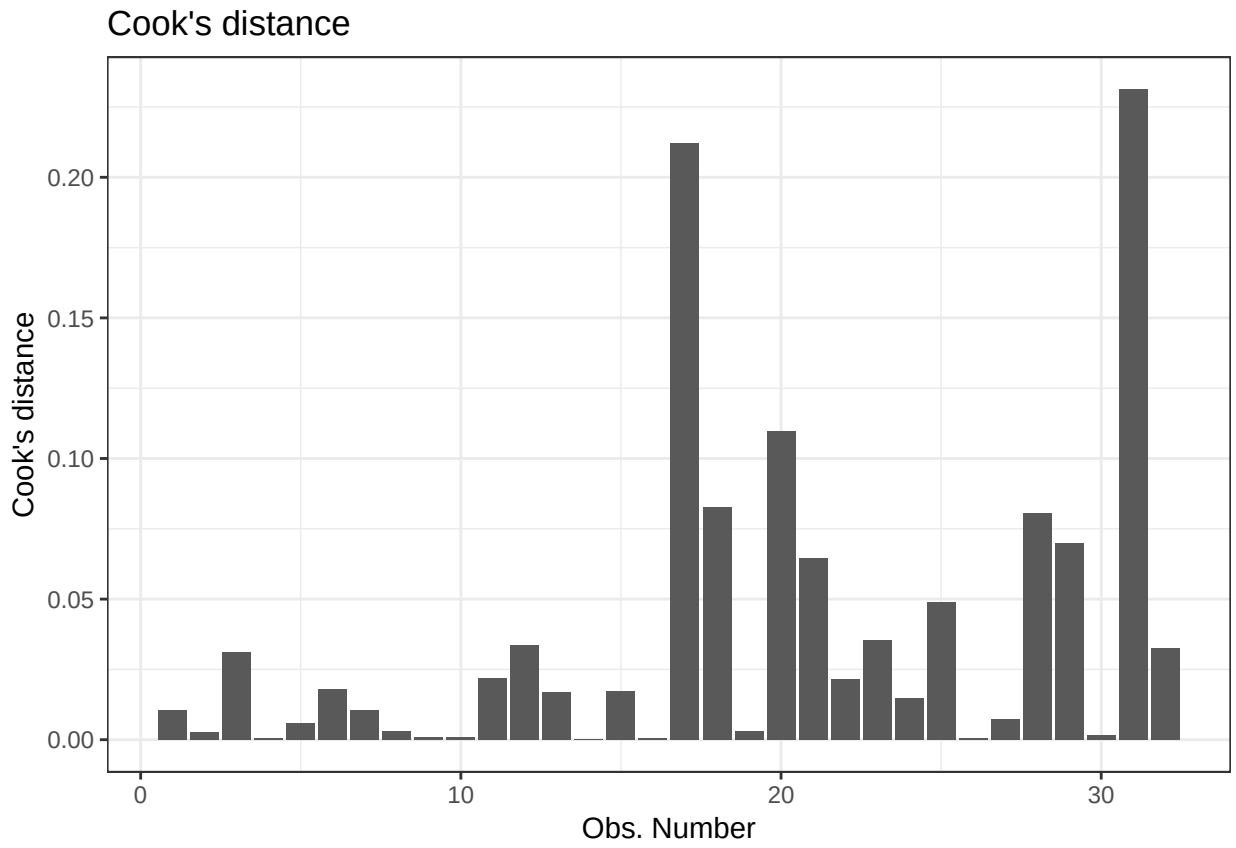
8.3 Scale-location

```
    p3 <- ggplot(model, aes(.fitted, sqrt(abs(.stdresid)))) +
    geom_point(na.rm=TRUE) +
    stat_smooth(method="loess", na.rm = TRUE) +
    xlab("Fitted Value") +
    ylab(expression(sqrt("|Standardized residuals|"))) +
    ggtitle("Scale-Location") +
    theme_bw()

    p3
```



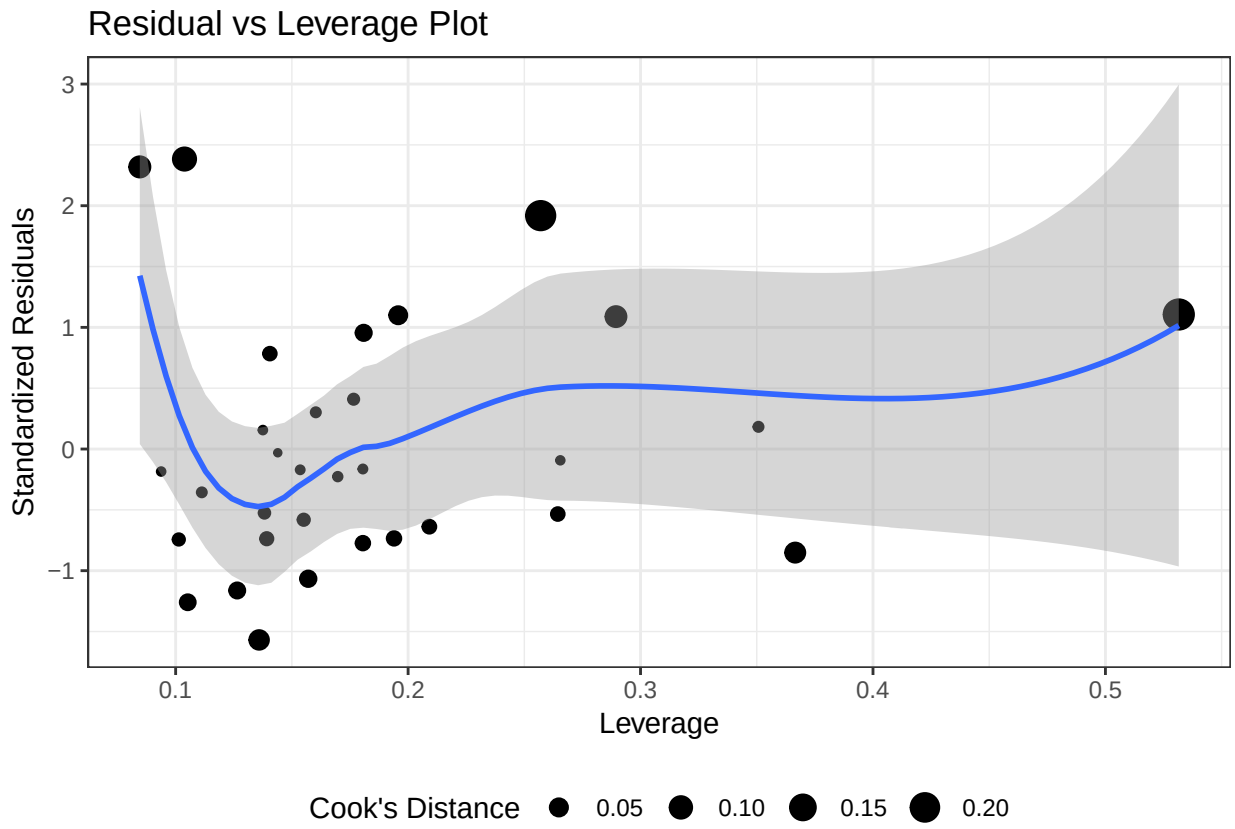
8.4 Cook's Distance

```
p4 <- ggplot(model, aes(seq_along(.cooksd), .cooksd)) +
geom_bar(stat="identity", position="identity") +
xlab("Obs. Number") +
ylab("Cook's distance") +
ggtitle("Cook's distance") +
theme_bw()

p4
```



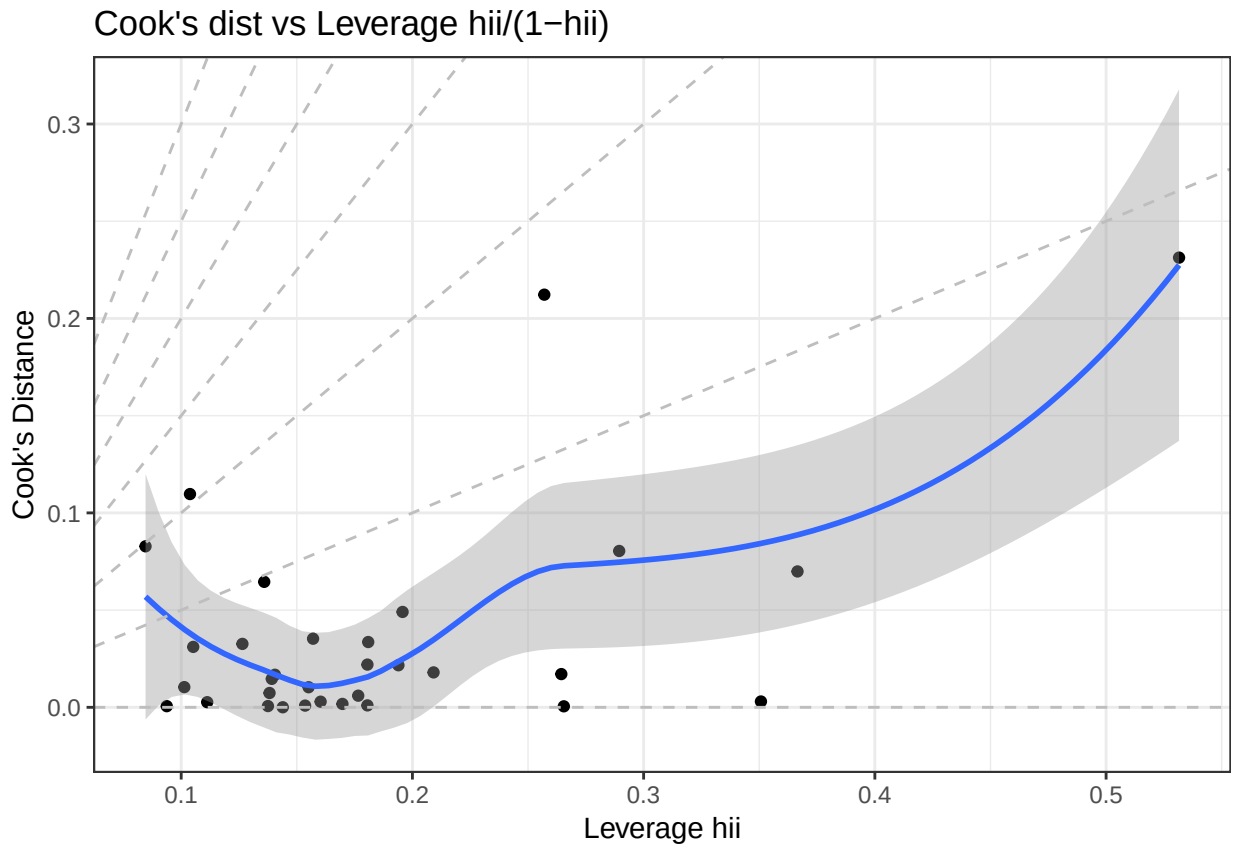
8.5 Residual vs Leverage Plot

```
p5 <- ggplot(model, aes(.hat, .stdresid)) +
geom_point(aes(size=.cooksd), na.rm=TRUE) +
stat_smooth(method="loess", na.rm=TRUE) +
xlab("Leverage") +
ylab("Standardized Residuals") +
ggtitle("Residual vs Leverage Plot") +
scale_size_continuous("Cook's Distance", range=c(1,5)) +
theme_bw() +
theme(legend.position="bottom")

p5
```



8.6 Cook's dist vs Leverage hii/(1-hii)

```
    p6 <- ggplot(model, aes(.hat, .cooksd)) +
    geom_point(na.rm=TRUE) +
    stat_smooth(method="loess", na.rm=TRUE) +
    xlab("Leverage hii") +
    ylab("Cook's Distance") +
    ggtitle("Cook's dist vs Leverage hii/(1-hii)") +
    geom_abline(slope=seq(0,3,0.5), color="gray", linetype="dashed") +
    theme_bw()

p6
```


Cook's dist vs Leverage hii/(1−hii)
Cook's Distance
0.0
0.1
0.2
0.3
0.1
0.2
0.3
0.4
0.5
Leverage hii

Chapter 9

Chapter 10

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] 0.04326896 0.03519677 0.05837573 0.03125017 0.03292182 0.03323551
:>  [7] 0.03544265 0.03127502 0.03140238 0.03292182 0.03292182 0.05575179
:> [13] 0.04010861 0.04192052 0.17047665 0.19533191 0.18379417 0.06611662
:> [19] 0.11774978 0.09562654 0.05031684 0.03433832 0.03284761 0.04431718
:> [25] 0.04452785 0.08664873 0.07035096 0.12911356 0.03132522 0.03798993
:> [31] 0.03544265 0.03769190
```

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)
```

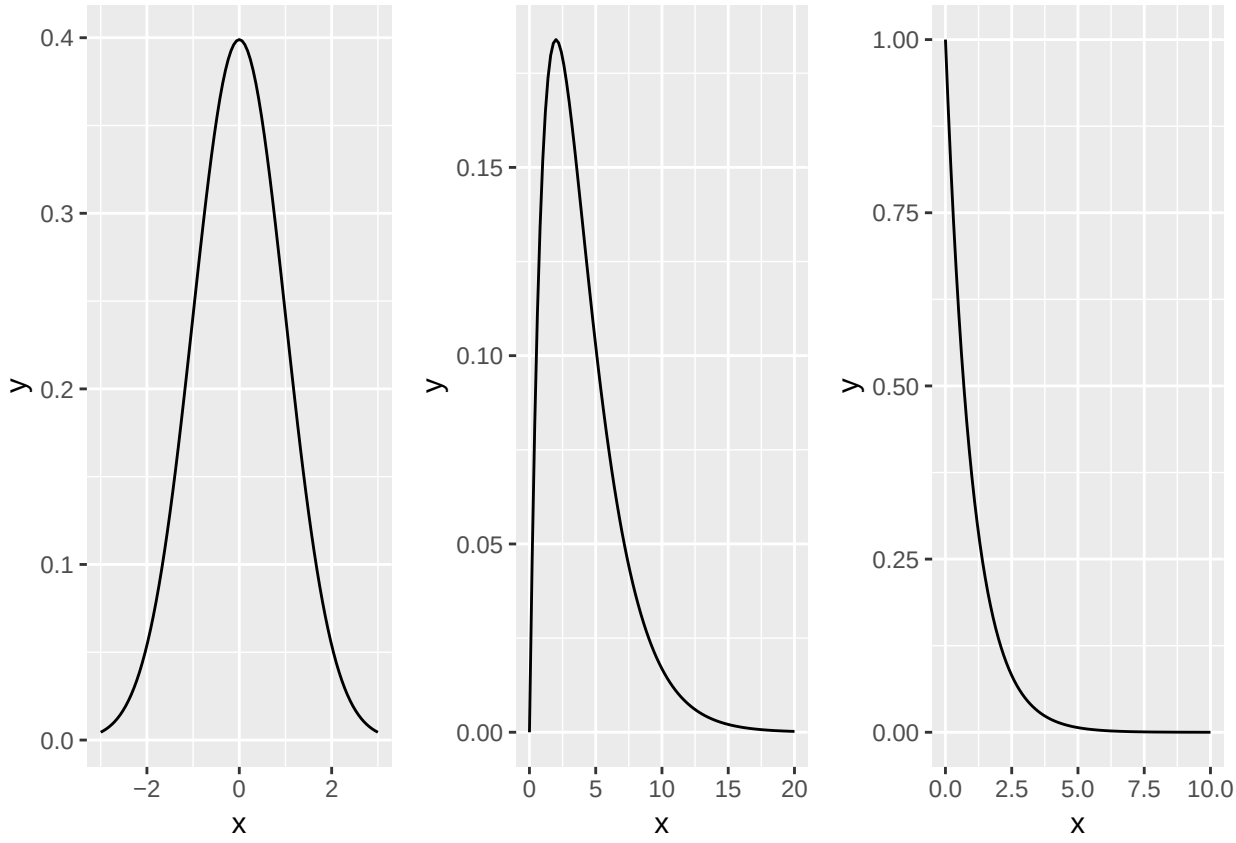
```
:>  [1] 0.04326896 0.03519677 0.05837573 0.03125017 0.03292182 0.03323551
:>  [7] 0.03544265 0.03127502 0.03140238 0.03292182 0.03292182 0.05575179
:> [13] 0.04010861 0.04192052 0.17047665 0.19533191 0.18379417 0.06611662
:> [19] 0.11774978 0.09562654 0.05031684 0.03433832 0.03284761 0.04431718
:> [25] 0.04452785 0.08664873 0.07035096 0.12911356 0.03132522 0.03798993
:> [31] 0.03544265 0.03769190
```

Chapter 11

Q-Q normal to compare data to distributions

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.



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.

Chapter 12

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.

12.1 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)

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

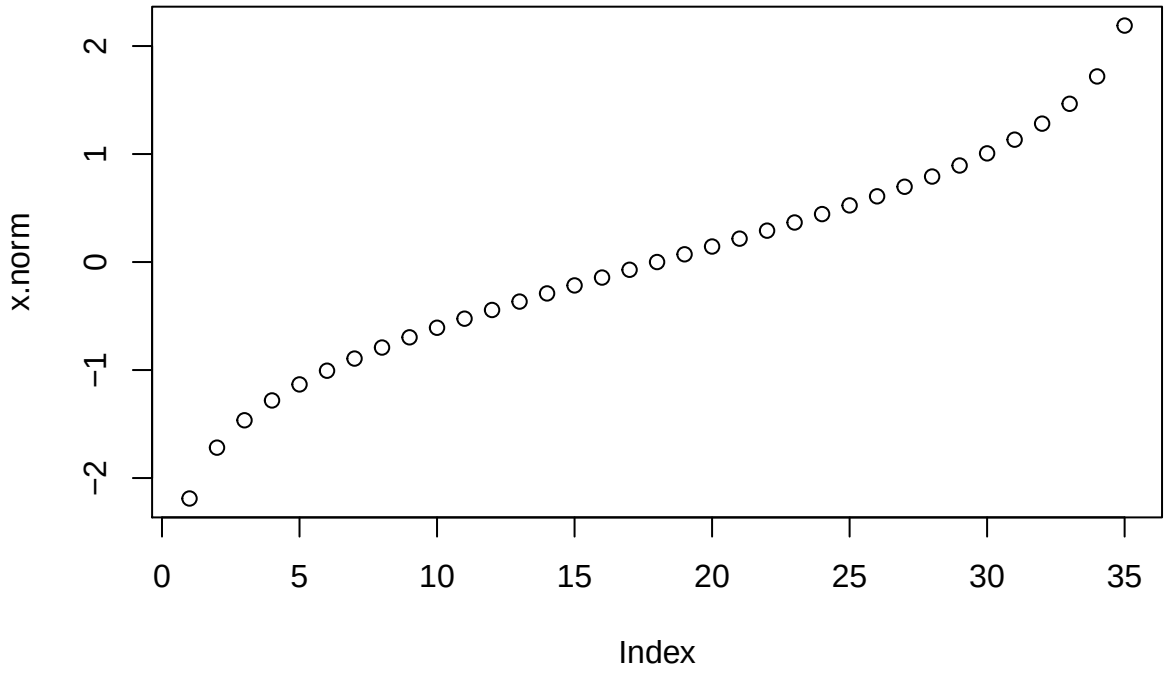
```
i <- 1:length(alto)
fi <- (i - 0.5) / length(alto)
fi
```

```
:>  [1] 0.01428571 0.04285714 0.07142857 0.10000000 0.12857143 0.15714286
:>  [7] 0.18571429 0.21428571 0.24285714 0.27142857 0.30000000 0.32857143
:> [13] 0.35714286 0.38571429 0.41428571 0.44285714 0.47142857 0.50000000
:> [19] 0.52857143 0.55714286 0.58571429 0.61428571 0.64285714 0.67142857
:> [25] 0.70000000 0.72857143 0.75714286 0.78571429 0.81428571 0.84285714
:> [31] 0.87142857 0.90000000 0.92857143 0.95714286 0.98571429
```

```
x.norm <- qnorm(fi)
x.norm
```

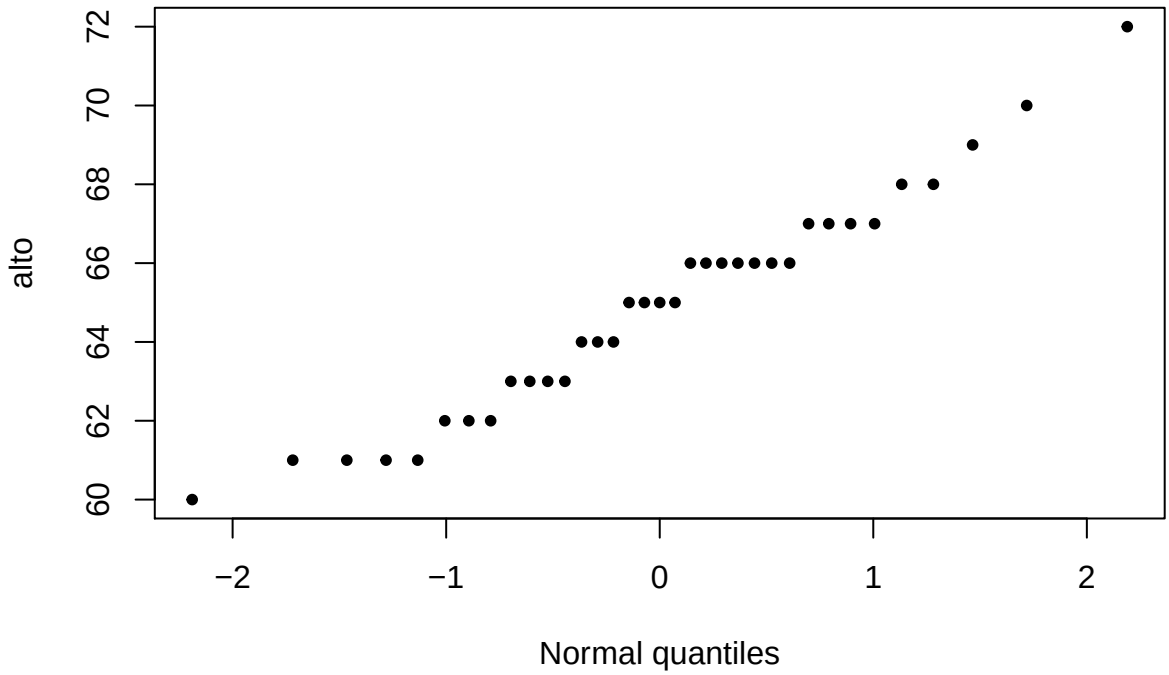
```
:>  [1] -2.18934976 -1.71845154 -1.46523379 -1.28155157 -1.13317003
:>  [6] -1.00626999 -0.89380063 -0.79163861 -0.69714143 -0.60849813
:> [11] -0.52440051 -0.44386131 -0.36610636 -0.29050677 -0.21653412
:> [16] -0.14372923 -0.07167928  0.00000000  0.07167928  0.14372923
:> [21]  0.21653412  0.29050677  0.36610636  0.44386131  0.52440051
:> [26]  0.60849813  0.69714143  0.79163861  0.89380063  1.00626999
:> [31]  1.13317003  1.28155157  1.46523379  1.71845154  2.18934976
```

```
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%
:> 63.00 66.75
```

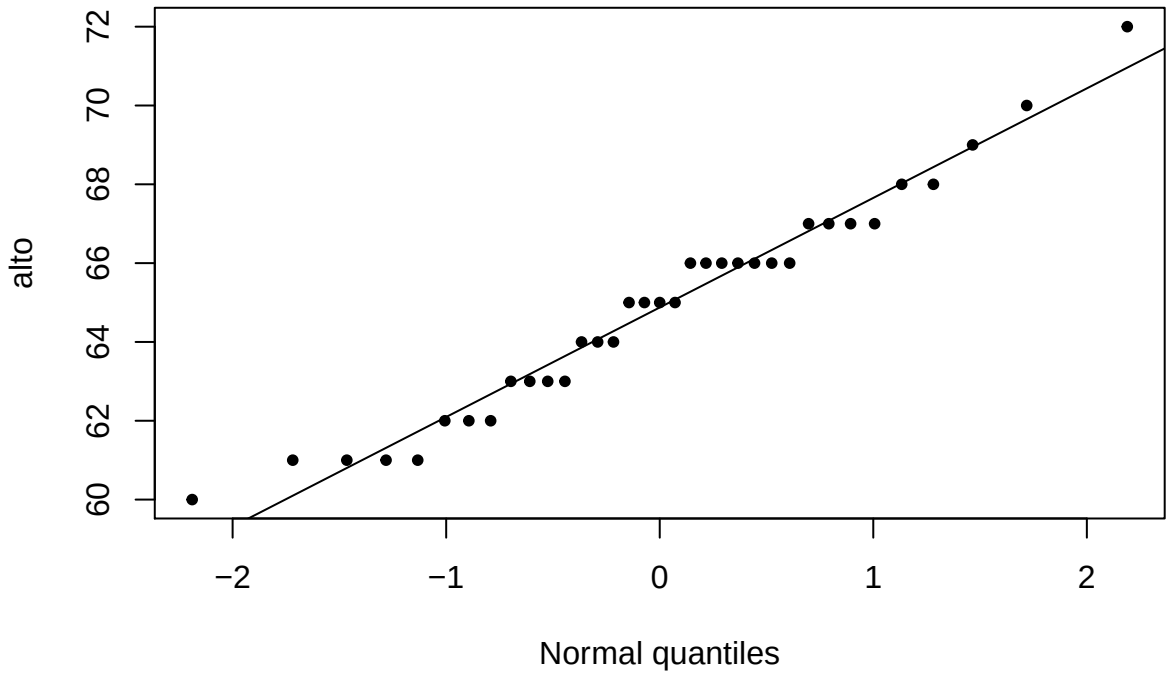
```
# Find the 1st and 3rd quartile of the normal distribution
x <- qnorm( c(0.25, 0.75))
x
```

```
:> [1] -0.6744898  0.6744898
```

```
# 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.

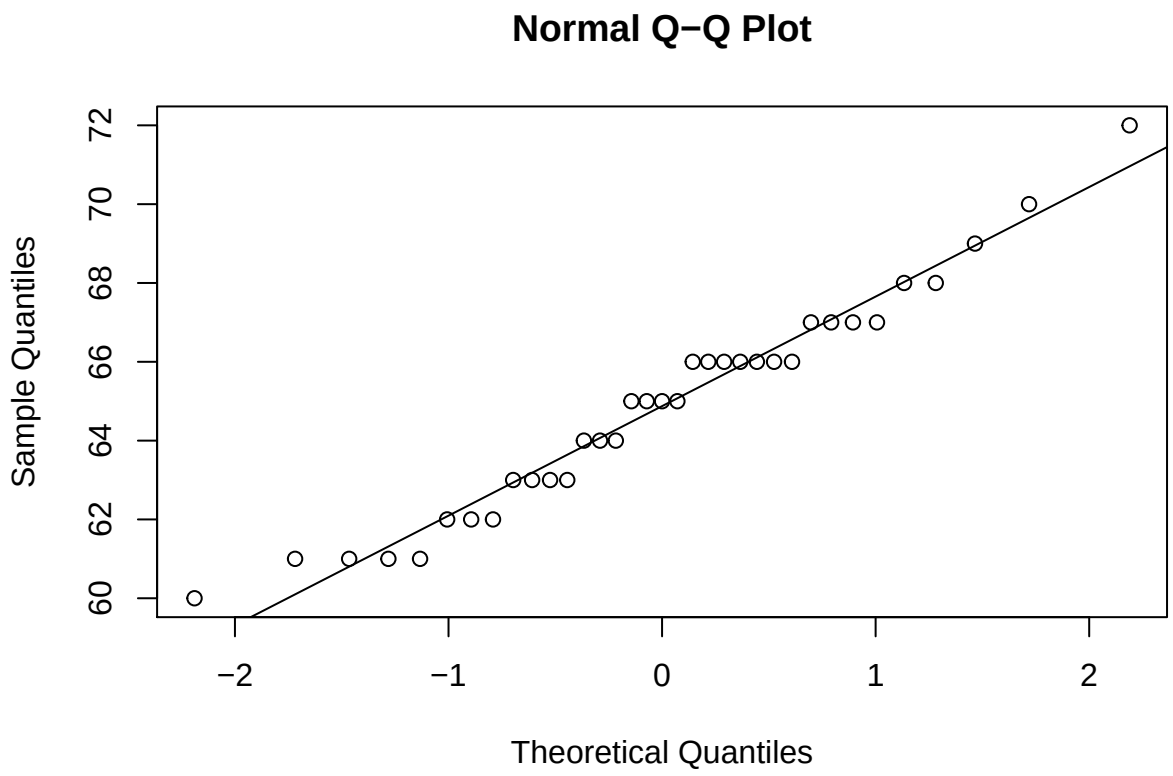
```
plot( alto ~ x.norm, type="p", xlab="Normal quantiles", pch=20)
abline(a=int, b=slope )
```



12.2 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
```

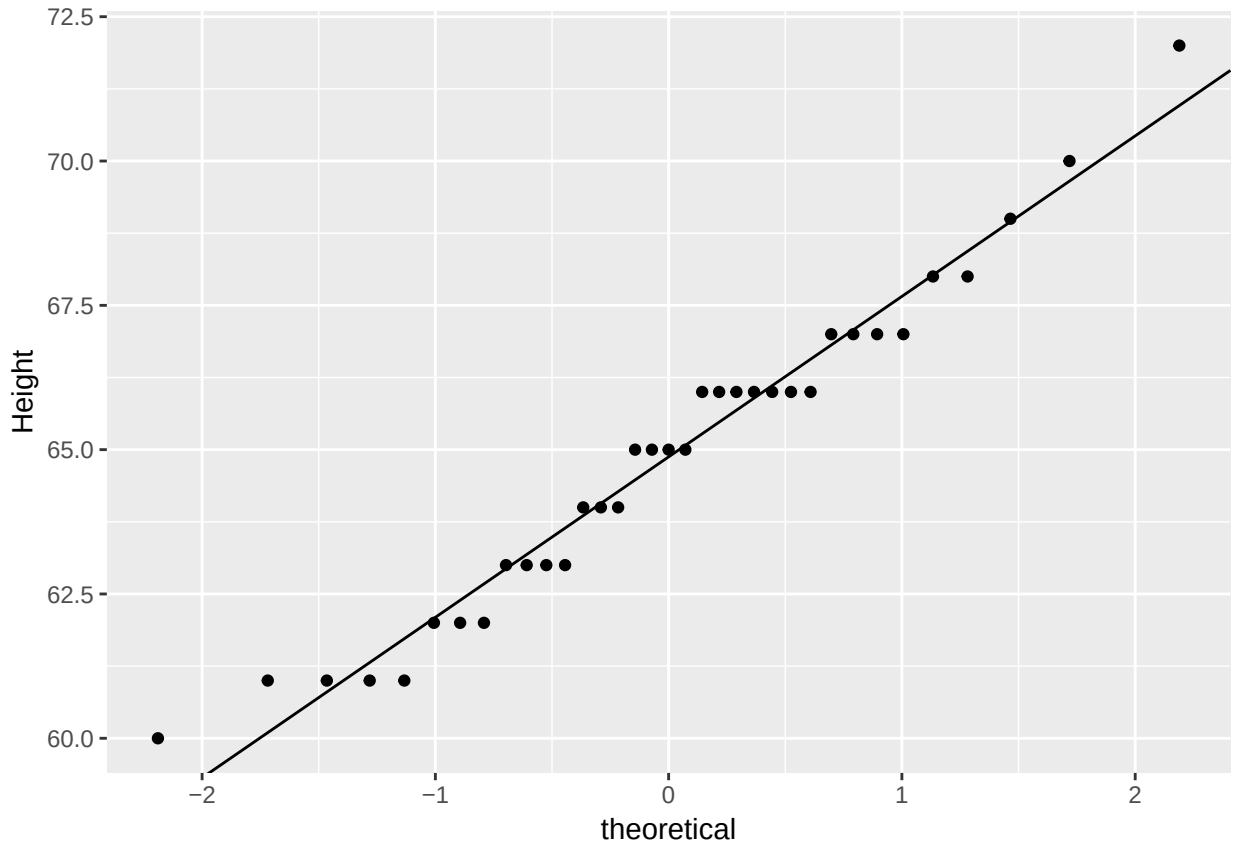


That's it. Just two lines of code!

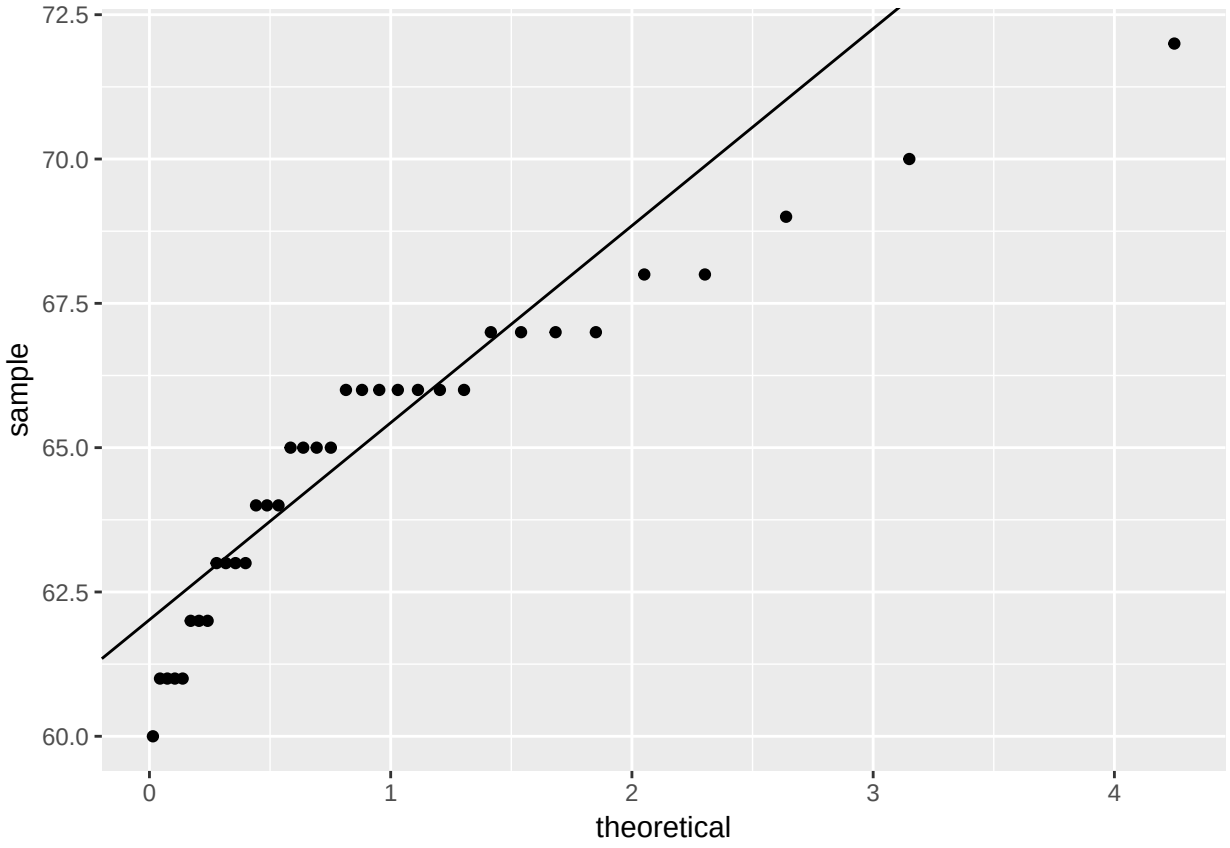
12.3 Using the ggplot2 plotting environment

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.

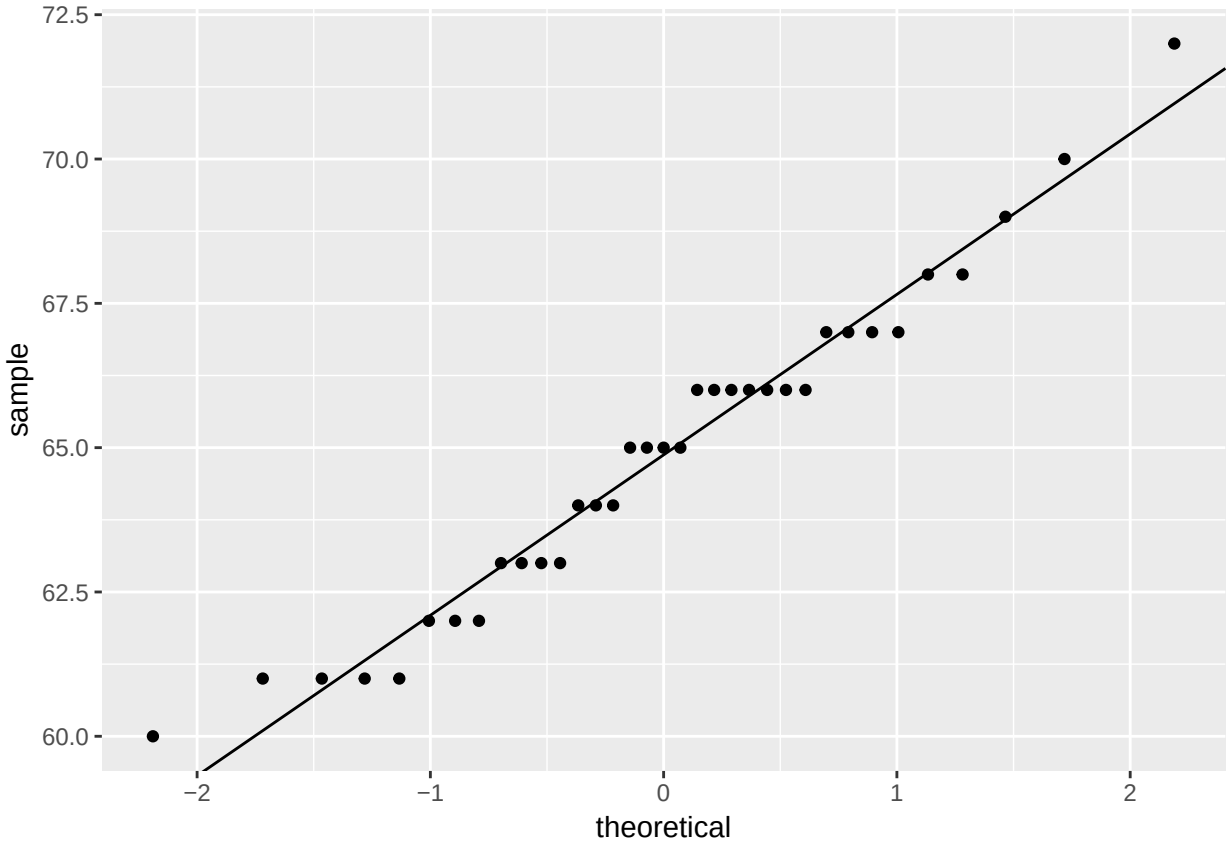
```
# normal distribution
library(ggplot2)

# 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")
```




```r
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.

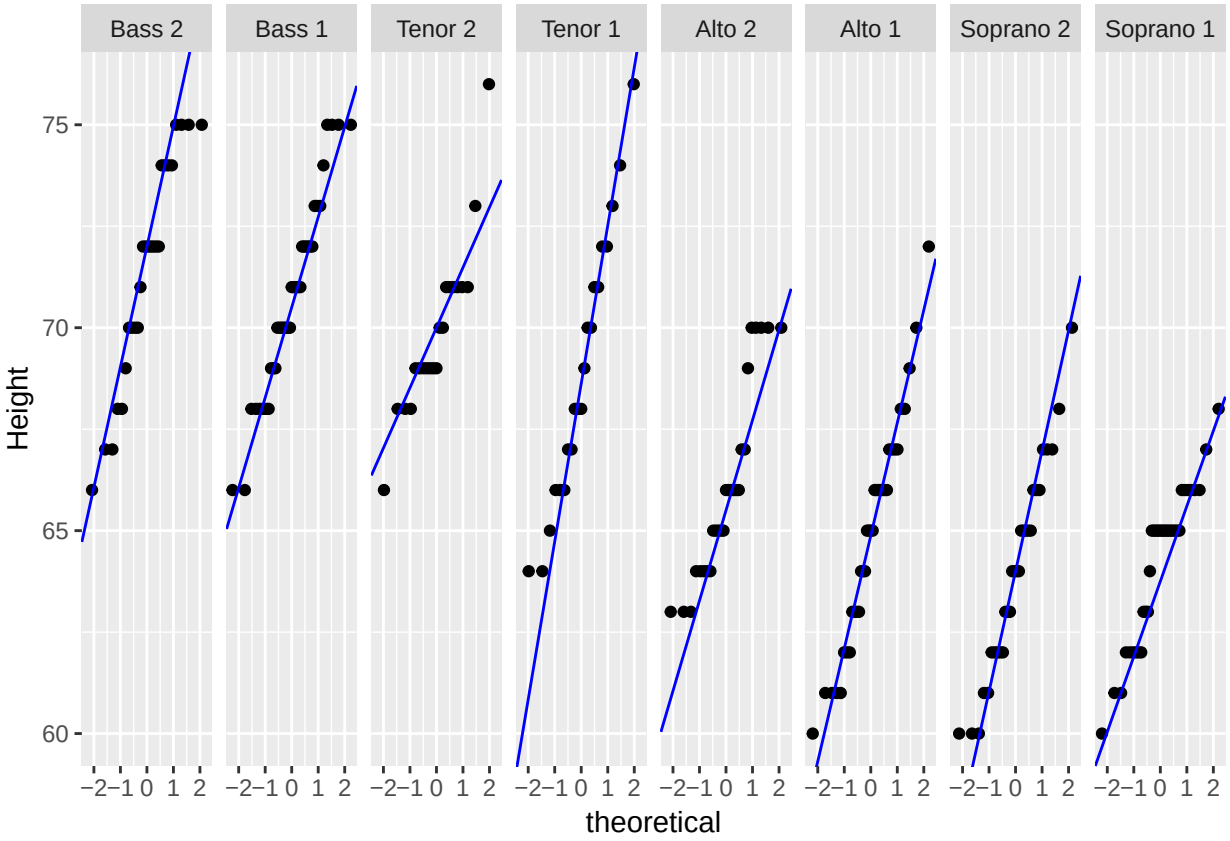
```
library(dplyr)

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
:> 7 Soprano 2   2.97  64
```

```
:> 8 Soprano 1   1.85  63.8
```

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 13

QQ and PP Plots

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

13.1 QQ Plot

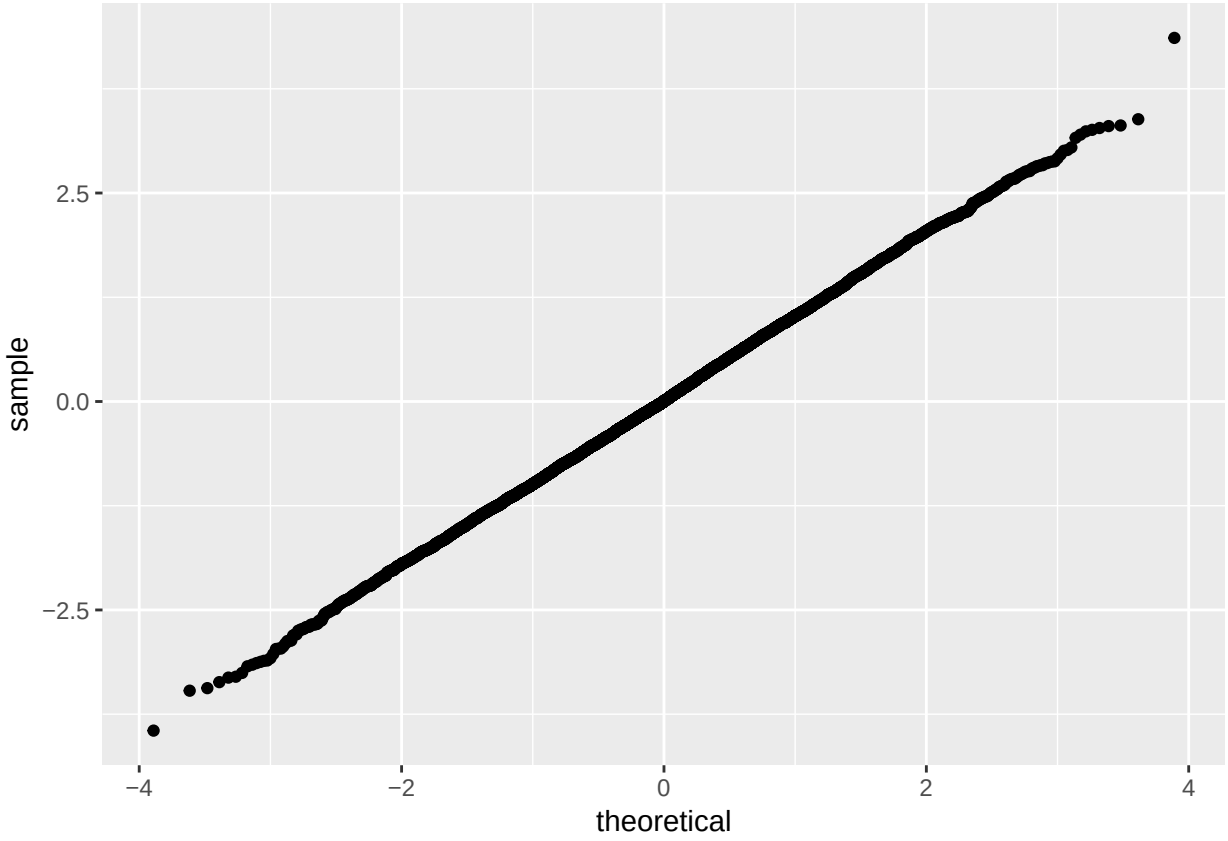
One way to assess how well a particular theoretical model describes a data distribution is to plot data quantiles against theoretical quantiles.

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:

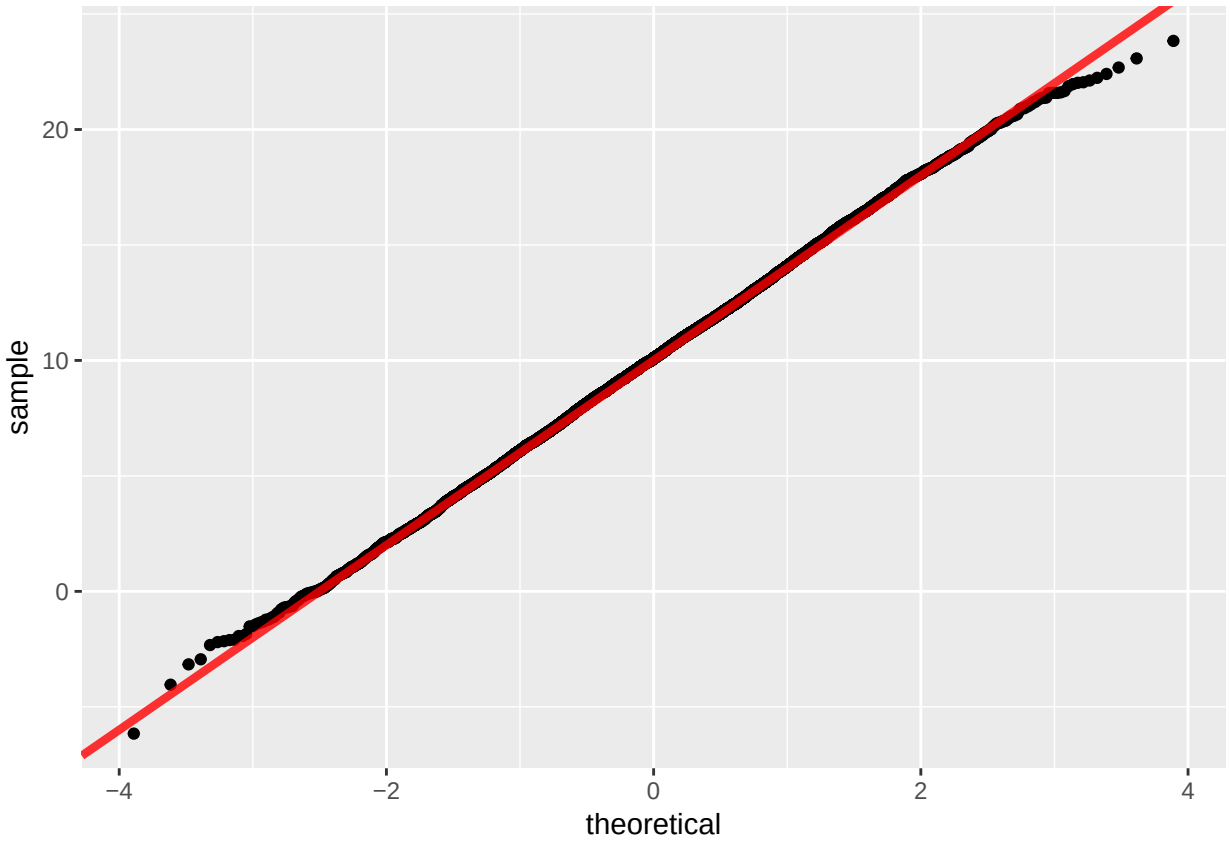
```
library(ggplot2)

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:

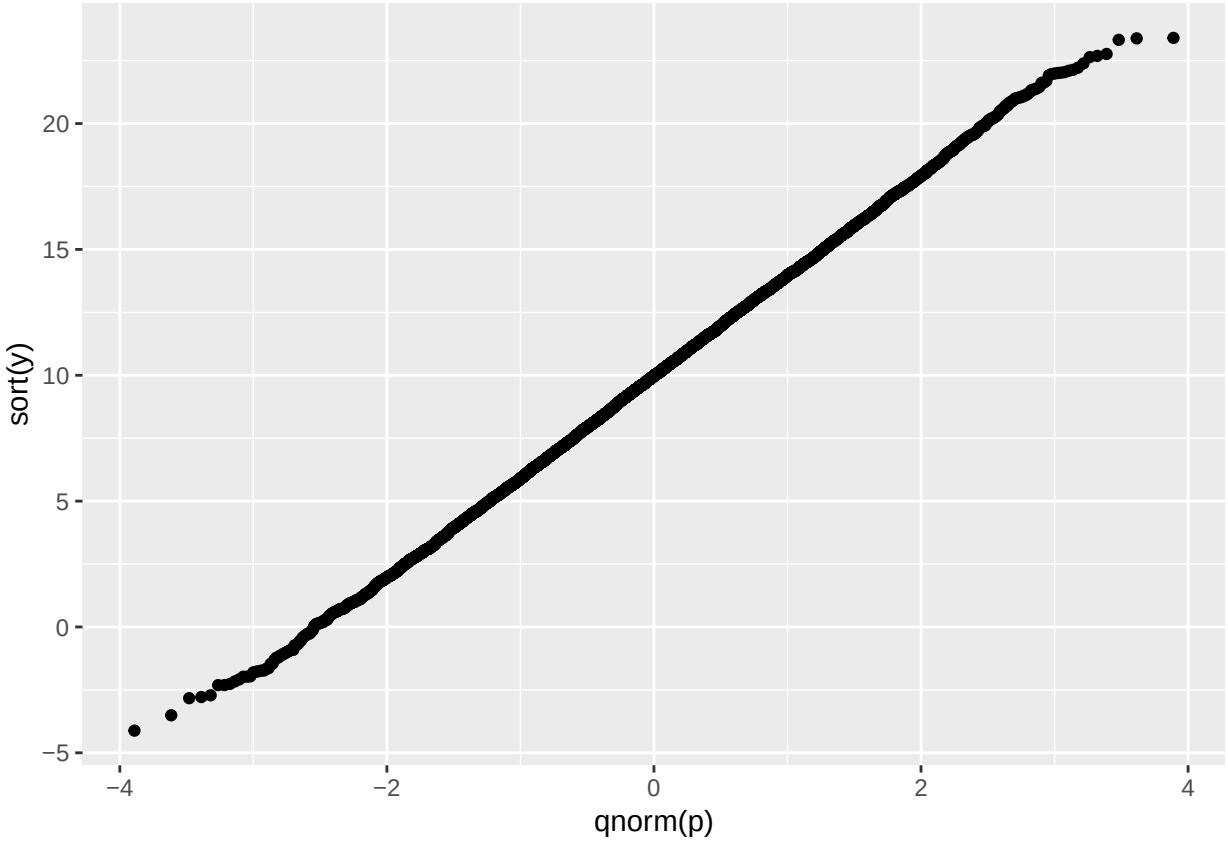
```
ggplot() +
    geom_qq(aes(sample = rnorm(n, 10, 4))) +
    geom_abline(intercept = 10, slope = 4,
                color = "red", size = 1.5, alpha = 0.8)
```



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

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

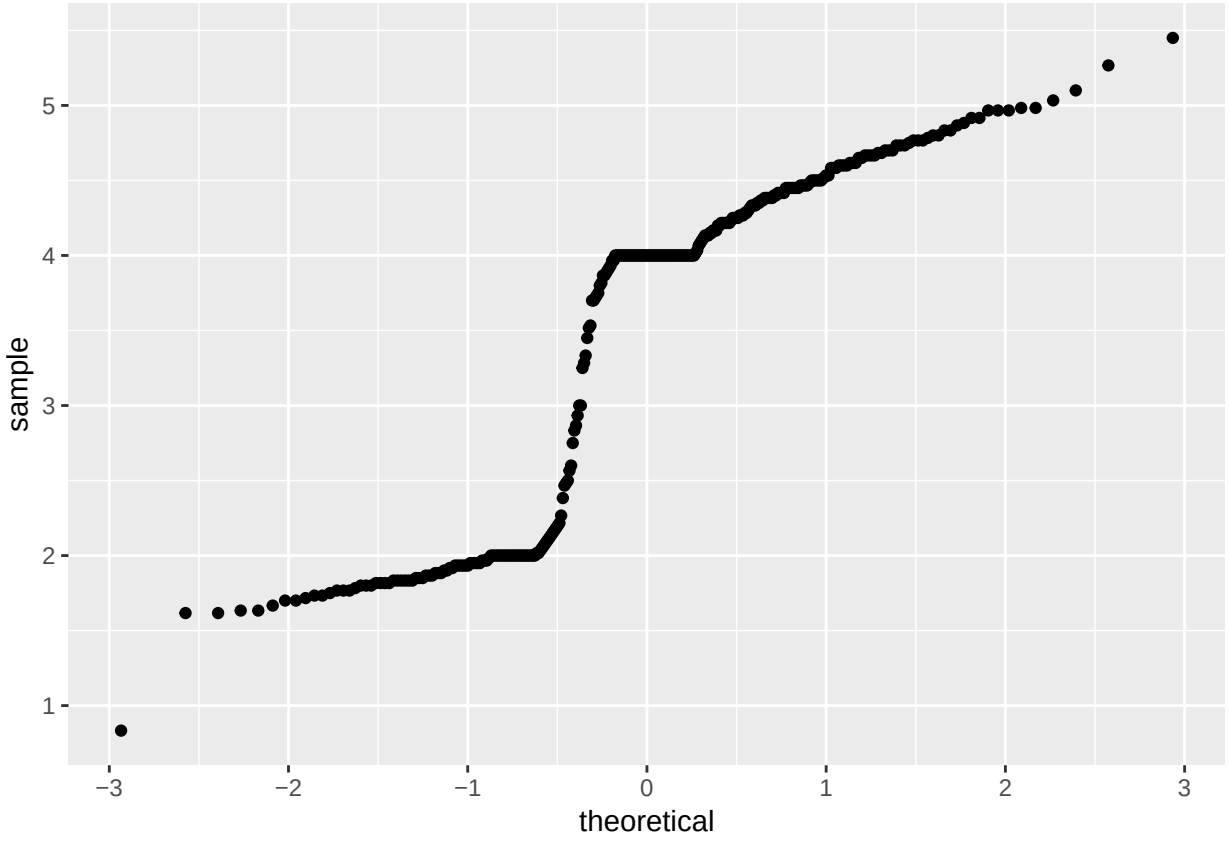
```
p <- (1 : n) / n - 0.5 / n
y <- rnorm(n, 10, 4)
ggplot() + geom_point(aes(x = qnorm(p), y = sort(y)))
```



13.2 Some Examples

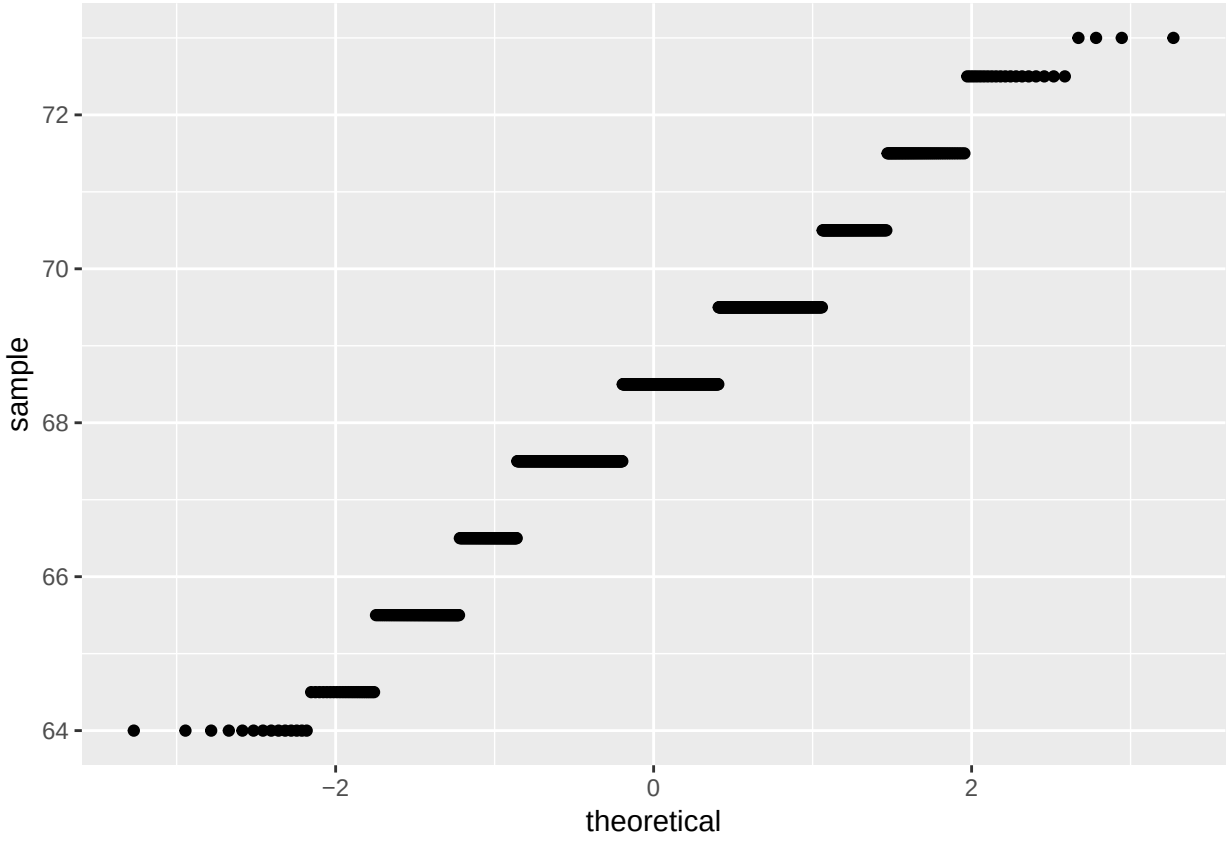
The histograms and density estimates for the duration variable in the `geyser` data set showed that the distribution is far from a normal distribution, and the normal QQ plot shows this as well:

```
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))
```

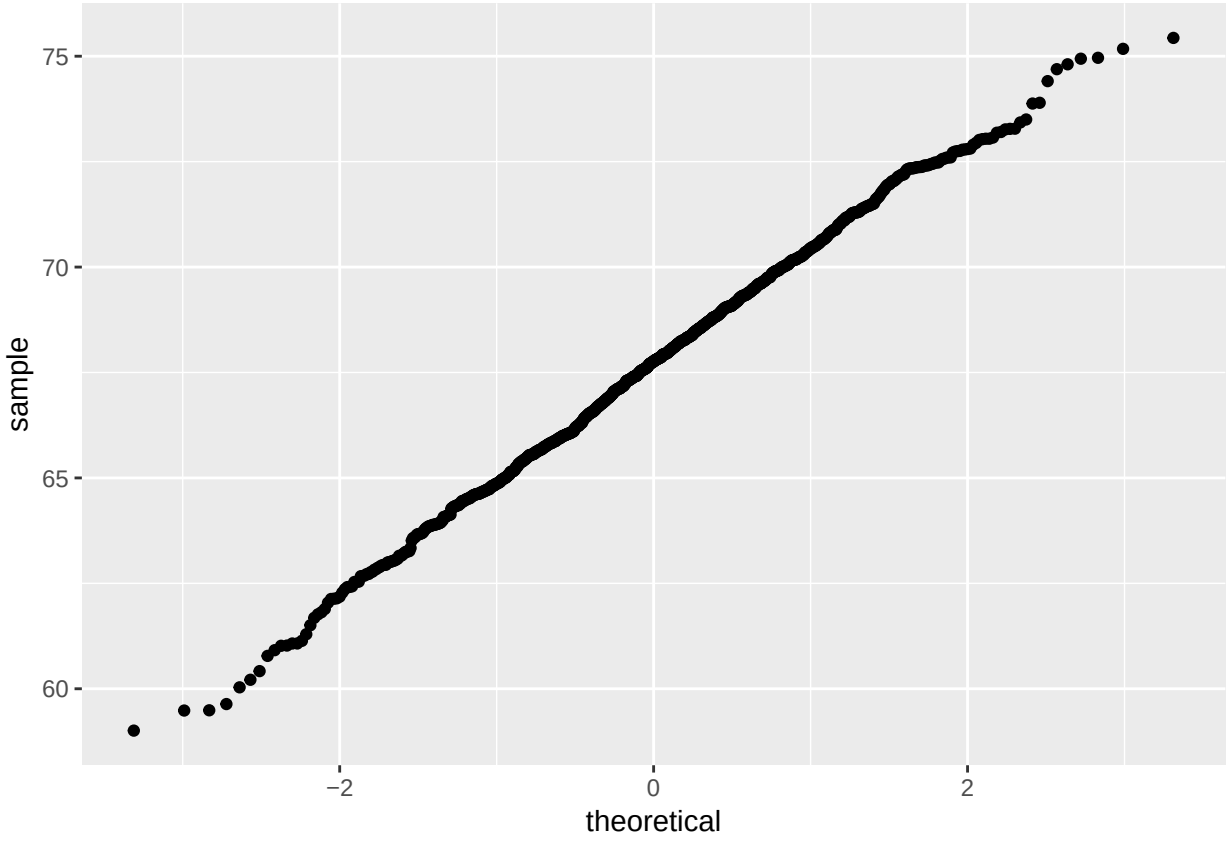


Rounding interferes more with this visualization than with a histogram or a density plot.

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?

13.3 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
```

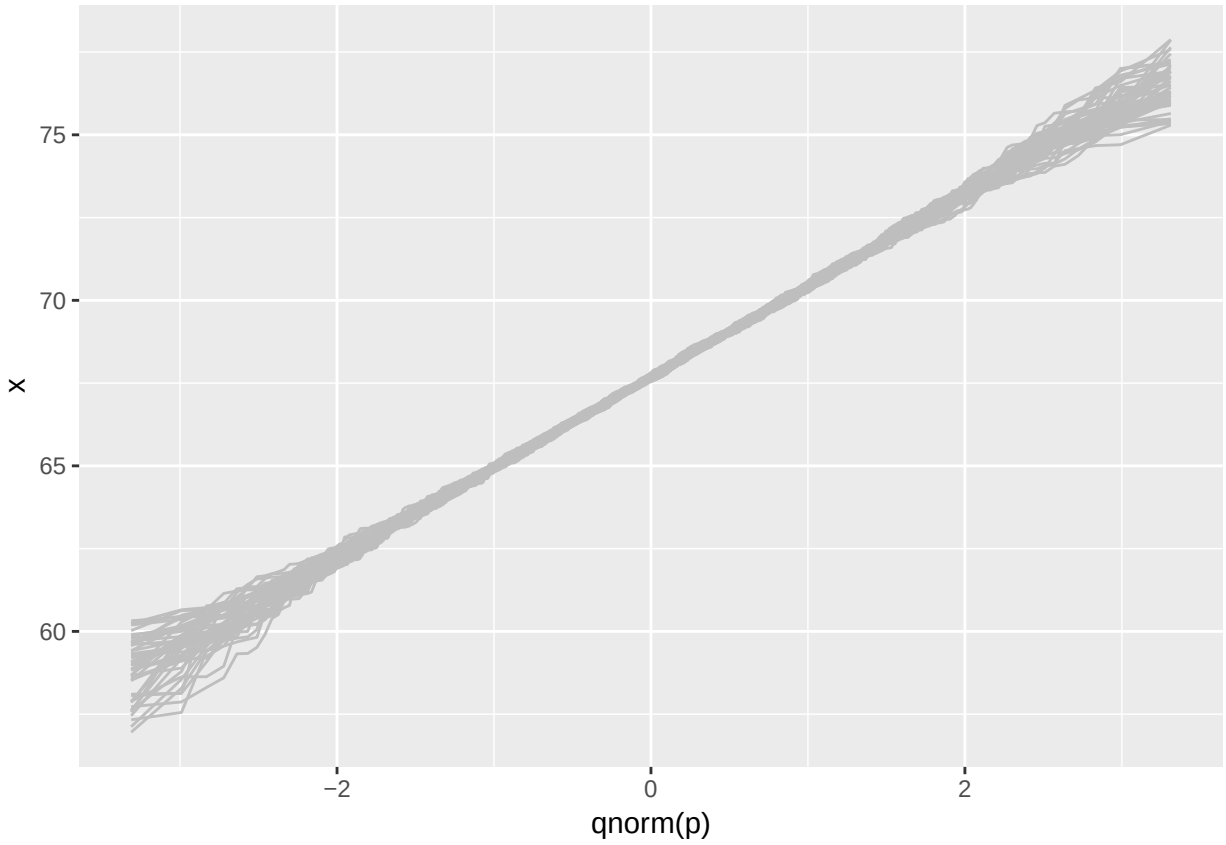
```
                        data.frame(x = xx, p = p, sim = i)
    }))
}
```

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)
```

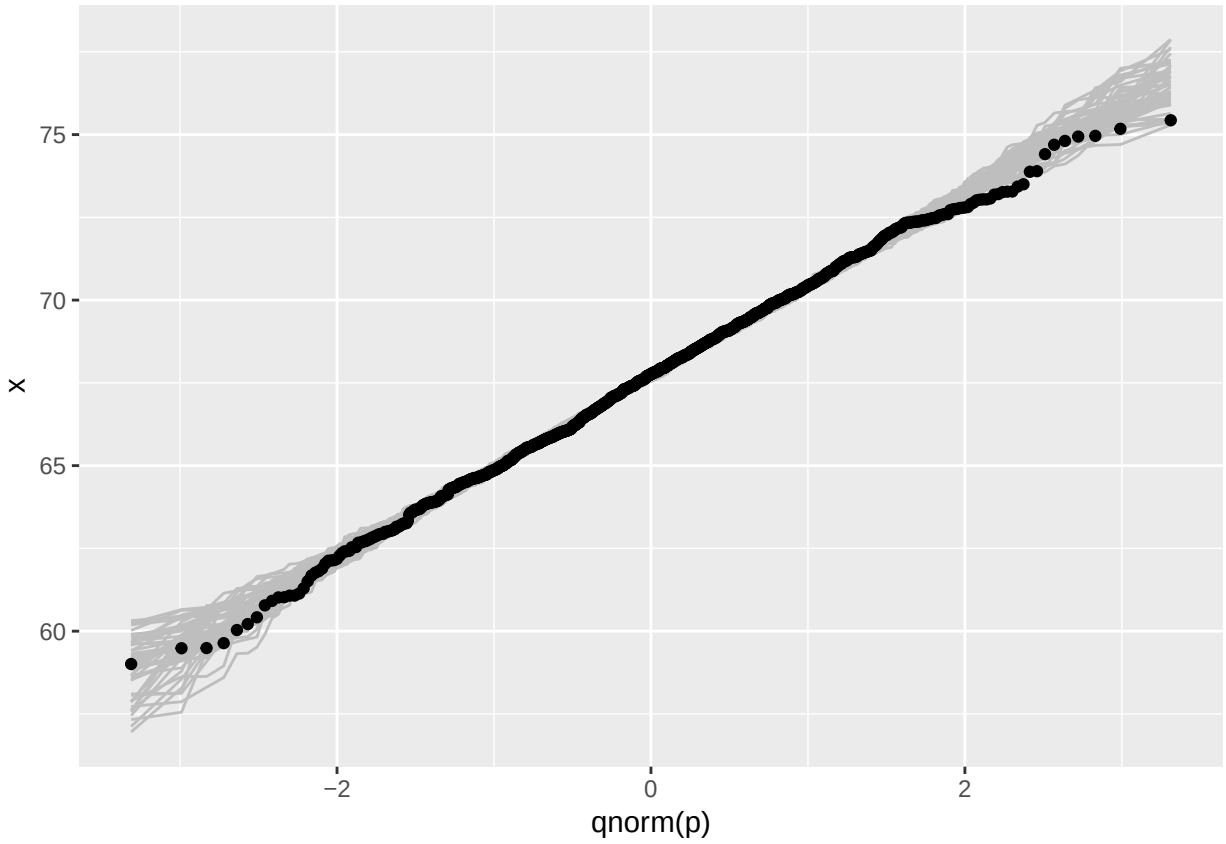
```
:> # A tibble: 53,900 x 3
:>        x        p   sim
:>    <dbl>    <dbl> <int>
:>  1  60.2 0.000464     1
:>  2  60.7 0.00139      1
:>  3  60.7 0.00232      1
:>  4  60.7 0.00325      1
:>  5  60.8 0.00417      1
:>  6  60.8 0.00510      1
:>  7  61.2 0.00603      1
:>  8  61.2 0.00696      1
:>  9  61.2 0.00788      1
:> 10  61.4 0.00881      1
:> # ... with 53,890 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:

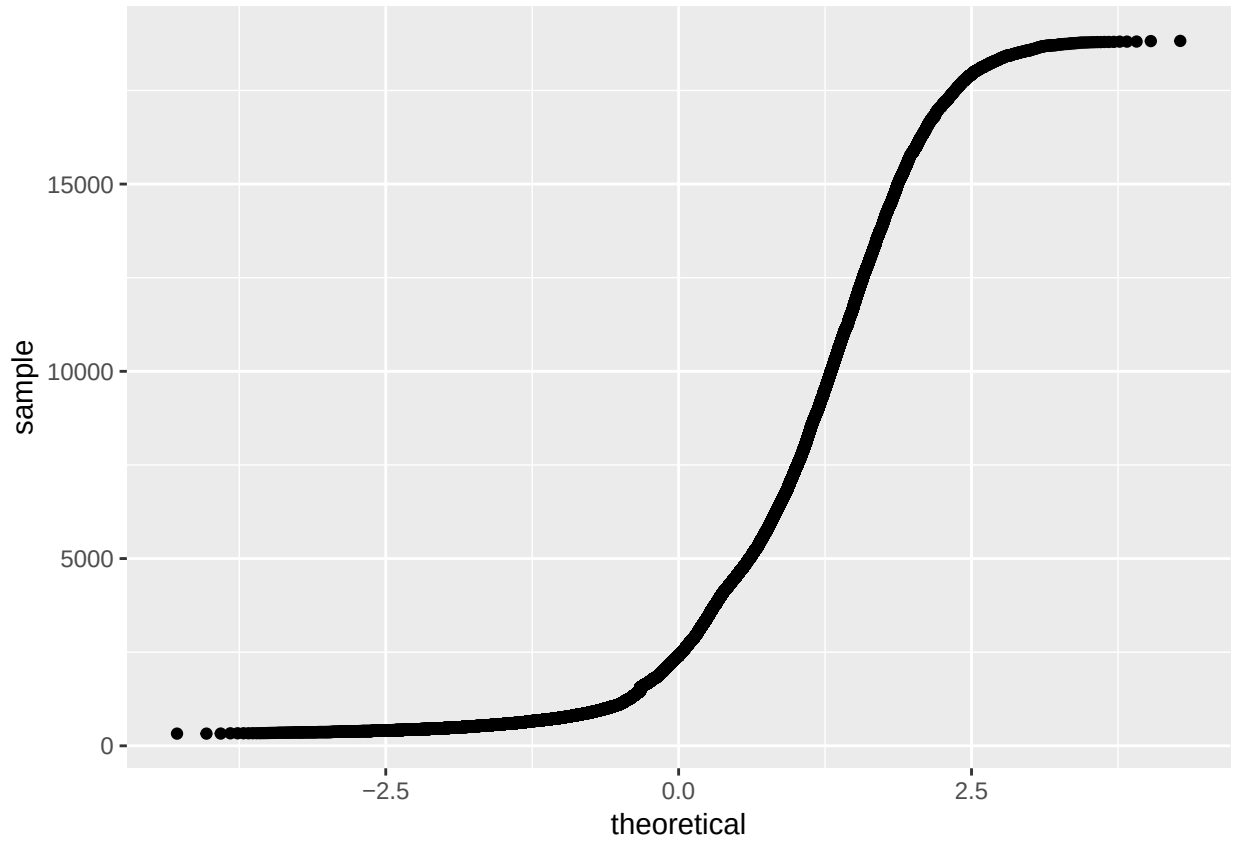
```
ggplot(father.son) +
    geom_line(aes(x = qnorm(p), y = x, group = sim),
              color = "gray", data = gb) +
    geom_qq(aes(sample = fheight))
```



13.4 Scalability

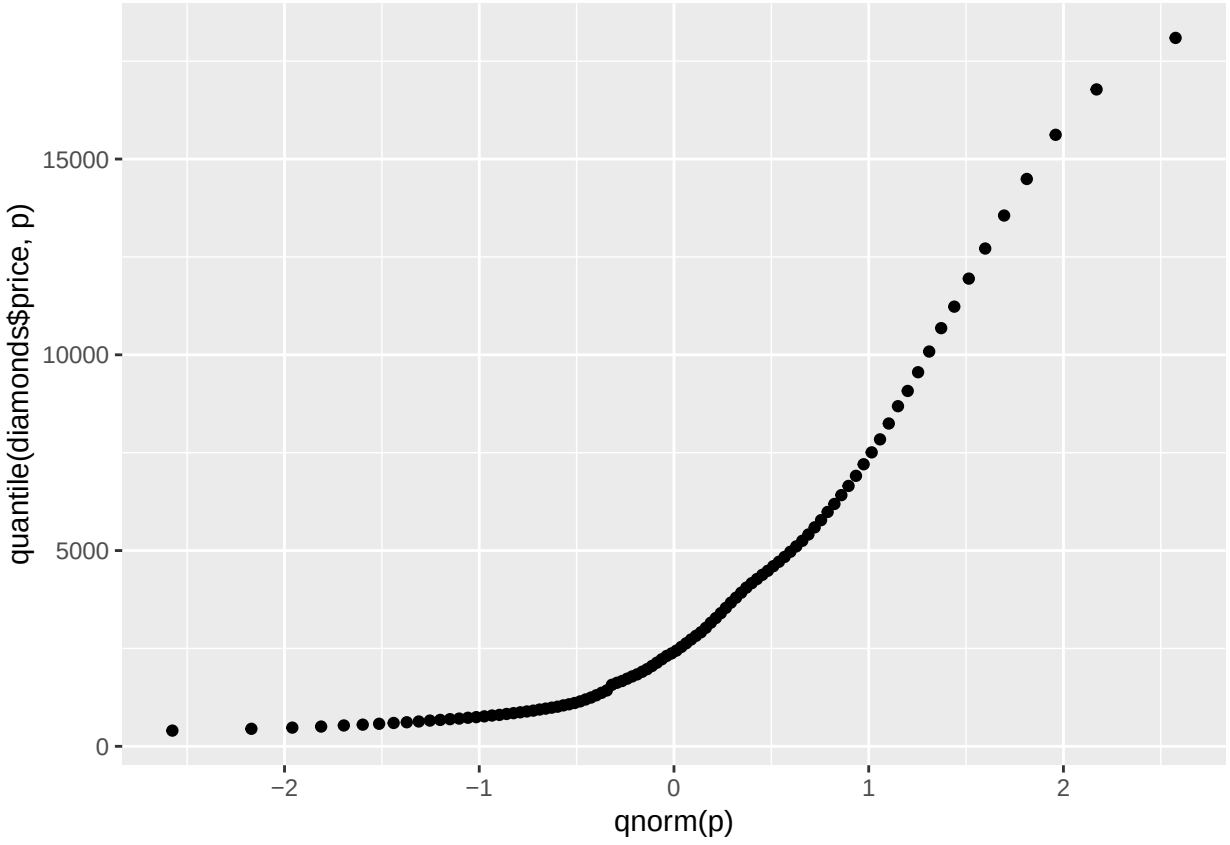
For large sample sizes overplotting will occur:

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



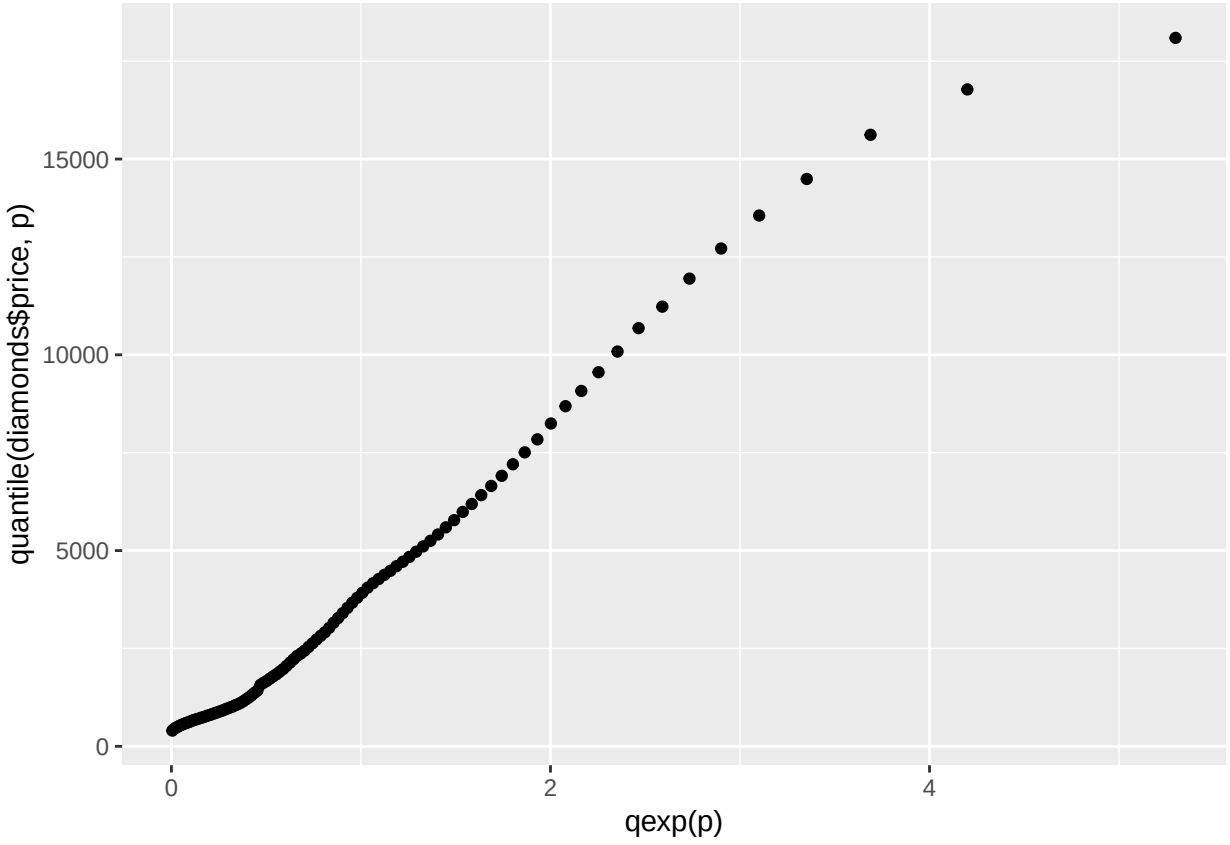
This can be alleviated by using a grid of quantiles:

```
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)))
```



13.5 Comparing Two Distributions

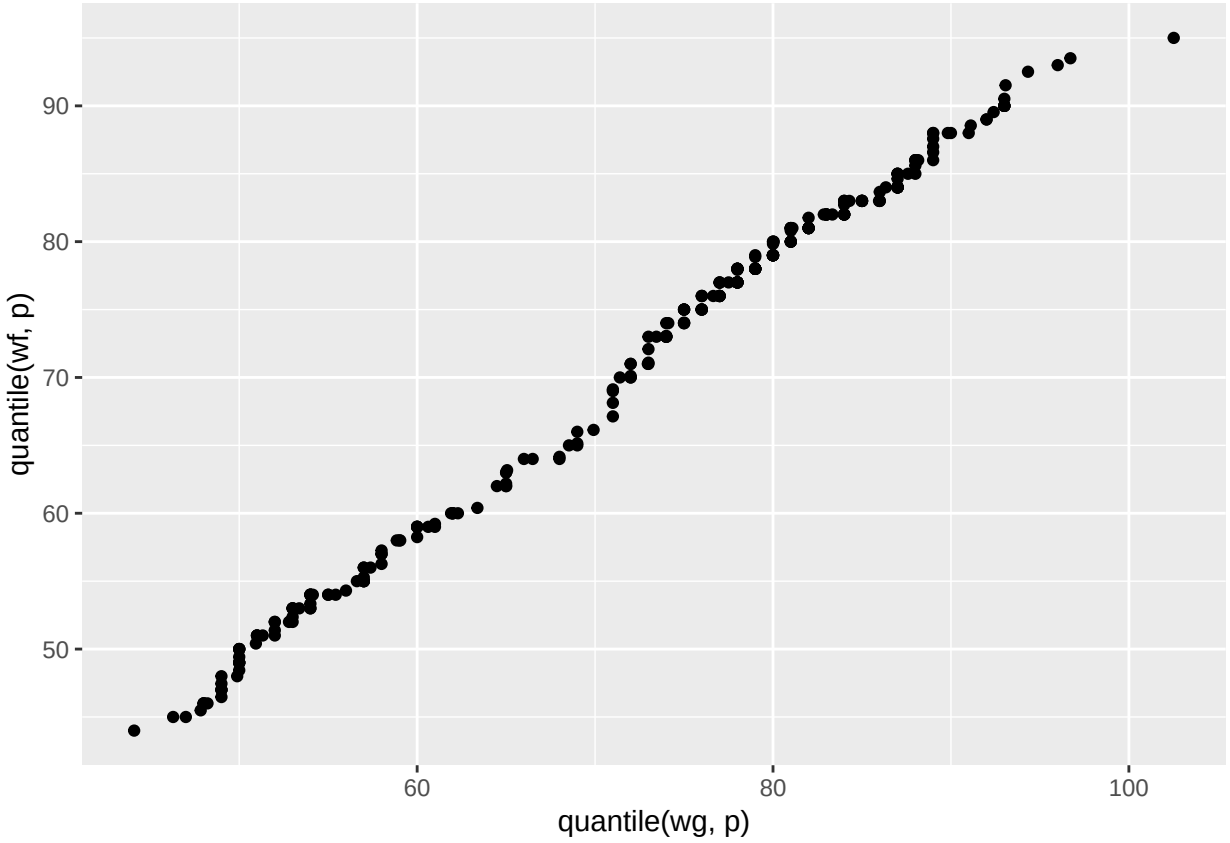
The QQ plot can also be used to compare two distributions based on a sample from each.

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)))
```


90
80
70
60
50
quantile(wf, p)
60
80
100
quantile(wg, p)

Chapter 14

PP Plots

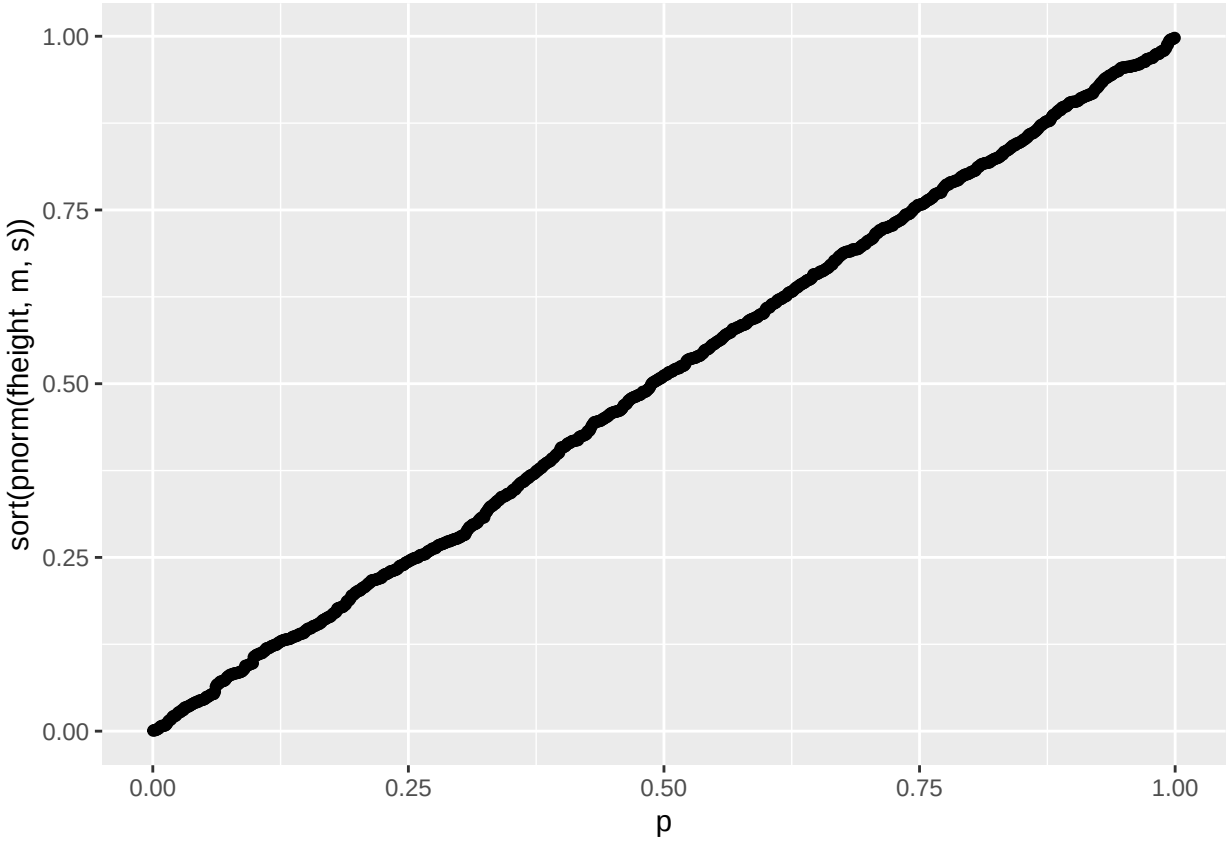
The PP plot for comparing a sample to a theoretical model plots the theoretical proportion less than or equal to each observed value against the actual proportion.

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))))
```



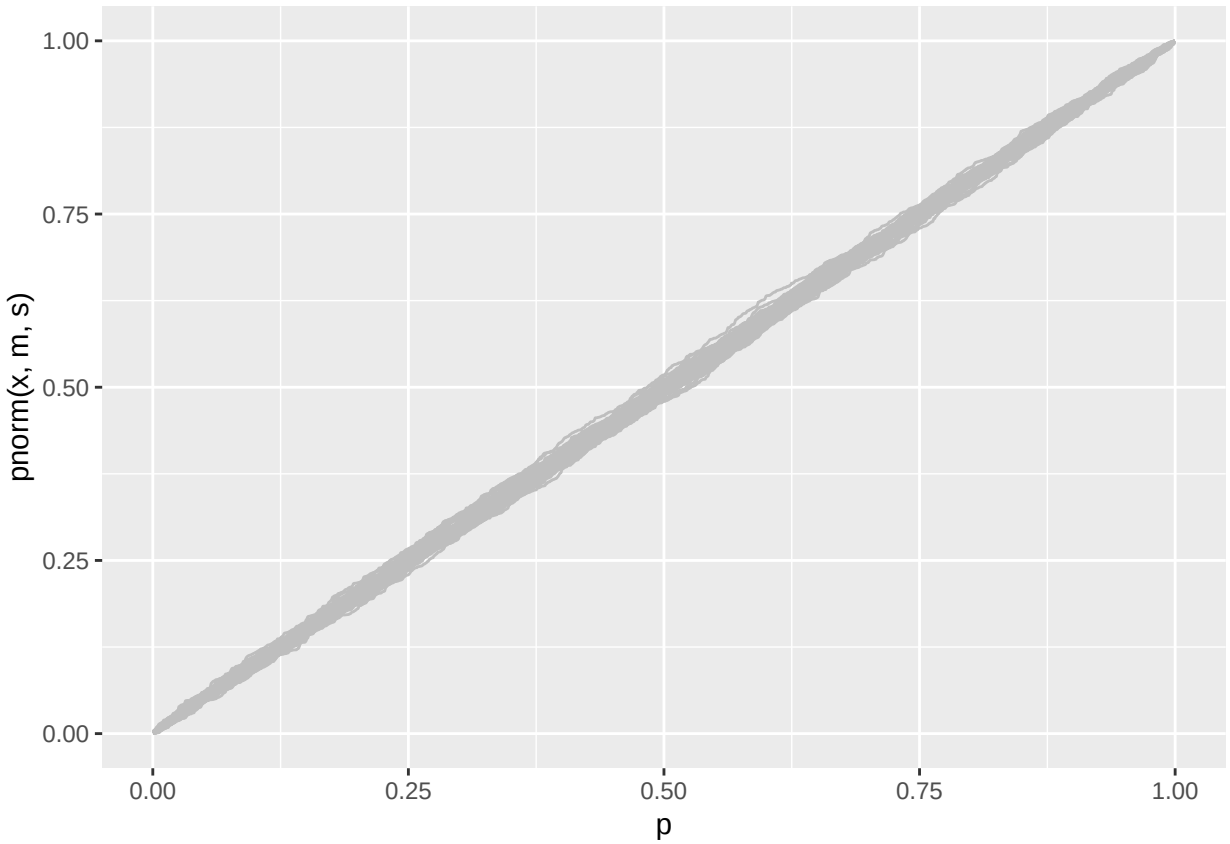
- The values on the vertical axis are the probability integral transform of the data for the theoretical distribution.

- 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.

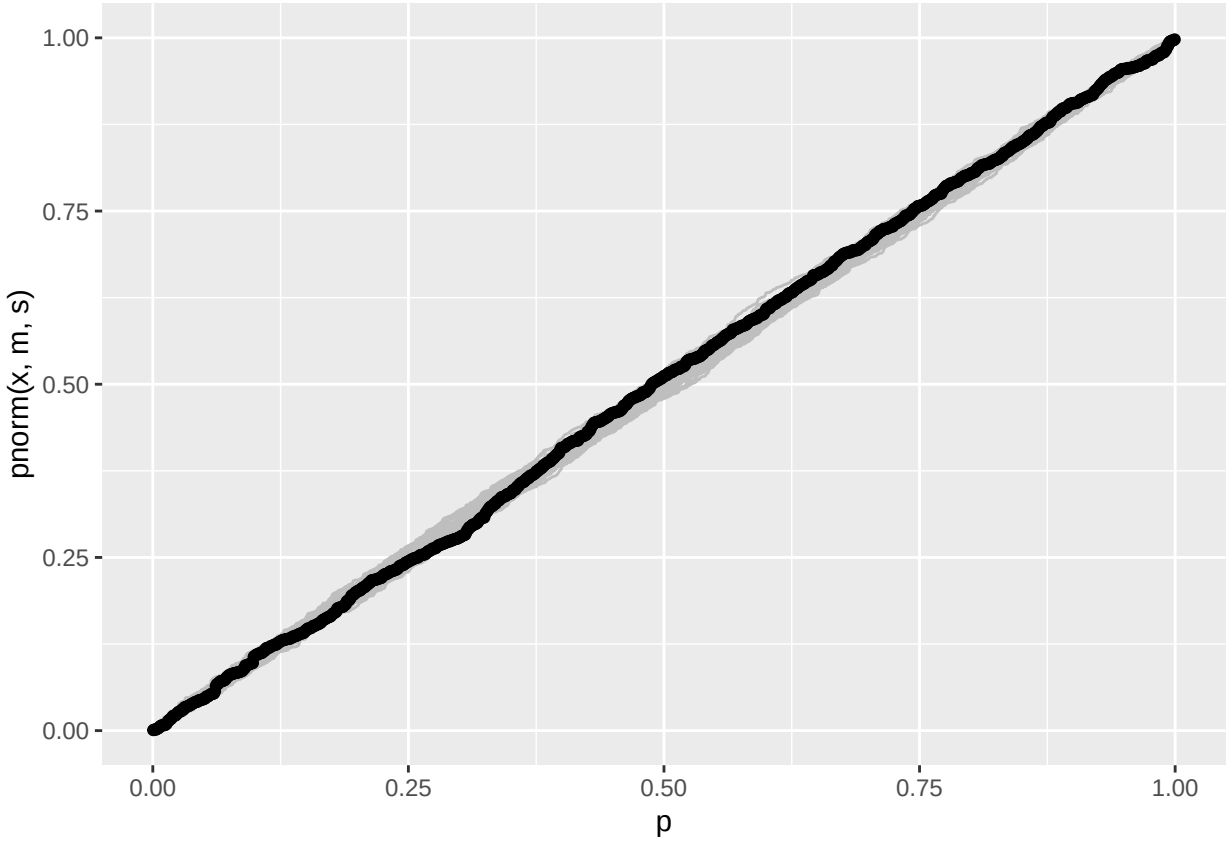
- The PP plot goes through the points (0,0) and (1,1) and so is much less variable in the tails:

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



Adding the data:

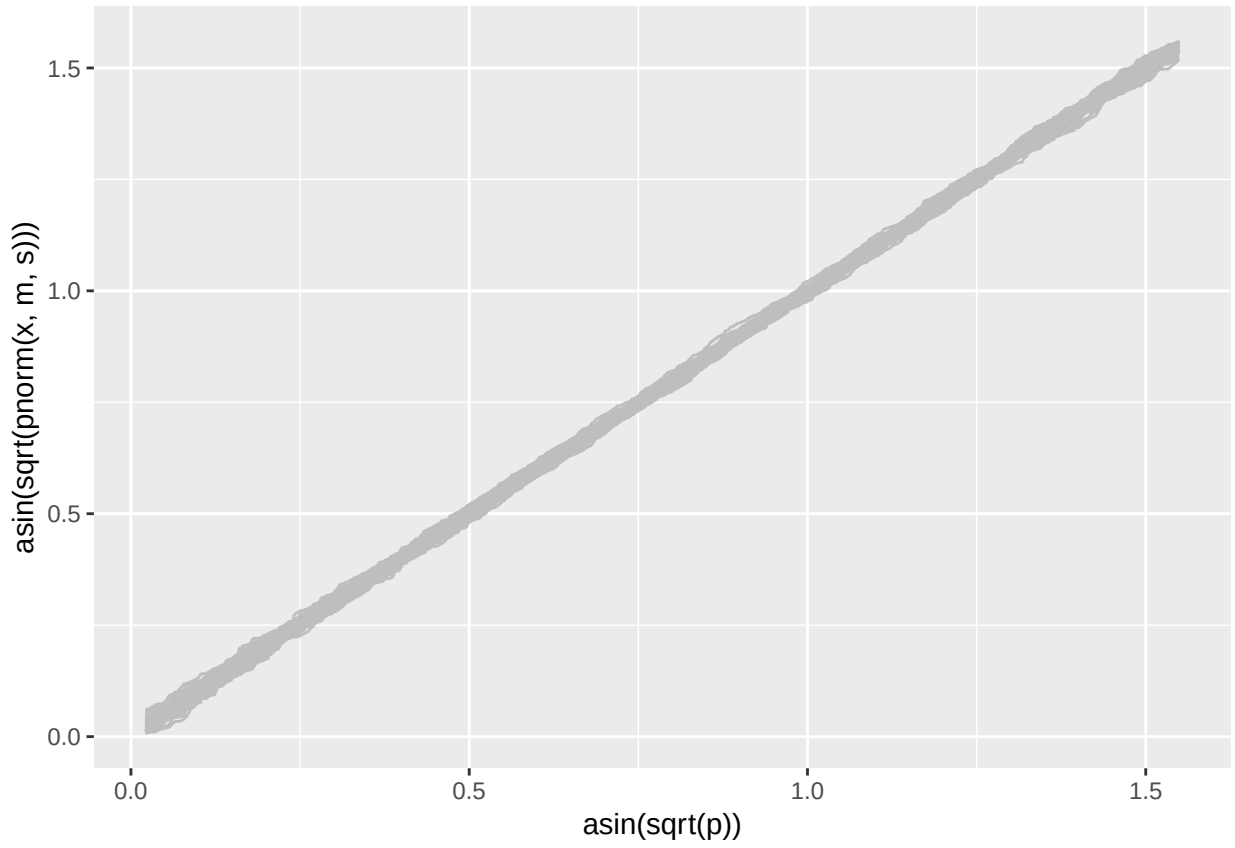
```
pp +
geom_point(aes(x = p, y = sort(pnorm(fheight, m, s))), data = (father.son))
```



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

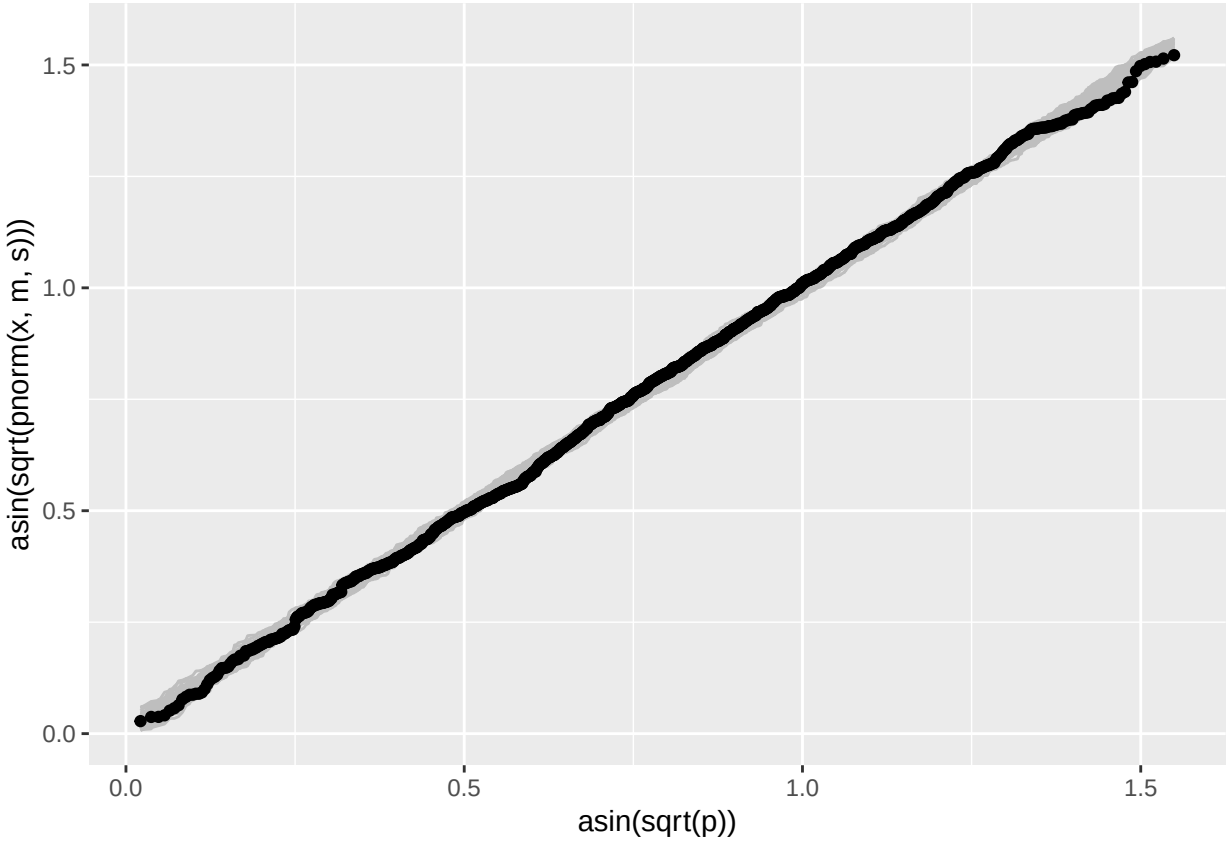
A compromise between the QQ and PP plots uses the arcsine square root variance-stabilizing transformation, which makes the variability approximately constant across the range of the plot:

```
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))
```


1.5
1.0
0.5
0.0
asin(sqrt(pnorm(x, m, s)))
0.0
0.5
1.0
1.5
asin(sqrt(p))

Chapter 15

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:

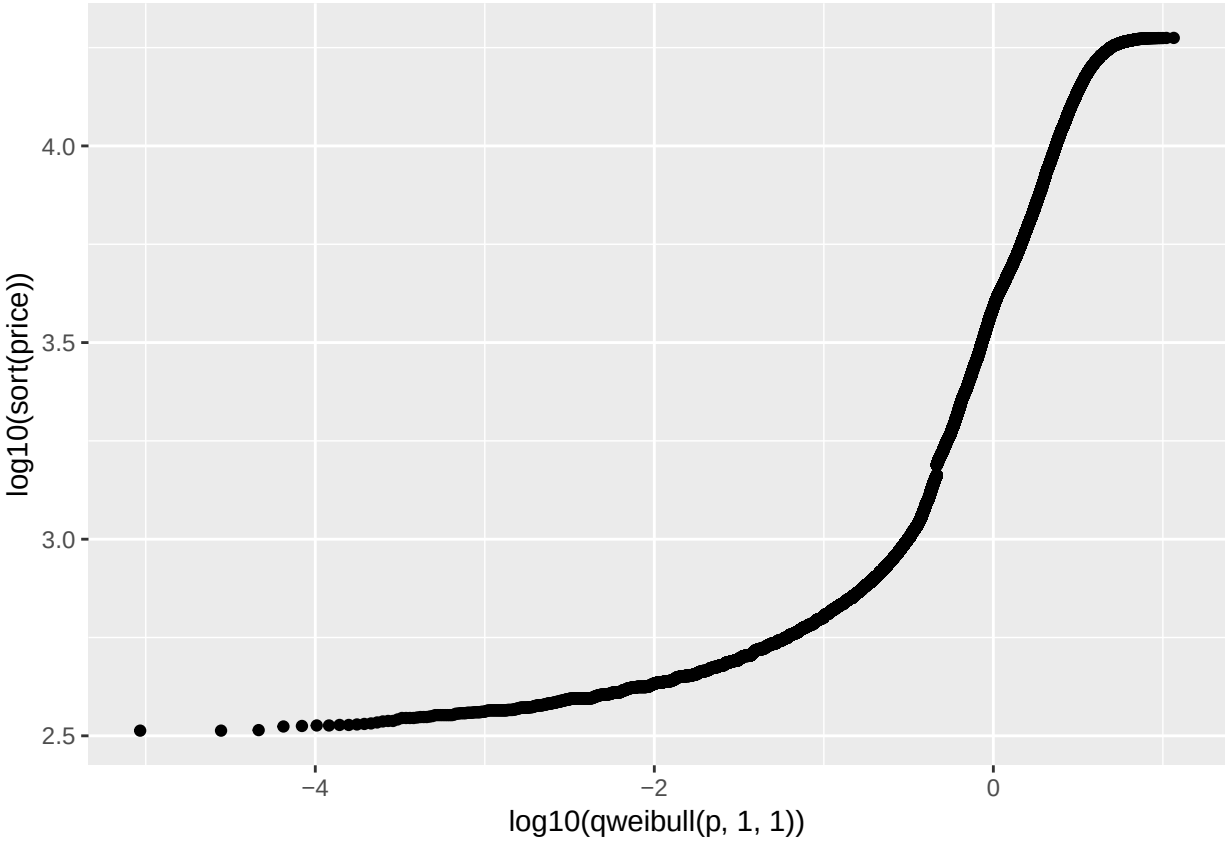
The Weibull family is widely used in reliability modeling; its CDF is

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

- 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 16

Compare classification algorithms

```
# load packages
library(mlbench)
library(caret)
# load the dataset
data(PimaIndiansDiabetes)
```

```
dplyr::glimpse(PimaIndiansDiabetes)
```

```
#> Observations: 768
#> Variables: 9
#> $ pregnant <dbl> 6, 1, 8, 1, 0, 5, 3, 10, 2, 8, 4, 10, 10, 1, 5, 7, 0,...
#> $ glucose  <dbl> 148, 85, 183, 89, 137, 116, 78, 115, 197, 125, 110, 1...
#> $ pressure <dbl> 72, 66, 64, 66, 40, 74, 50, 0, 70, 96, 92, 74, 80, 60...
#> $ triceps  <dbl> 35, 29, 0, 23, 35, 0, 32, 0, 45, 0, 0, 0, 0, 23, 19, ...
#> $ insulin  <dbl> 0, 0, 0, 94, 168, 0, 88, 0, 543, 0, 0, 0, 0, 846, 175...
#> $ mass     <dbl> 33.6, 26.6, 23.3, 28.1, 43.1, 25.6, 31.0, 35.3, 30.5,...
#> $ pedigree <dbl> 0.627, 0.351, 0.672, 0.167, 2.288, 0.201, 0.248, 0.13...
#> $ age      <dbl> 50, 31, 32, 21, 33, 30, 26, 29, 53, 54, 30, 34, 57, 5...
#> $ diabetes <fct> pos, neg, pos, neg, pos, neg, pos, neg, pos, pos, neg...
```

```
tibble::as_tibble(PimaIndiansDiabetes)
```

```
#> # A tibble: 768 x 9
#>    pregnant glucose pressure triceps insulin  mass pedigree   age diabetes
#>       <dbl>   <dbl>    <dbl>   <dbl>   <dbl> <dbl>    <dbl> <dbl> <fct>
#>  1        6     148       72      35       0  33.6    0.627    50 pos
#>  2        1      85       66      29       0  26.6    0.351    31 neg
#>  3        8     183       64       0       0  23.3    0.672    32 pos
#>  4        1      89       66      23      94  28.1    0.167    21 neg
#>  5        0     137       40      35     168  43.1    2.29     33 pos
#>  6        5     116       74       0       0  25.6    0.201    30 neg
#>  7        3      78       50      32      88  31      0.248    26 pos
#>  8       10     115        0       0       0  35.3    0.134    29 neg
#>  9        2     197       70      45     543  30.5    0.158    53 pos
#> 10        8     125       96       0       0   0      0.232    54 pos
#> # ... with 758 more rows
```

16.1 Train the models

```
# prepare training scheme
trainControl <- trainControl(method = "repeatedcv", number=10, repeats=3)
```

```
# CART
set.seed(7)
fit.cart <- train(diabetes~., data=PimaIndiansDiabetes,
                  method = "rpart", trControl=trainControl)
```

```
# LDA: Linear Discriminant Analysis
set.seed(7)
fit.lda <- train(diabetes~., data=PimaIndiansDiabetes,
                 method="lda", trControl=trainControl)
```

```
# SVM
set.seed(7)
fit.svm <- train(diabetes~., data=PimaIndiansDiabetes,
                 method="svmRadial", trControl=trainControl)
```

```
# KNN
set.seed(7)
fit.knn <- train(diabetes~., data=PimaIndiansDiabetes,
                 method="knn", trControl=trainControl)
```

```
# Random Forest
set.seed(7)
fit.rf <- train(diabetes~., data=PimaIndiansDiabetes,
                method="rf", trControl=trainControl)
```

```
# collect resamples
results <- resamples(list(CART=fit.cart,
                          LDA=fit.lda,
                          SVM=fit.svm,
                          KNN=fit.knn,
                          RF=fit.rf))
```

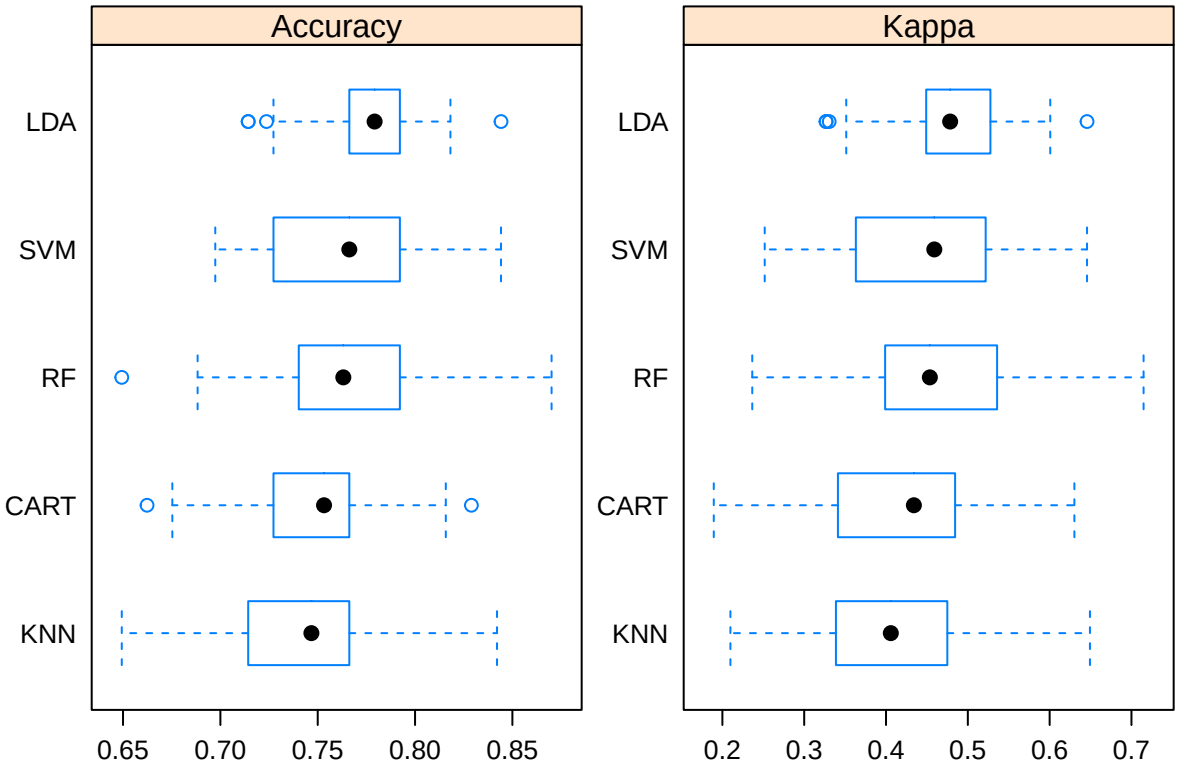
16.2 Compare models

```
# summarize differences between models
summary(results)
```

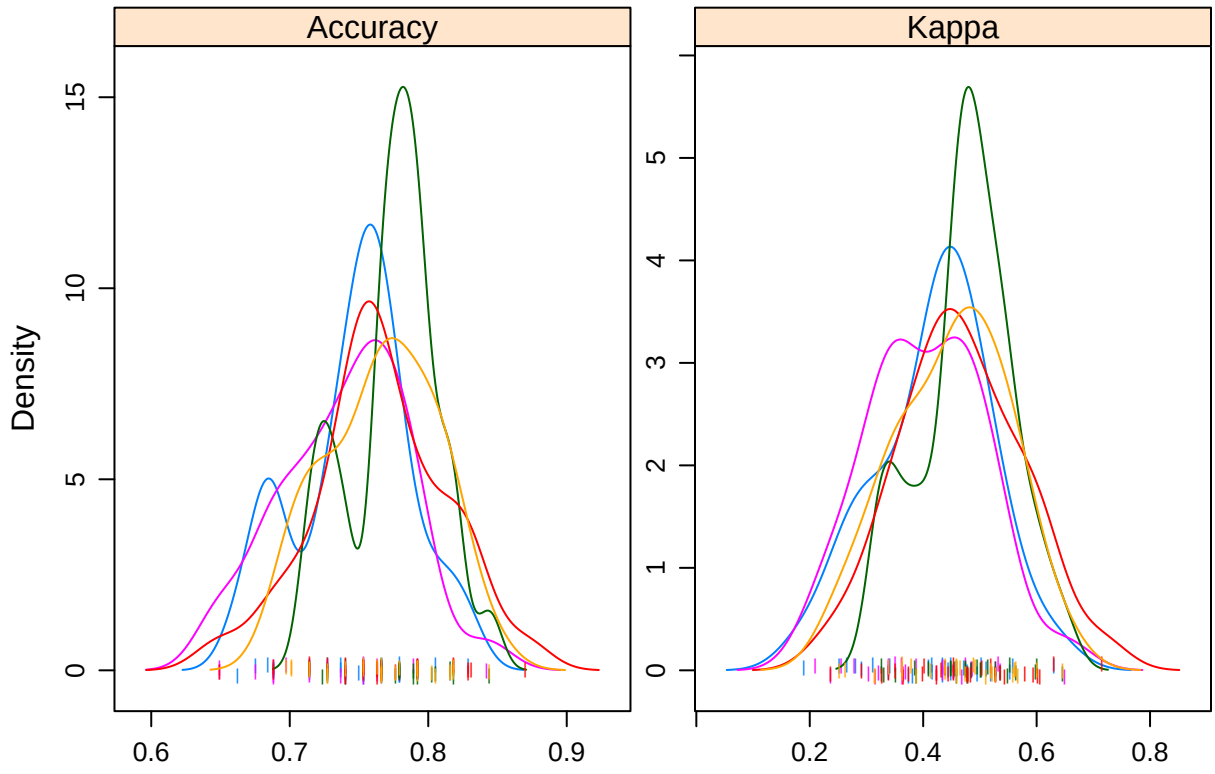
```
#>
#> Call:
#> summary.resamples(object = results)
#>
#> Models: CART, LDA, SVM, KNN, RF
#> Number of resamples: 30
#>
#> Accuracy
#>           Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
#> CART 0.6623377 0.7272727 0.7532468 0.7465596 0.7662338 0.8289474    0
#> LDA  0.7142857 0.7662338 0.7792208 0.7747608 0.7922078 0.8441558    0
```

```
#> SVM  0.6973684 0.7305195 0.7662338 0.7665243 0.7922078 0.8441558    0
#> KNN  0.6493506 0.7142857 0.7467532 0.7400832 0.7662338 0.8421053    0
#> RF   0.6493506 0.7402597 0.7631579 0.7643825 0.7922078 0.8701299    0
#>
#> Kappa
#>           Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
#> CART 0.1894737 0.3495477 0.4340426 0.4184446 0.4827744 0.6302395    0
#> LDA  0.3267091 0.4491256 0.4784008 0.4764965 0.5276074 0.6457055    0
#> SVM  0.2517123 0.3670435 0.4590164 0.4500126 0.5211405 0.6457055    0
#> KNN  0.2098062 0.3388961 0.4058531 0.4047018 0.4733369 0.6492308    0
#> RF   0.2365039 0.3997116 0.4535834 0.4649871 0.5337481 0.7148148    0
```

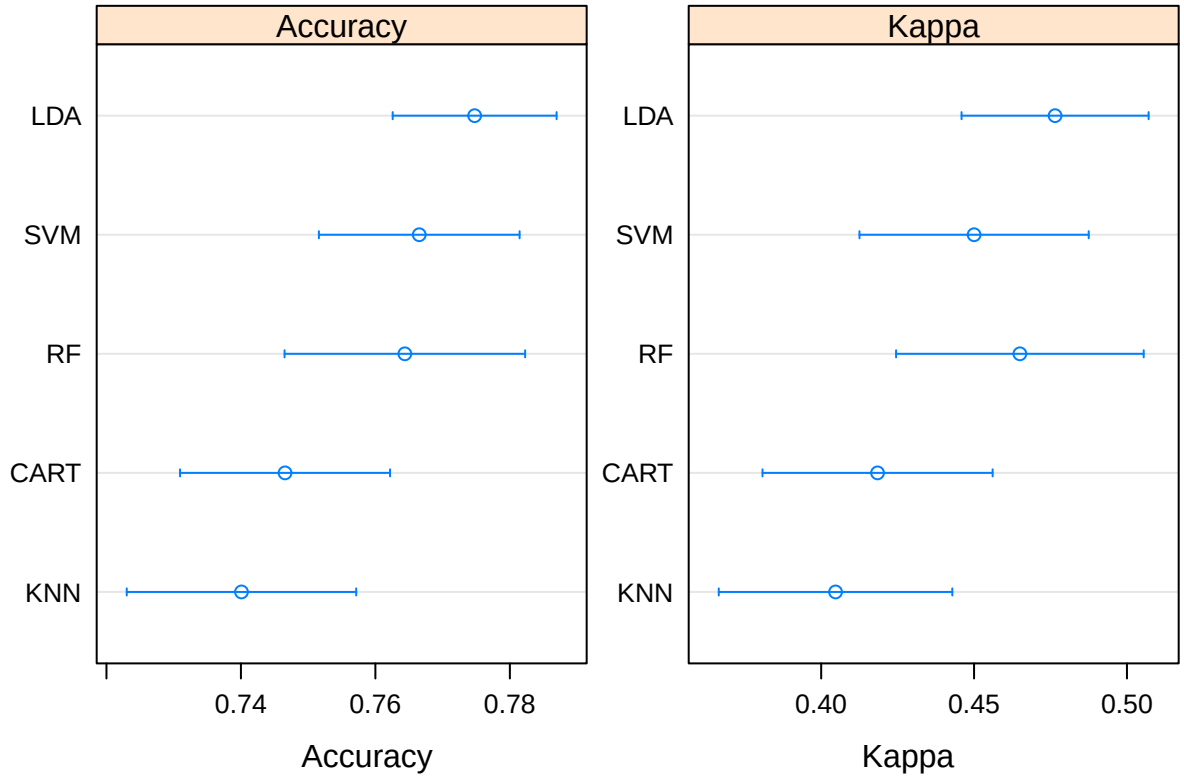
```
# box and whisker plots to compare models
scales <- list(x=list(relation="free"), y=list(relation="free"))
bwplot(results, scales=scales)
```



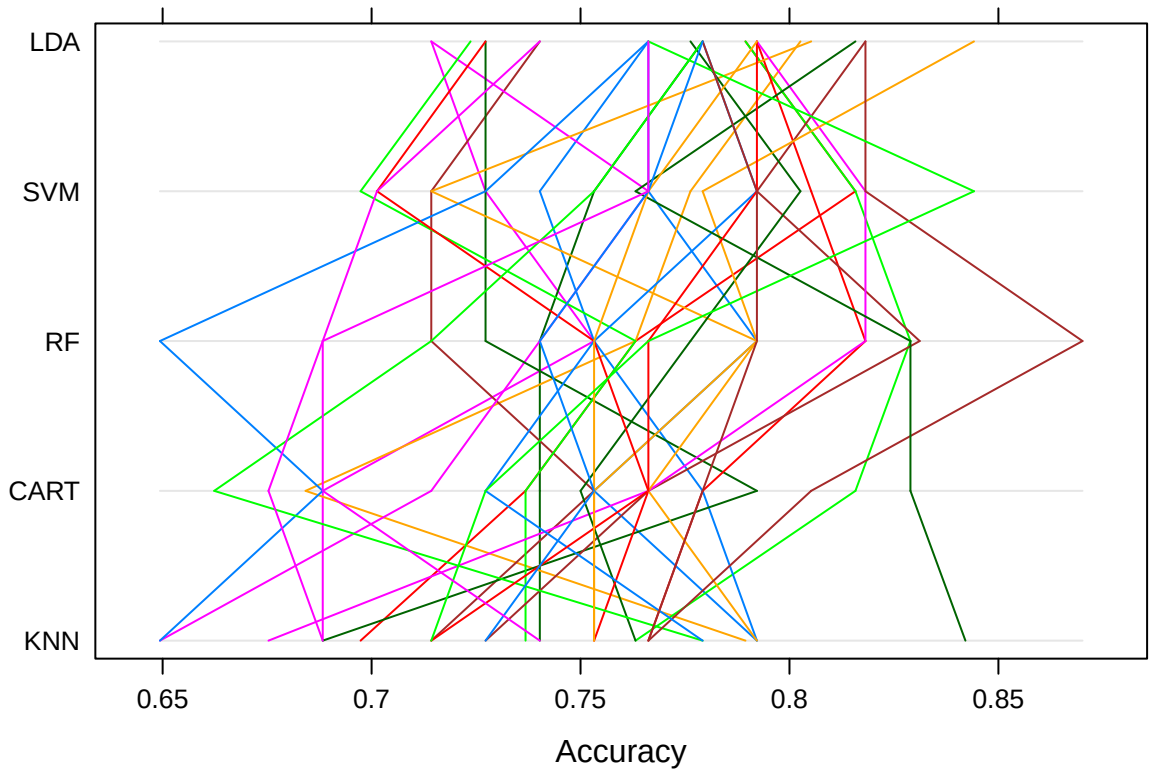

```
# density plots of accuracy
scales <- list(x=list(relation="free"), y=list(relation="free"))
densityplot(results, scales=scales, pch = "|")
```



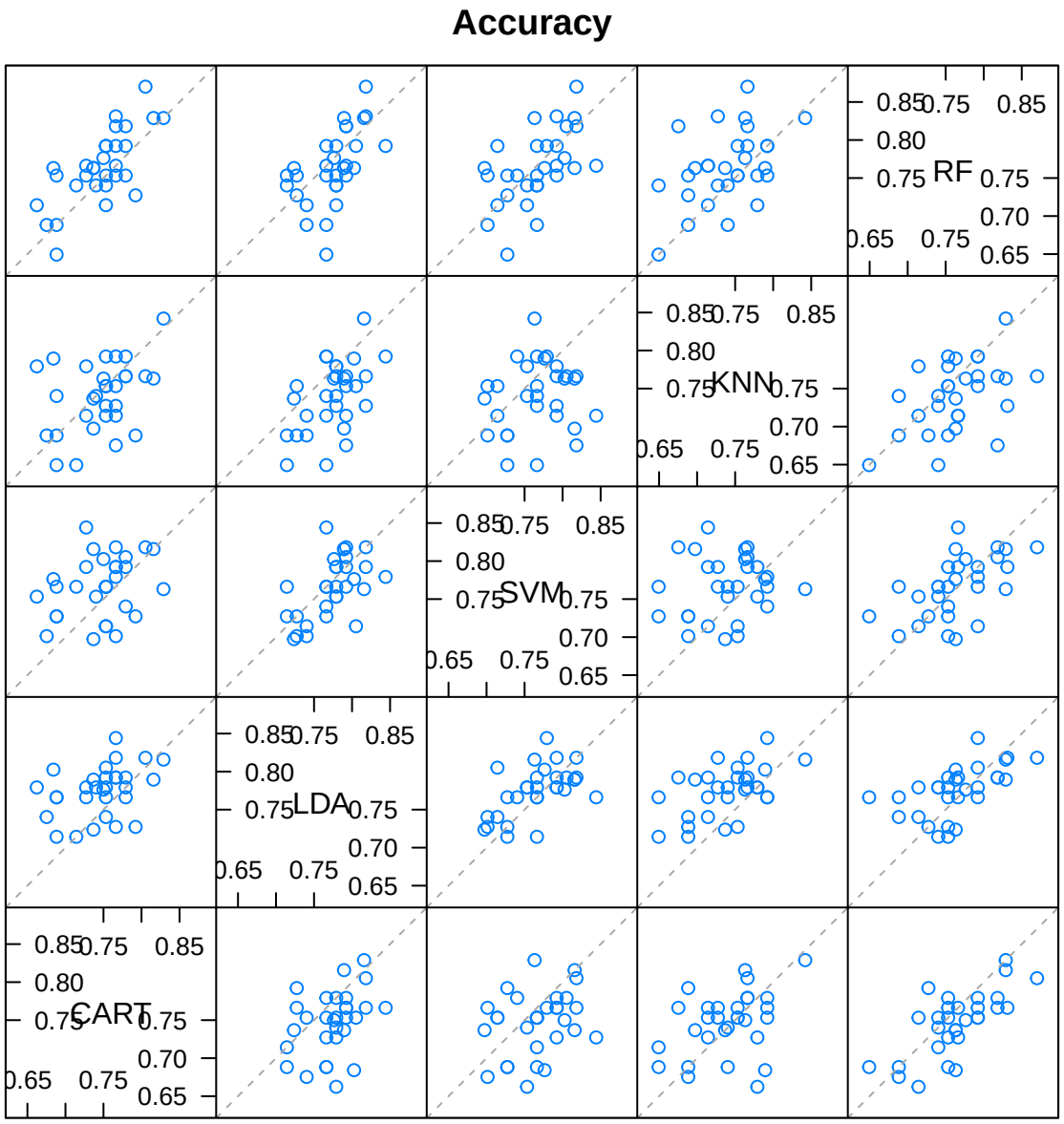

```
# dot plots of accuracy
scales <- list(x=list(relation="free"), y=list(relation="free"))
dotplot(results, scales=scales)
```



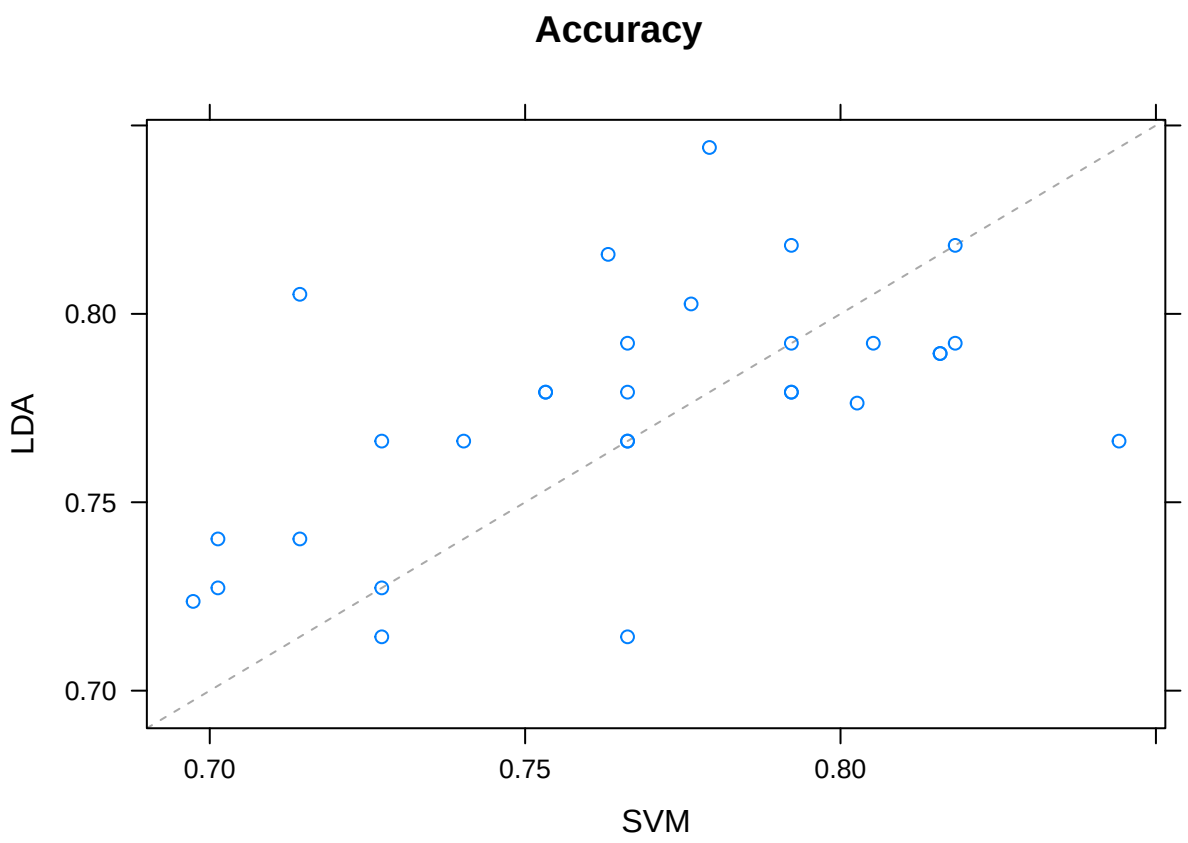
```
# parallel plots to compare models
parallelplot(results)
```




```
# pairwise scatter plots of predictions to compare models
splom(results)
```




```
# xyplot plots to compare models
xyplot(results, models=c("LDA", "SVM"))
```




```
# difference in model predictions
diffs <- diff(results)
# summarize p-values for pairwise comparisons
summary(diffs)
```

```
#>
#> Call:
#> summary.diff.resamples(object = diffs)
#>
#> p-value adjustment: bonferroni
#> Upper diagonal: estimates of the difference
#> Lower diagonal: p-value for H0: difference = 0
#>
#> Accuracy
#>      CART      LDA       SVM       KNN       RF
#> CART           -0.028201 -0.019965  0.006476 -0.017823
#> LDA  0.0109883            0.008237  0.034678  0.010378
#> SVM  0.3270843 1.0000000            0.026441  0.002142
#> KNN  1.0000000 0.0004698 0.1545886           -0.024299
#> RF   0.0770235 1.0000000 1.0000000 0.1030800
#>
#> Kappa
#>      CART     LDA      SVM      KNN      RF
#> CART          -0.05805 -0.03157  0.01374 -0.04654
#> LDA  0.025797           0.02648  0.07179  0.01151
#> SVM  1.000000 0.986395           0.04531 -0.01497
#> KNN  1.000000 0.001399 0.637573          -0.06029
#> RF   0.034983 1.000000 1.000000 0.047566
```

Chapter 17

Temperature modeling using nested dataframes

17.1 Prepare the data

http://ijlyttle.github.io/isugg_purrr/presentation.html#(1)

17.1.1 Packages to run this presentation

```
library("readr")
library("tibble")
library("dplyr")
library("tidyr")
library("stringr")
library("ggplot2")
library("purrr")
library("broom")
```

17.1.2 Motivation

As you know, purrr is a recent package from Hadley Wickham, focused on lists and functional programming, like dplyr is focused on data-frames.

I figure a good way to learn a new package is to try to solve a problem, so we have a dataset:

- you can view or download
- you can download the source of this presentation
- these are three temperatures recorded simultaneously in a piece of electronics
- it will be very valuable to be able to characterize the transient temperature for each sensor
- we want to apply the same set of models across all three sensors
- it will be easier to show using pictures

17.1.3 Let's get the data into shape

Using the readr package

```
temperature_wide <-
  read_csv(file.path(data_raw_dir, "temperature.csv")) %>%
  print()
```

```
# A tibble: 327 x 4
   instant             temperature_a temperature_b temperature_c
   <dttm>                      <dbl>         <dbl>         <dbl>
 1 2015-11-13 06:10:19          116.          91.7          84.2
 2 2015-11-13 06:10:23          116.          91.7          84.2
 3 2015-11-13 06:10:27          116.          91.6          84.2
 4 2015-11-13 06:10:31          116.          91.7          84.2
 5 2015-11-13 06:10:36          116.          91.7          84.2
 6 2015-11-13 06:10:41          116.          91.6          84.2
 7 2015-11-13 06:10:46          116.          91.5          84.2
 8 2015-11-13 06:10:51          116.          91.5          84.2
 9 2015-11-13 06:10:56          116.          91.5          84.2
10 2015-11-13 06:11:01          115.          91.5          84.2
# ... with 317 more rows
```

17.1.4 Is `temperature_wide` "tidy"?


```
# A tibble: 327 x 4
   instant             temperature_a temperature_b temperature_c
   <dttm>                      <dbl>         <dbl>         <dbl>
 1 2015-11-13 06:10:19          116.          91.7          84.2
 2 2015-11-13 06:10:23          116.          91.7          84.2
 3 2015-11-13 06:10:27          116.          91.6          84.2
 4 2015-11-13 06:10:31          116.          91.7          84.2
 5 2015-11-13 06:10:36          116.          91.7          84.2
 6 2015-11-13 06:10:41          116.          91.6          84.2
 7 2015-11-13 06:10:46          116.          91.5          84.2
 8 2015-11-13 06:10:51          116.          91.5          84.2
 9 2015-11-13 06:10:56          116.          91.5          84.2
10 2015-11-13 06:11:01          115.          91.5          84.2
# ... with 317 more rows
```


Why or why not?

17.1.5 Tidy data

1. Each column is a variable
2. Each row is an observation
3. Each cell is a value

(http://www.jstatsoft.org/v59/i10/paper)

My personal observation is that "tidy" can depend on the context, on what you want to do with the data.

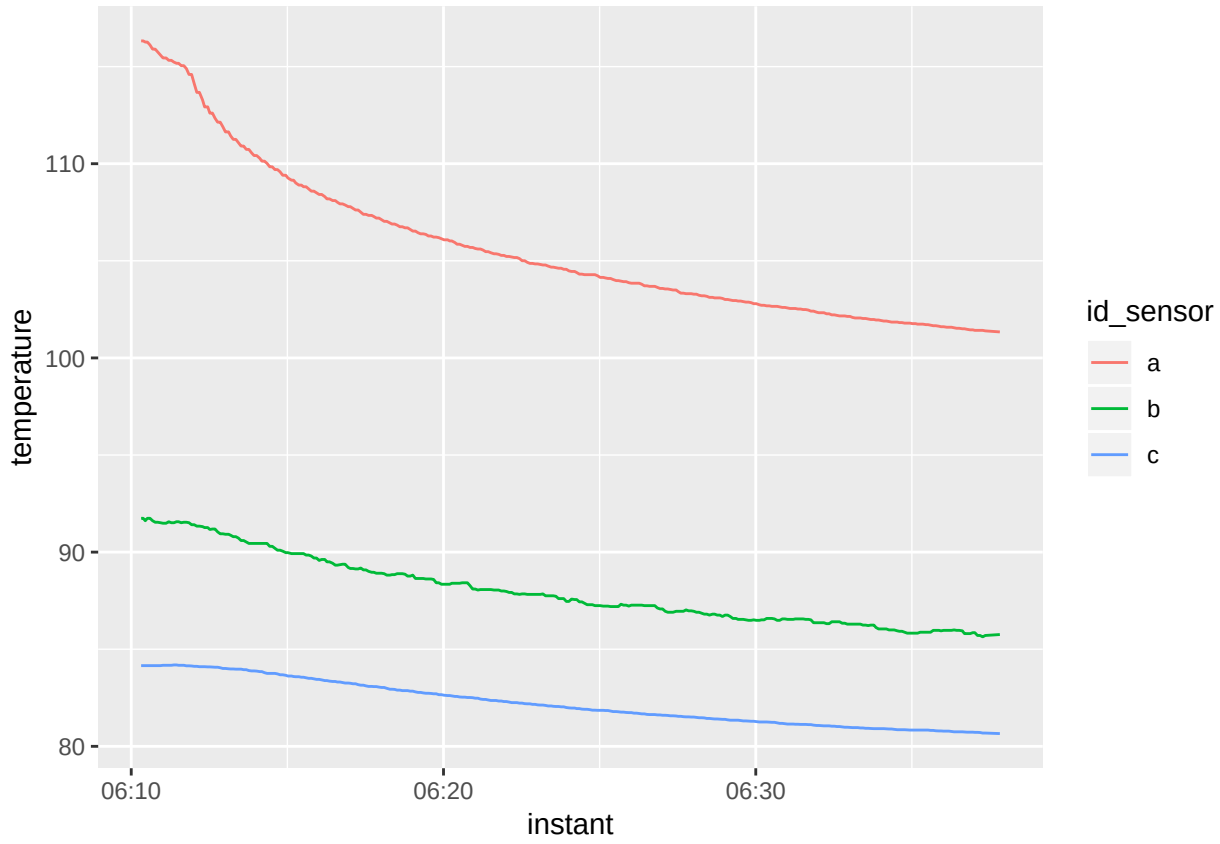
17.1.6 Let's get this into a tidy form

```
temperature_tall <-
  temperature_wide %>%
  gather(key = "id_sensor", value = "temperature", starts_with("temp")) %>%
  mutate(id_sensor = str_replace(id_sensor, "temperature_", "")) %>%
  print()
```

```
# A tibble: 981 x 3
   instant             id_sensor temperature
   <dttm>              <chr>           <dbl>
 1 2015-11-13 06:10:19 a                116.
 2 2015-11-13 06:10:23 a                116.
 3 2015-11-13 06:10:27 a                116.
 4 2015-11-13 06:10:31 a                116.
 5 2015-11-13 06:10:36 a                116.
 6 2015-11-13 06:10:41 a                116.
 7 2015-11-13 06:10:46 a                116.
 8 2015-11-13 06:10:51 a                116.
 9 2015-11-13 06:10:56 a                116.
10 2015-11-13 06:11:01 a                115.
# ... with 971 more rows
```

17.1.7 Now, it's easier to visualize

```
temperature_tall %>%
  ggplot(aes(x = instant, y = temperature, color = id_sensor)) +
  geom_line()
```



17.1.8 Calculate delta time (Δt) and delta temperature (ΔT)

`delta_time` Δt

change in time since event started, s

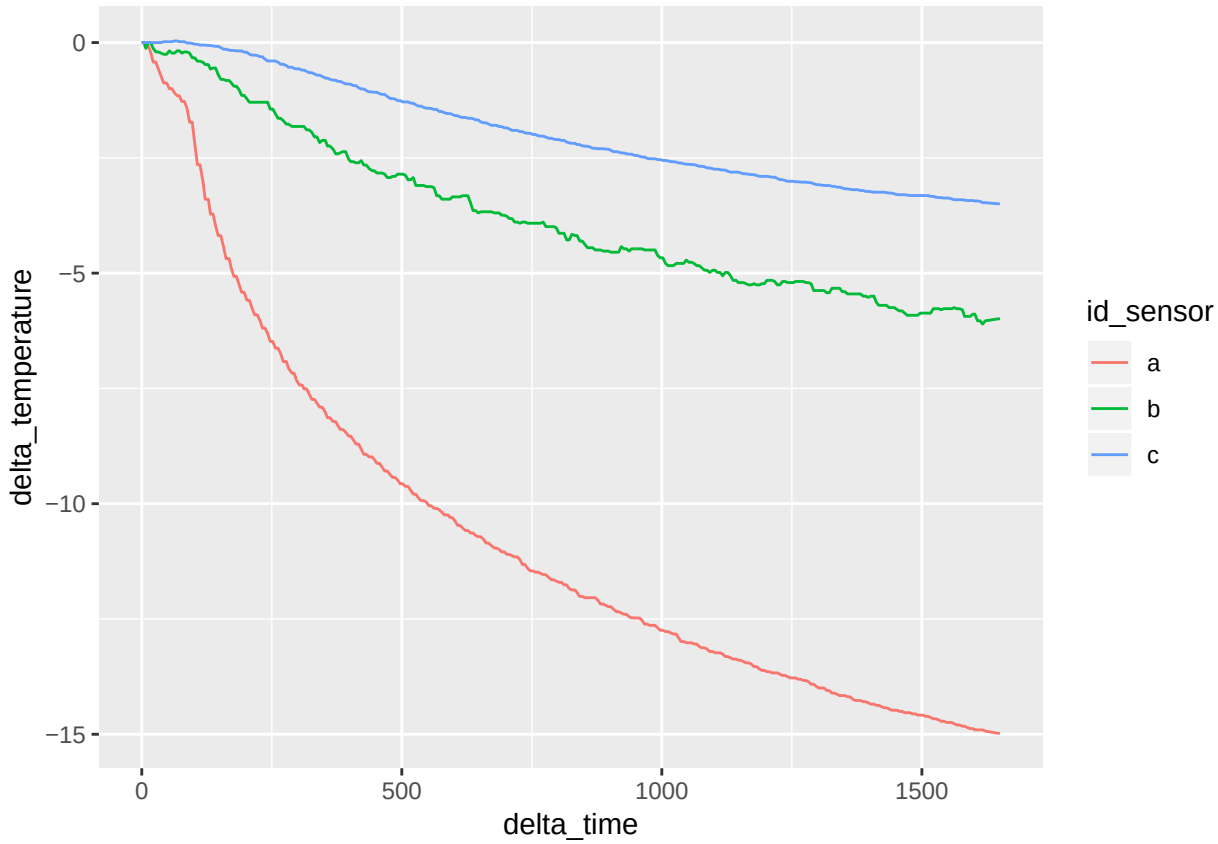
`delta_temperature`: ΔT

change in temperature since event started, °C

```
delta <-
  temperature_tall %>%
  arrange(id_sensor, instant) %>%
  group_by(id_sensor) %>%
  mutate(
    delta_time = as.numeric(instant) - as.numeric(instant[[1]]),
    delta_temperature = temperature - temperature[[1]]
  ) %>%
  select(id_sensor, delta_time, delta_temperature)
```

17.1.9 Let's have a look

```
# plot delta time vs delta temperature, by sensor
delta %>%
  ggplot(aes(x = delta_time, y = delta_temperature, color = id_sensor)) +
  geom_line()
```



17.2 Define the models

We want to see how three different curve-fits might perform on these three data-sets:

17.2.0.1 Newtonian cooling

$$\Delta T = \Delta T_0 * (1 - e^{-\frac{\delta t}{\tau_0}})$$

17.2.1 Semi-infinite solid

$$\Delta T = \Delta T_0 * erfc(\sqrt{\frac{\tau_0}{\delta t}}))$$

17.2.2 Semi-infinite solid with convection

$$\Delta T = \Delta T_0 * \left[\operatorname{erfc}(\sqrt{\frac{\tau_0}{\delta t}}) - e^{Bi_0 + (\frac{Bi_0}{2})^2 \frac{\delta t}{\tau_0}} * \operatorname{erfc}(\sqrt{\frac{\tau_0}{\delta t}} + \frac{Bi_0}{2} * \sqrt{\frac{\delta t}{\tau_0}} \right]$$

17.2.3 `erf` and `erfc` functions

```
# reference: http://stackoverflow.com/questions/29067916/r-error-function-erfz
# (see Abramowitz and Stegun 29.2.29)
erf <- function(x) 2 * pnorm(x * sqrt(2)) - 1
erfc <- function(x) 2 * pnorm(x * sqrt(2), lower = FALSE)
```

17.2.4 Newton cooling equation

```
newton_cooling <- function(x) {
  nls(
    delta_temperature ~ delta_temperature_0 * (1 - exp(-delta_time/tau_0)),
    start = list(delta_temperature_0 = -10, tau_0 = 50),
    data = x
  )
}
```

17.2.5 Temperature models: simple and convection

```
semi_infinite_simple <- function(x) {
  nls(
    delta_temperature ~ delta_temperature_0 * erfc(sqrt(tau_0 / delta_time)),
    start = list(delta_temperature_0 = -10, tau_0 = 50),
    data = x
  )
}

semi_infinite_convection <- function(x){
  nls(
    delta_temperature ~
      delta_temperature_0 * (
        erfc(sqrt(tau_0 / delta_time)) -
        exp(Bi_0 + (Bi_0/2)^2 * delta_time / tau_0) *
          erfc(sqrt(tau_0 / delta_time) +
        (Bi_0/2) * sqrt(delta_time / tau_0))
      ),
    start = list(delta_temperature_0 = -5, tau_0 = 50, Bi_0 = 1.e6),
    data = x
  )
}
```

17.3 Test modeling on one dataset

17.3.1 Before going into purrr

Before doing anything, we want to show that we can do something with one dataset and one model-function:

```
# only one sensor; it is a test
tmp_data <- delta %>% filter(id_sensor == "a")

tmp_model <- newton_cooling(tmp_data)
```

```
summary(tmp_model)
```

```
Formula: delta_temperature ~ delta_temperature_0 * (1 - exp(-delta_time/tau_0))

Parameters:
                     Estimate Std. Error t value Pr(>|t|)
delta_temperature_0 -15.06085    0.05262  -286.2   <2e-16 ***
tau_0               500.01382    4.83673   103.4   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3267 on 325 degrees of freedom

Number of iterations to convergence: 7
Achieved convergence tolerance: 4.136e-06
```

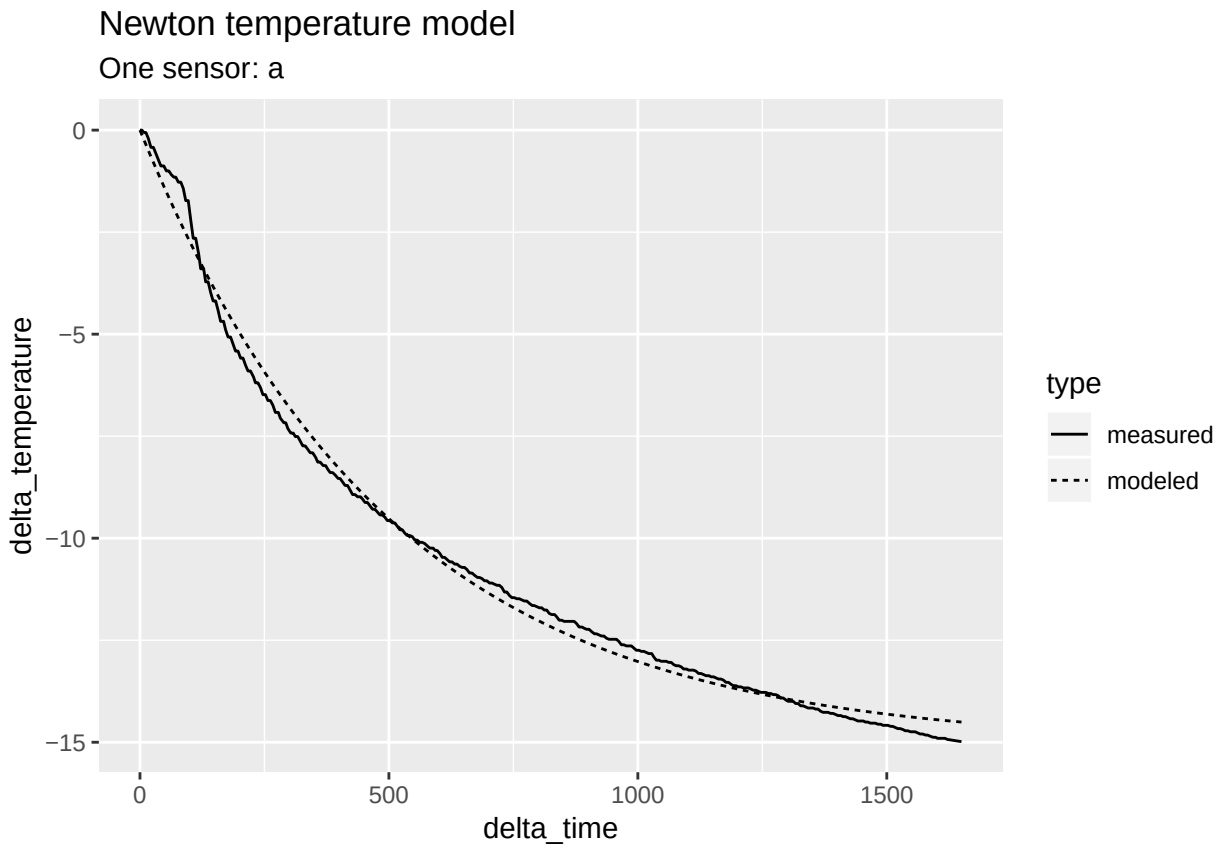
17.3.2 Look at predictions

```
# apply prediction and make it tidy
tmp_pred <-
  tmp_data %>%
  mutate(modeled = predict(tmp_model, data = .)) %>%
  select(id_sensor, delta_time, measured = delta_temperature, modeled) %>%
  gather("type", "delta_temperature", measured:modeled) %>%
  print()
```

```
# A tibble: 654 x 4
# Groups:   id_sensor [1]
   id_sensor delta_time type     delta_temperature
   <chr>          <dbl> <chr>                <dbl>
 1 a                  0 measured             0
 2 a                  4 measured             0
 3 a                  8 measured            -0.06
 4 a                 12 measured            -0.06
 5 a                 17 measured            -0.211
 6 a                 22 measured            -0.423
 7 a                 27 measured            -0.423
 8 a                 32 measured            -0.574
 9 a                 37 measured            -0.726
10 a                 42 measured            -0.878
# ... with 644 more rows
```

17.3.3 Plot Newton model

```
tmp_pred %>%
  ggplot(aes(x = delta_time, y = delta_temperature, linetype = type)) +
  geom_line() +
  labs(title = "Newton temperature model", subtitle = "One sensor: a")
```



17.3.4 "Regular" data-frame (deltas)

```
print(delta)
```

```
# A tibble: 981 x 3
# Groups:   id_sensor [3]
   id_sensor delta_time delta_temperature
   <chr>          <dbl>             <dbl>
 1 a                  0              0
 2 a                  4              0
 3 a                  8             -0.06
 4 a                 12             -0.06
 5 a                 17             -0.211
 6 a                 22             -0.423
 7 a                 27             -0.423
 8 a                 32             -0.574
 9 a                 37             -0.726
10 a                 42             -0.878
# ... with 971 more rows
```

Each column of the dataframe is a vector - in this case, a character vector and two doubles

17.4 Making a nested dataframe

17.4.1 How to make a weird data-frame

Here's where the fun starts - a column of a data-frame can be a list.

- use `tidyr::nest()` to makes a column `data`, which is a list of data-frames
- this seems like a stronger expression of the `dplyr::group_by()` idea

```
# nest delta_time and delta_temperature variables
delta_nested <-
  delta %>%
  nest(-id_sensor) %>%
  print()
```

```
# A tibble: 3 x 2
  id_sensor data
  <chr>     <list>
1 a         <tibble [327 x 2]>
2 b         <tibble [327 x 2]>
3 c         <tibble [327 x 2]>
```

17.4.2 Map dataframes to a modeling function (Newton)

- `map()` is like `lapply()`
- `map()` returns a list-column (it keeps the weirdness)

```
model_nested <-
  delta_nested %>%
  mutate(model = map(data, newton_cooling)) %>%
  print()
```

```
# A tibble: 3 x 3
  id_sensor data               model
  <chr>     <list>             <list>
1 a         <tibble [327 x 2]> <nls>
2 b         <tibble [327 x 2]> <nls>
3 c         <tibble [327 x 2]> <nls>
```

We get an additional list-column `model`.

17.4.3 We can use `map2()` to make the predictions

- `map2()` is like `mapply()`
- designed to map two colunms (`model`, `data`) to a function `predict()`

```
predict_nested <-
  model_nested %>%
  mutate(pred = map2(model, data, predict)) %>%
  print()
```

```
# A tibble: 3 x 4
  id_sensor data               model  pred
  <chr>     <list>             <list> <list>
```

```
1 a         <tibble [327 x 2]> <nls>  <dbl [327]>
2 b         <tibble [327 x 2]> <nls>  <dbl [327]>
3 c         <tibble [327 x 2]> <nls>  <dbl [327]>
```

Another list-column `pred` for the prediction results.

17.4.4 We need to get out of the weirdness

- use `unnest()` to get back to a regular data-frame

```
predict_unnested <-
  predict_nested %>%
  unnest(data, pred) %>%
  print()
```

```
# A tibble: 981 x 4
   id_sensor   pred delta_time delta_temperature
   <chr>      <dbl>      <dbl>             <dbl>
 1 a          0              0             0
 2 a         -0.120          4             0
 3 a         -0.239          8            -0.06
 4 a         -0.357         12            -0.06
 5 a         -0.503         17            -0.211
 6 a         -0.648         22            -0.423
 7 a         -0.792         27            -0.423
 8 a         -0.934         32            -0.574
 9 a         -1.07          37            -0.726
10 a         -1.21          42            -0.878
# ... with 971 more rows
```

17.4.5 We can wrangle the predictions

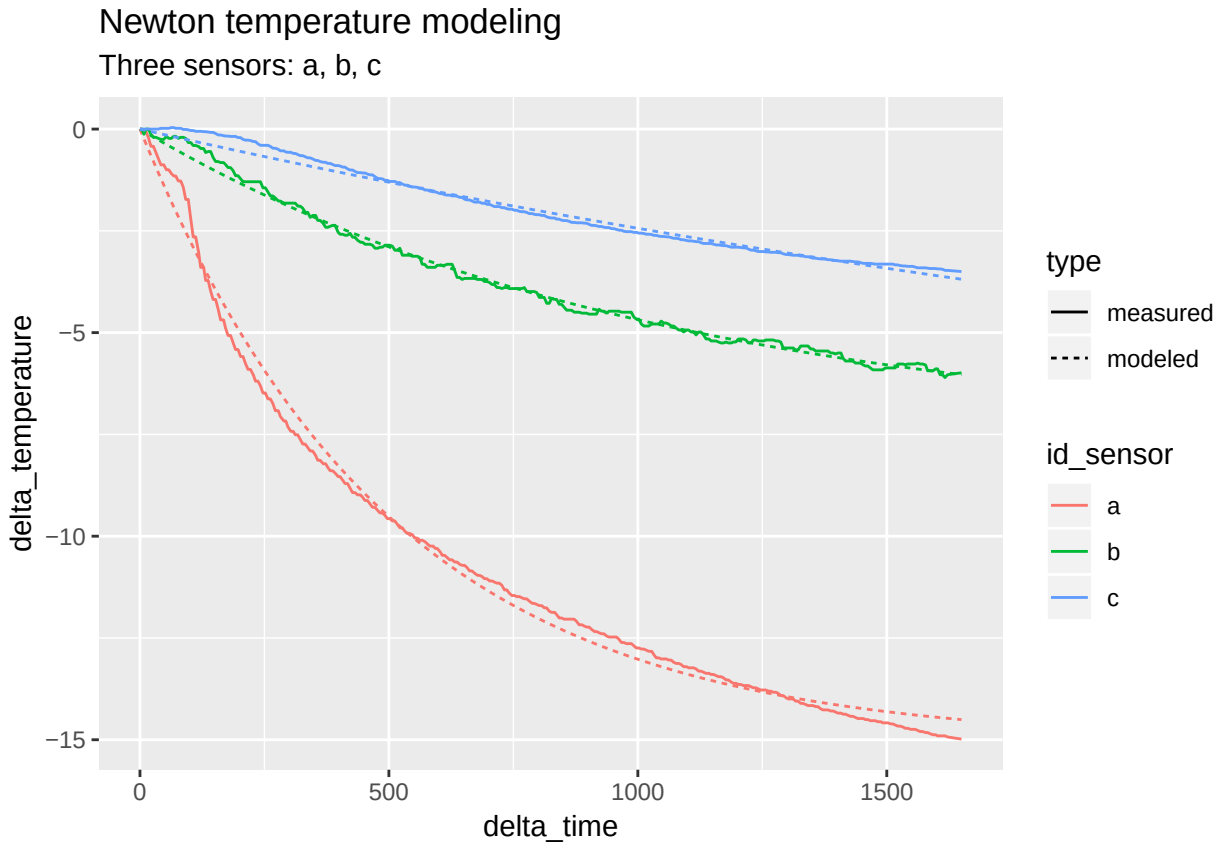
- get into a form that makes it easier to plot

```
predict_tall <-
  predict_unnested %>%
  rename(modeled = pred, measured = delta_temperature) %>%
  gather("type", "delta_temperature", modeled, measured) %>%
  print()
```

```
# A tibble: 1,962 x 4
   id_sensor delta_time type    delta_temperature
   <chr>          <dbl> <chr>               <dbl>
 1 a                  0 modeled             0
 2 a                  4 modeled            -0.120
 3 a                  8 modeled            -0.239
 4 a                 12 modeled            -0.357
 5 a                 17 modeled            -0.503
 6 a                 22 modeled            -0.648
 7 a                 27 modeled            -0.792
 8 a                 32 modeled            -0.934
 9 a                 37 modeled            -1.07
10 a                 42 modeled            -1.21
# ... with 1,952 more rows
```

17.4.6 We can visualize the predictions

```
predict_tall %>%
  ggplot(aes(x = delta_time, y = delta_temperature)) +
  geom_line(aes(color = id_sensor, linetype = type)) +
  labs(title = "Newton temperature modeling",
       subtitle = "Three sensors: a, b, c")
```



17.5 Apply multiple models on a nested structure

17.5.1 Step 1: Selection of models

Make a list of functions to model:

```
list_model <-
  list(
    newton_cooling = newton_cooling,
    semi_infinite_simple = semi_infinite_simple,
    semi_infinite_convection = semi_infinite_convection
  )
```

17.5.2 Step 2: write a function to define the "inner" loop

```
# add additional variable with the model name

fn_model <- function(.model, df) {
  # one parameter for the model in the list, the second for the data
  # safer to avoid non-standard evaluation
  # df %>% mutate(model = map(data, .model))

  df$model <- map(df$data, possibly(.model, NULL))
  df
}
```

- for a given model-function and a given (weird) data-frame, return a modified version of that data-frame with a column `model`, which is the model-function applied to each element of the data-frame's `data` column (which is itself a list of data-frames)
- the purrr functions `safely()` and `possibly()` are **very** interesting. I think they could be useful outside of purrr as a friendlier way to do error-handling.

17.5.3 Step 3: Use `map_df()` to define the "outer" loop

```
# this dataframe will be the second input of fn_model
delta_nested %>%
  print()
```

```
# A tibble: 3 x 2
  id_sensor data
  <chr>     <list>
1 a         <tibble [327 x 2]>
2 b         <tibble [327 x 2]>
3 c         <tibble [327 x 2]>
```

```
# fn_model is receiving two inputs: one from list_model and from delta_nested
model_nested_new <-
  list_model %>%
  map_df(fn_model, delta_nested, .id = "id_model") %>%
  print()
```

```
# A tibble: 9 x 4
  id_model                id_sensor data               model
  <chr>                   <chr>     <list>             <list>
1 newton_cooling          a         <tibble [327 x 2]> <nls>
2 newton_cooling          b         <tibble [327 x 2]> <nls>
3 newton_cooling          c         <tibble [327 x 2]> <nls>
4 semi_infinite_simple    a         <tibble [327 x 2]> <nls>
5 semi_infinite_simple    b         <tibble [327 x 2]> <nls>
6 semi_infinite_simple    c         <tibble [327 x 2]> <nls>
7 semi_infinite_convection a        <tibble [327 x 2]> <NULL>
8 semi_infinite_convection b        <tibble [327 x 2]> <NULL>
9 semi_infinite_convection c        <tibble [327 x 2]> <NULL>
```

- for each element of a list of model-functions, run the inner-loop function, and row-bind the results into a data-frame

- we want to discard the rows where the model failed
- we also want to investigate why they failed, but that's a different talk

17.5.4 Step 4: Use `map()` to identify the null models

```
model_nested_new <-
  list_model %>%
  map_df(fn_model, delta_nested, .id = "id_model") %>%
  mutate(is_null = map(model, is.null)) %>%
  print()
```

```
# A tibble: 9 x 5
  id_model                id_sensor data               model  is_null
  <chr>                   <chr>     <list>             <list> <list>
1 newton_cooling          a         <tibble [327 x 2]> <nls>  <lgl [1]>
2 newton_cooling          b         <tibble [327 x 2]> <nls>  <lgl [1]>
3 newton_cooling          c         <tibble [327 x 2]> <nls>  <lgl [1]>
4 semi_infinite_simple    a         <tibble [327 x 2]> <nls>  <lgl [1]>
5 semi_infinite_simple    b         <tibble [327 x 2]> <nls>  <lgl [1]>
6 semi_infinite_simple    c         <tibble [327 x 2]> <nls>  <lgl [1]>
7 semi_infinite_convection a        <tibble [327 x 2]> <NULL> <lgl [1]>
8 semi_infinite_convection b        <tibble [327 x 2]> <NULL> <lgl [1]>
9 semi_infinite_convection c        <tibble [327 x 2]> <NULL> <lgl [1]>
```

- using `map(model, is.null)` returns a list column
- to use `filter()`, we have to escape the weirdness

17.5.5 Step 5: `map_lgl()` to identify nulls and get out of the weirdness

```
model_nested_new <-
  list_model %>%
  map_df(fn_model, delta_nested, .id = "id_model") %>%
  mutate(is_null = map_lgl(model, is.null)) %>%
  print()
```

```
# A tibble: 9 x 5
  id_model                id_sensor data               model  is_null
  <chr>                   <chr>     <list>             <list> <lgl>
1 newton_cooling          a         <tibble [327 x 2]> <nls>  FALSE
2 newton_cooling          b         <tibble [327 x 2]> <nls>  FALSE
3 newton_cooling          c         <tibble [327 x 2]> <nls>  FALSE
4 semi_infinite_simple    a         <tibble [327 x 2]> <nls>  FALSE
5 semi_infinite_simple    b         <tibble [327 x 2]> <nls>  FALSE
6 semi_infinite_simple    c         <tibble [327 x 2]> <nls>  FALSE
7 semi_infinite_convection a        <tibble [327 x 2]> <NULL> TRUE
8 semi_infinite_convection b        <tibble [327 x 2]> <NULL> TRUE
9 semi_infinite_convection c        <tibble [327 x 2]> <NULL> TRUE
```

- using `map_lgl(model, is.null)` returns a vector column

17.5.6 Step 6: `filter()` nulls and `select()` variables to clean up

```
model_nested_new <-
  list_model %>%
  map_df(fn_model, delta_nested, .id = "id_model") %>%
  mutate(is_null = map_lgl(model, is.null)) %>%
  filter(!is_null) %>%
  select(-is_null) %>%
  print()
```

```
# A tibble: 6 x 4
  id_model             id_sensor data              model
  <chr>                <chr>     <list>            <list>
1 newton_cooling       a         <tibble [327 x 2]> <nls>
2 newton_cooling       b         <tibble [327 x 2]> <nls>
3 newton_cooling       c         <tibble [327 x 2]> <nls>
4 semi_infinite_simple a         <tibble [327 x 2]> <nls>
5 semi_infinite_simple b         <tibble [327 x 2]> <nls>
6 semi_infinite_simple c         <tibble [327 x 2]> <nls>
```

17.5.7 Step 7: Calculate predictions on nested dataframe

```
predict_nested <-
  model_nested_new %>%
  mutate(pred = map2(model, data, predict)) %>%
  print()
```

```
# A tibble: 6 x 5
  id_model             id_sensor data              model  pred
  <chr>                <chr>     <list>            <list> <list>
1 newton_cooling       a         <tibble [327 x 2]> <nls>  <dbl [327]>
2 newton_cooling       b         <tibble [327 x 2]> <nls>  <dbl [327]>
3 newton_cooling       c         <tibble [327 x 2]> <nls>  <dbl [327]>
4 semi_infinite_simple a         <tibble [327 x 2]> <nls>  <dbl [327]>
5 semi_infinite_simple b         <tibble [327 x 2]> <nls>  <dbl [327]>
6 semi_infinite_simple c         <tibble [327 x 2]> <nls>  <dbl [327]>
```

17.5.8 `unnest()`, make it tall and tidy

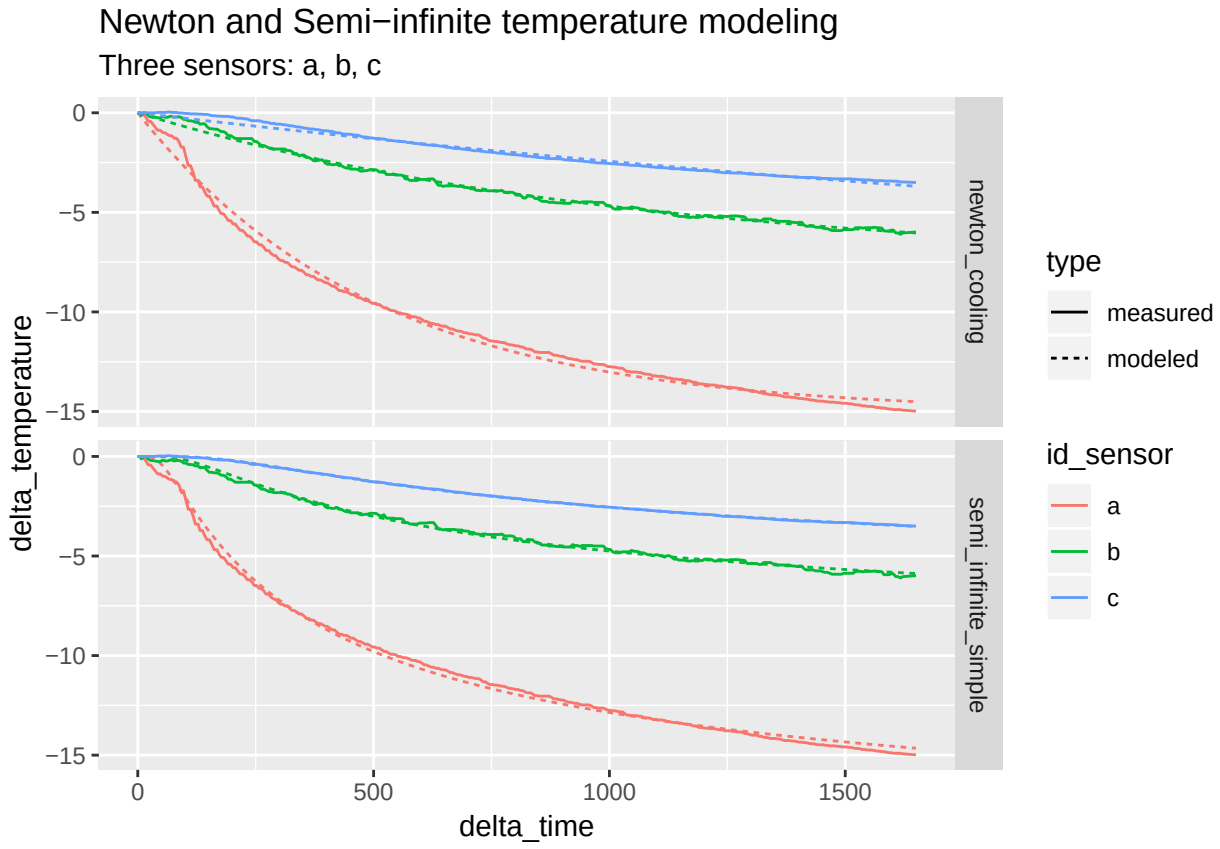
```
predict_tall <-
  predict_nested %>%
  unnest(data, pred) %>%
  rename(modeled = pred, measured = delta_temperature) %>%
  gather("type", "delta_temperature", modeled, measured) %>%
  print()
```

```
# A tibble: 3,924 x 5
   id_model       id_sensor delta_time type    delta_temperature
   <chr>          <chr>          <dbl> <chr>               <dbl>
 1 newton_cooling a                  0 modeled             0
 2 newton_cooling a                  4 modeled            -0.120
 3 newton_cooling a                  8 modeled            -0.239
```

```
 4 newton_cooling a                12 modeled            -0.357
 5 newton_cooling a                17 modeled            -0.503
 6 newton_cooling a                22 modeled            -0.648
 7 newton_cooling a                27 modeled            -0.792
 8 newton_cooling a                32 modeled            -0.934
 9 newton_cooling a                37 modeled            -1.07
10 newton_cooling a                42 modeled            -1.21
# ... with 3,914 more rows
```

17.5.9 Visualize the predictions

```
predict_tall %>%
  ggplot(aes(x = delta_time, y = delta_temperature)) +
  geom_line(aes(color = id_sensor, linetype = type)) +
  facet_grid(id_model ~ .) +
  labs(title = "Newton and Semi-infinite temperature modeling",
       subtitle = "Three sensors: a, b, c")
```



17.5.10 Let's get the residuals

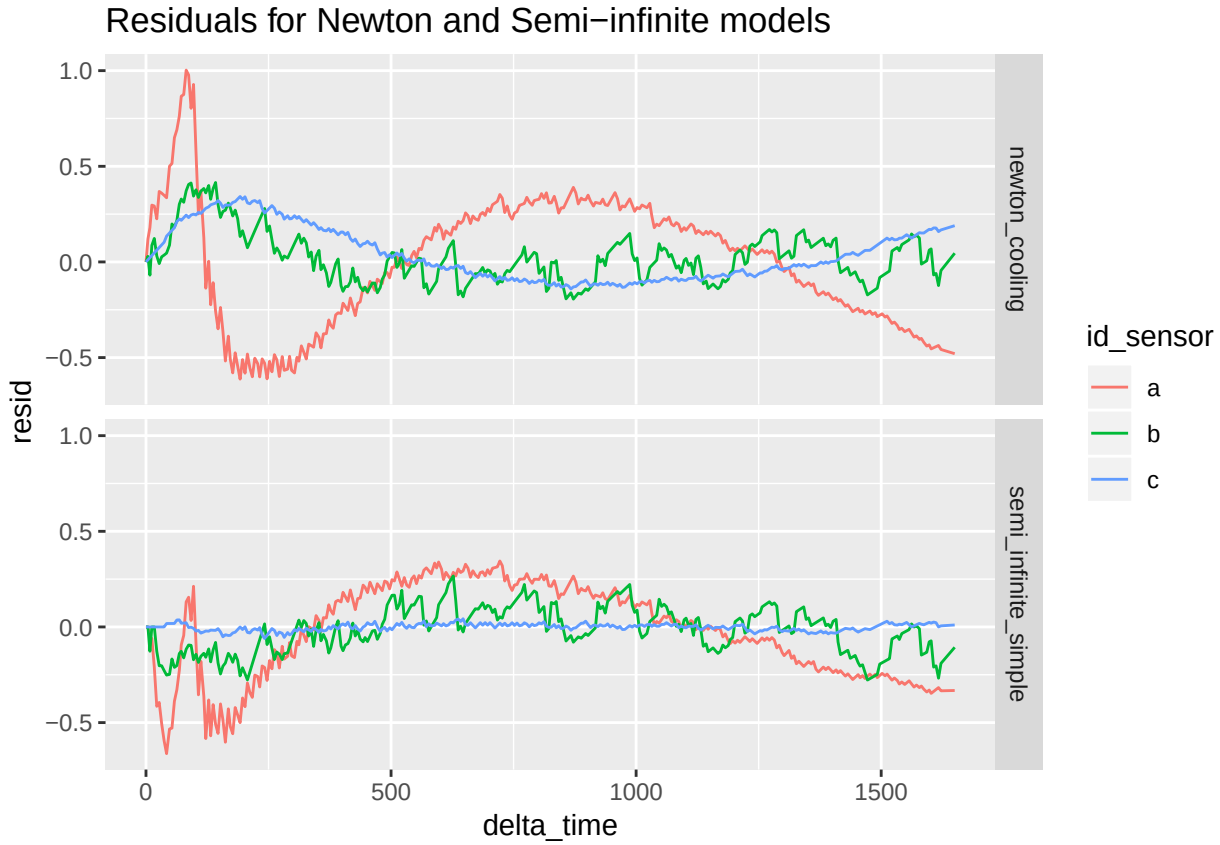
```
resid <-
  model_nested_new %>%
  mutate(resid = map(model, resid)) %>%
```

```
  unnest(data, resid) %>%
  print()
```

```
# A tibble: 1,962 x 5
   id_model       id_sensor resid delta_time delta_temperature
   <chr>          <chr>     <dbl>      <dbl>             <dbl>
 1 newton_cooling a         0              0              0
 2 newton_cooling a         0.120          4              0
 3 newton_cooling a         0.179          8             -0.06
 4 newton_cooling a         0.297         12             -0.06
 5 newton_cooling a         0.292         17             -0.211
 6 newton_cooling a         0.225         22             -0.423
 7 newton_cooling a         0.369         27             -0.423
 8 newton_cooling a         0.360         32             -0.574
 9 newton_cooling a         0.348         37             -0.726
10 newton_cooling a         0.335         42             -0.878
# ... with 1,952 more rows
```

17.5.11 And visualize them

```
resid %>%
  ggplot(aes(x = delta_time, y = resid)) +
  geom_line(aes(color = id_sensor)) +
  facet_grid(id_model ~ .) +
  labs(title = "Residuals for Newton and Semi-infinite models")
```



17.6 Using broom package to look at model-statistics

We will use a previous defined dataframe with the model and data:

```
model_nested_new %>%
  print()
```

```
# A tibble: 6 x 4
  id_model             id_sensor data               model
  <chr>                <chr>     <list>             <list>
1 newton_cooling       a         <tibble [327 x 2]> <nls>
2 newton_cooling       b         <tibble [327 x 2]> <nls>
3 newton_cooling       c         <tibble [327 x 2]> <nls>
4 semi_infinite_simple a         <tibble [327 x 2]> <nls>
5 semi_infinite_simple b         <tibble [327 x 2]> <nls>
6 semi_infinite_simple c         <tibble [327 x 2]> <nls>
```

The `tidy()` function extracts statistics from a model.

```
# apply over model_nested_new but only three variables
model_parameters <-
  model_nested_new %>%
  select(id_model, id_sensor, model) %>%
  mutate(tidy = map(model, tidy)) %>%
  select(-model) %>%
  unnest() %>%
  print()
```

```
# A tibble: 12 x 7
   id_model      id_sensor term      estimate std.error statistic   p.value
   <chr>         <chr>     <chr>        <dbl>     <dbl>     <dbl>     <dbl>
 1 newton_cooli~ a         delta_te~   -15.1     0.0526    -286.  0.
 2 newton_cooli~ a         tau_0       500.      4.84       103.  1.07e-250
 3 newton_cooli~ b         delta_te~    -7.59    0.0676    -112.  6.38e-262
 4 newton_cooli~ b         tau_0      1041.     16.2         64.2 9.05e-187
 5 newton_cooli~ c         delta_te~    -9.87    0.704      -14.0 3.16e- 35
 6 newton_cooli~ c         tau_0      3525.    299.          11.8 5.61e- 27
 7 semi_infinit~ a         delta_te~   -21.5     0.0649    -332.  0.
 8 semi_infinit~ a         tau_0       139.      1.15       121.  2.14e-272
 9 semi_infinit~ b         delta_te~   -10.6     0.0515    -206.  0.
10 semi_infinit~ b         tau_0       287.      2.58       111.  1.46e-260
11 semi_infinit~ c         delta_te~    -8.04    0.0129    -626.  0.
12 semi_infinit~ c         tau_0       500.      1.07       468.  0.
```

17.6.1 Get a sense of the coefficients

```
model_summary <-
  model_parameters %>%
  select(id_model, id_sensor, term, estimate) %>%
  spread(key = "term", value = "estimate") %>%
  print()
```

```
# A tibble: 6 x 4
  id_model             id_sensor delta_temperature_0 tau_0
  <chr>                <chr>                   <dbl> <dbl>
```

```
1 newton_cooling       a                    -15.1   500.
2 newton_cooling       b                     -7.59 1041.
3 newton_cooling       c                     -9.87 3525.
4 semi_infinite_simple a                    -21.5   139.
5 semi_infinite_simple b                    -10.6   287.
6 semi_infinite_simple c                     -8.04  500.
```

17.6.2 Summary

- this is just a smalll part of purrr
- there seem to be parallels between `tidyr::nest()/purrr::map()` and `dplyr::group_by()/dplyr::do()`
  - to my mind, the purrr framework is more understandable
  - update tweet from Hadley

References from Hadley:

- purrr 0.1.0 announcement
- purrr 0.2.0 announcement
- chapter from Garrett Grolemund and Hadley's forthcoming book

Chapter 18

Standalone Model

Classification problem

- `mtry`: Number of variables randomly sampled as candidates at each split.
- `ntree`: Number of trees to grow.

18.1 Load libraries

```
# load packages
library(caret)
library(mlbench)
library(randomForest)

# load dataset
data(Sonar)
set.seed(7)
```

18.2 Explore data

```
dplyr::glimpse(Sonar)
```

```
#> Observations: 208
#> Variables: 61
#> $ V1    <dbl> 0.0200, 0.0453, 0.0262, 0.0100, 0.0762, 0.0286, 0.0317, ...
#> $ V2    <dbl> 0.0371, 0.0523, 0.0582, 0.0171, 0.0666, 0.0453, 0.0956, ...
#> $ V3    <dbl> 0.0428, 0.0843, 0.1099, 0.0623, 0.0481, 0.0277, 0.1321, ...
#> $ V4    <dbl> 0.0207, 0.0689, 0.1083, 0.0205, 0.0394, 0.0174, 0.1408, ...
#> $ V5    <dbl> 0.0954, 0.1183, 0.0974, 0.0205, 0.0590, 0.0384, 0.1674, ...
#> $ V6    <dbl> 0.0986, 0.2583, 0.2280, 0.0368, 0.0649, 0.0990, 0.1710, ...
#> $ V7    <dbl> 0.1539, 0.2156, 0.2431, 0.1098, 0.1209, 0.1201, 0.0731, ...
#> $ V8    <dbl> 0.1601, 0.3481, 0.3771, 0.1276, 0.2467, 0.1833, 0.1401, ...
#> $ V9    <dbl> 0.3109, 0.3337, 0.5598, 0.0598, 0.3564, 0.2105, 0.2083, ...
#> $ V10   <dbl> 0.2111, 0.2872, 0.6194, 0.1264, 0.4459, 0.3039, 0.3513, ...
#> $ V11   <dbl> 0.1609, 0.4918, 0.6333, 0.0881, 0.4152, 0.2988, 0.1786, ...
#> $ V12   <dbl> 0.1582, 0.6552, 0.7060, 0.1992, 0.3952, 0.4250, 0.0658, ...
```

```
#> $ V13   <dbl> 0.2238, 0.6919, 0.5544, 0.0184, 0.4256, 0.6343, 0.0513, ...
#> $ V14   <dbl> 0.0645, 0.7797, 0.5320, 0.2261, 0.4135, 0.8198, 0.3752, ...
#> $ V15   <dbl> 0.0660, 0.7464, 0.6479, 0.1729, 0.4528, 1.0000, 0.5419, ...
#> $ V16   <dbl> 0.2273, 0.9444, 0.6931, 0.2131, 0.5326, 0.9988, 0.5440, ...
#> $ V17   <dbl> 0.3100, 1.0000, 0.6759, 0.0693, 0.7306, 0.9508, 0.5150, ...
#> $ V18   <dbl> 0.2999, 0.8874, 0.7551, 0.2281, 0.6193, 0.9025, 0.4262, ...
#> $ V19   <dbl> 0.5078, 0.8024, 0.8929, 0.4060, 0.2032, 0.7234, 0.2024, ...
#> $ V20   <dbl> 0.4797, 0.7818, 0.8619, 0.3973, 0.4636, 0.5122, 0.4233, ...
#> $ V21   <dbl> 0.5783, 0.5212, 0.7974, 0.2741, 0.4148, 0.2074, 0.7723, ...
#> $ V22   <dbl> 0.5071, 0.4052, 0.6737, 0.3690, 0.4292, 0.3985, 0.9735, ...
#> $ V23   <dbl> 0.4328, 0.3957, 0.4293, 0.5556, 0.5730, 0.5890, 0.9390, ...
#> $ V24   <dbl> 0.5550, 0.3914, 0.3648, 0.4846, 0.5399, 0.2872, 0.5559, ...
#> $ V25   <dbl> 0.6711, 0.3250, 0.5331, 0.3140, 0.3161, 0.2043, 0.5268, ...
#> $ V26   <dbl> 0.6415, 0.3200, 0.2413, 0.5334, 0.2285, 0.5782, 0.6826, ...
#> $ V27   <dbl> 0.7104, 0.3271, 0.5070, 0.5256, 0.6995, 0.5389, 0.5713, ...
#> $ V28   <dbl> 0.8080, 0.2767, 0.8533, 0.2520, 1.0000, 0.3750, 0.5429, ...
#> $ V29   <dbl> 0.6791, 0.4423, 0.6036, 0.2090, 0.7262, 0.3411, 0.2177, ...
#> $ V30   <dbl> 0.3857, 0.2028, 0.8514, 0.3559, 0.4724, 0.5067, 0.2149, ...
#> $ V31   <dbl> 0.1307, 0.3788, 0.8512, 0.6260, 0.5103, 0.5580, 0.5811, ...
#> $ V32   <dbl> 0.2604, 0.2947, 0.5045, 0.7340, 0.5459, 0.4778, 0.6323, ...
#> $ V33   <dbl> 0.5121, 0.1984, 0.1862, 0.6120, 0.2881, 0.3299, 0.2965, ...
#> $ V34   <dbl> 0.7547, 0.2341, 0.2709, 0.3497, 0.0981, 0.2198, 0.1873, ...
#> $ V35   <dbl> 0.8537, 0.1306, 0.4232, 0.3953, 0.1951, 0.1407, 0.2969, ...
#> $ V36   <dbl> 0.8507, 0.4182, 0.3043, 0.3012, 0.4181, 0.2856, 0.5163, ...
#> $ V37   <dbl> 0.6692, 0.3835, 0.6116, 0.5408, 0.4604, 0.3807, 0.6153, ...
#> $ V38   <dbl> 0.6097, 0.1057, 0.6756, 0.8814, 0.3217, 0.4158, 0.4283, ...
#> $ V39   <dbl> 0.4943, 0.1840, 0.5375, 0.9857, 0.2828, 0.4054, 0.5479, ...
#> $ V40   <dbl> 0.2744, 0.1970, 0.4719, 0.9167, 0.2430, 0.3296, 0.6133, ...
#> $ V41   <dbl> 0.0510, 0.1674, 0.4647, 0.6121, 0.1979, 0.2707, 0.5017, ...
#> $ V42   <dbl> 0.2834, 0.0583, 0.2587, 0.5006, 0.2444, 0.2650, 0.2377, ...
#> $ V43   <dbl> 0.2825, 0.1401, 0.2129, 0.3210, 0.1847, 0.0723, 0.1957, ...
#> $ V44   <dbl> 0.4256, 0.1628, 0.2222, 0.3202, 0.0841, 0.1238, 0.1749, ...
#> $ V45   <dbl> 0.2641, 0.0621, 0.2111, 0.4295, 0.0692, 0.1192, 0.1304, ...
#> $ V46   <dbl> 0.1386, 0.0203, 0.0176, 0.3654, 0.0528, 0.1089, 0.0597, ...
#> $ V47   <dbl> 0.1051, 0.0530, 0.1348, 0.2655, 0.0357, 0.0623, 0.1124, ...
#> $ V48   <dbl> 0.1343, 0.0742, 0.0744, 0.1576, 0.0085, 0.0494, 0.1047, ...
#> $ V49   <dbl> 0.0383, 0.0409, 0.0130, 0.0681, 0.0230, 0.0264, 0.0507, ...
#> $ V50   <dbl> 0.0324, 0.0061, 0.0106, 0.0294, 0.0046, 0.0081, 0.0159, ...
#> $ V51   <dbl> 0.0232, 0.0125, 0.0033, 0.0241, 0.0156, 0.0104, 0.0195, ...
#> $ V52   <dbl> 0.0027, 0.0084, 0.0232, 0.0121, 0.0031, 0.0045, 0.0201, ...
#> $ V53   <dbl> 0.0065, 0.0089, 0.0166, 0.0036, 0.0054, 0.0014, 0.0248, ...
#> $ V54   <dbl> 0.0159, 0.0048, 0.0095, 0.0150, 0.0105, 0.0038, 0.0131, ...
#> $ V55   <dbl> 0.0072, 0.0094, 0.0180, 0.0085, 0.0110, 0.0013, 0.0070, ...
#> $ V56   <dbl> 0.0167, 0.0191, 0.0244, 0.0073, 0.0015, 0.0089, 0.0138, ...
#> $ V57   <dbl> 0.0180, 0.0140, 0.0316, 0.0050, 0.0072, 0.0057, 0.0092, ...
#> $ V58   <dbl> 0.0084, 0.0049, 0.0164, 0.0044, 0.0048, 0.0027, 0.0143, ...
#> $ V59   <dbl> 0.0090, 0.0052, 0.0095, 0.0040, 0.0107, 0.0051, 0.0036, ...
#> $ V60   <dbl> 0.0032, 0.0044, 0.0078, 0.0117, 0.0094, 0.0062, 0.0103, ...
#> $ Class <fct> R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R, R,...
```

```
tibble::as_tibble(Sonar)
```

```
#> # A tibble: 208 x 61
#>        V1     V2     V3     V4     V5     V6     V7     V8     V9    V10
#>     <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
```

```
#>  1 0.02   0.0371 0.0428 0.0207 0.0954 0.0986 0.154  0.160  0.311  0.211
#>  2 0.0453 0.0523 0.0843 0.0689 0.118  0.258  0.216  0.348  0.334  0.287
#>  3 0.0262 0.0582 0.110  0.108  0.0974 0.228  0.243  0.377  0.560  0.619
#>  4 0.01   0.0171 0.0623 0.0205 0.0205 0.0368 0.110  0.128  0.0598 0.126
#>  5 0.0762 0.0666 0.0481 0.0394 0.059  0.0649 0.121  0.247  0.356  0.446
#>  6 0.0286 0.0453 0.0277 0.0174 0.0384 0.099  0.120  0.183  0.210  0.304
#>  7 0.0317 0.0956 0.132  0.141  0.167  0.171  0.0731 0.140  0.208  0.351
#>  8 0.0519 0.0548 0.0842 0.0319 0.116  0.0922 0.103  0.0613 0.146  0.284
#>  9 0.0223 0.0375 0.0484 0.0475 0.0647 0.0591 0.0753 0.0098 0.0684 0.149
#> 10 0.0164 0.0173 0.0347 0.007  0.0187 0.0671 0.106  0.0697 0.0962 0.0251
#> # ... with 198 more rows, and 51 more variables: V11 <dbl>, V12 <dbl>,
#> #   V13 <dbl>, V14 <dbl>, V15 <dbl>, V16 <dbl>, V17 <dbl>, V18 <dbl>,
#> #   V19 <dbl>, V20 <dbl>, V21 <dbl>, V22 <dbl>, V23 <dbl>, V24 <dbl>,
#> #   V25 <dbl>, V26 <dbl>, V27 <dbl>, V28 <dbl>, V29 <dbl>, V30 <dbl>,
#> #   V31 <dbl>, V32 <dbl>, V33 <dbl>, V34 <dbl>, V35 <dbl>, V36 <dbl>,
#> #   V37 <dbl>, V38 <dbl>, V39 <dbl>, V40 <dbl>, V41 <dbl>, V42 <dbl>,
#> #   V43 <dbl>, V44 <dbl>, V45 <dbl>, V46 <dbl>, V47 <dbl>, V48 <dbl>,
#> #   V49 <dbl>, V50 <dbl>, V51 <dbl>, V52 <dbl>, V53 <dbl>, V54 <dbl>,
#> #   V55 <dbl>, V56 <dbl>, V57 <dbl>, V58 <dbl>, V59 <dbl>, V60 <dbl>,
#> #   Class <fct>
```

```
# create 80%/20% for training and validation datasets
validationIndex <- createDataPartition(Sonar$Class, p=0.80, list=FALSE)
validation <- Sonar[-validationIndex,]
training   <- Sonar[validationIndex,]
```

```
# train a model and summarize model
set.seed(7)
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
fit.rf <- train(Class~., data=training,
                method = "rf",
                metric = "Accuracy",
                trControl = trainControl,
                ntree = 2000)

print(fit.rf)
```

```
#> Random Forest
#>
#> 167 samples
#>  60 predictor
#>   2 classes: 'M', 'R'
#>
#> No pre-processing
#> Resampling: Cross-Validated (10 fold, repeated 3 times)
#> Summary of sample sizes: 150, 151, 150, 150, 150, 151, ...
#> Resampling results across tuning parameters:
#>
#>   mtry  Accuracy   Kappa
#>    2    0.8525572  0.7008578
#>   31    0.8184722  0.6341306
#>   60    0.7944526  0.5856805
#>
#> Accuracy was used to select the optimal model using the largest value.
#> The final value used for the model was mtry = 2.
```

```
print(fit.rf$finalModel)
```

```
#>
#> Call:
#>  randomForest(x = x, y = y, ntree = 2000, mtry = param$mtry)
#>                Type of random forest: classification
#>                      Number of trees: 2000
#> No. of variables tried at each split: 2
#>
#>         OOB estimate of  error rate: 12.57%
#> Confusion matrix:
#>    M  R class.error
#> M 84  5  0.05617978
#> R 16 62  0.20512821
```

Accuracy: 85.26% at mtry=2

18.3 Apply tuning parameters for final model

```
# create standalone model using all training data
set.seed(7)
finalModel <- randomForest(Class~., training, mtry=2, ntree=2000)

# make a predictions on "new data" using the final model
finalPredictions <- predict(finalModel, validation[,1:60])
confusionMatrix(finalPredictions, validation$Class)
```

```
#> Confusion Matrix and Statistics
#>
#>           Reference
#> Prediction  M  R
#>          M 20  5
#>          R  2 14
#>
#>                Accuracy : 0.8293
#>                  95% CI : (0.6794, 0.9285)
#>     No Information Rate : 0.5366
#>     P-Value [Acc > NIR] : 8.511e-05
#>
#>                   Kappa : 0.653
#>
#>  Mcnemar's Test P-Value : 0.4497
#>
#>             Sensitivity : 0.9091
#>             Specificity : 0.7368
#>          Pos Pred Value : 0.8000
#>          Neg Pred Value : 0.8750
#>              Prevalence : 0.5366
#>          Detection Rate : 0.4878
#>    Detection Prevalence : 0.6098
#>       Balanced Accuracy : 0.8230
#>
```

```
#>        'Positive' Class : M
#>
```

Accuracy: 82.93%

18.4 Save model

```
# save the model to disk
saveRDS(finalModel, file.path(model_out_dir, "sonar-finalModel.rds"))
```

18.5 Use the saved model

```
# load the model
superModel <- readRDS(file.path(model_out_dir, "sonar-finalModel.rds"))
print(superModel)
```

```
#>
#> Call:
#>  randomForest(formula = Class ~ ., data = training, mtry = 2,      ntree = 2000)
#>                Type of random forest: classification
#>                      Number of trees: 2000
#> No. of variables tried at each split: 2
#>
#>         OOB estimate of  error rate: 14.97%
#> Confusion matrix:
#>    M  R class.error
#> M 81  8  0.08988764
#> R 17 61  0.21794872
```

18.6 Make prediction with new data

```
# make a predictions on "new data" using the final model
finalPredictions <- predict(superModel, validation[,1:60])
confusionMatrix(finalPredictions, validation$Class)
```

```
#> Confusion Matrix and Statistics
#>
#>           Reference
#> Prediction  M  R
#>          M 20  5
#>          R  2 14
#>
#>                Accuracy : 0.8293
#>                  95% CI : (0.6794, 0.9285)
#>     No Information Rate : 0.5366
#>     P-Value [Acc > NIR] : 8.511e-05
#>
#>                   Kappa : 0.653
#>
```

```
#>  Mcnemar's Test P-Value : 0.4497
#>
#>             Sensitivity : 0.9091
#>             Specificity : 0.7368
#>          Pos Pred Value : 0.8000
#>          Neg Pred Value : 0.8750
#>              Prevalence : 0.5366
#>          Detection Rate : 0.4878
#>    Detection Prevalence : 0.6098
#>       Balanced Accuracy : 0.8230
#>
#>        'Positive' Class : M
#>
```

Chapter 19

Glass classification

https://cran.r-project.org/web/packages/e1071/vignettes/svmdoc.pdf

Glass Classification In this example, we use the glass data from the UCI Repository of Machine Learning Databases for classification. The task is to predict the type of a glass on basis of its chemical analysis. We start by splitting the data into a train and test set:

```
library(caret)
library(e1071)
library(rpart)

data(Glass, package="mlbench")
str(Glass)
```

```
#> 'data.frame':    214 obs. of  10 variables:
#>  $ RI  : num  1.52 1.52 1.52 1.52 1.52 ...
#>  $ Na  : num  13.6 13.9 13.5 13.2 13.3 ...
#>  $ Mg  : num  4.49 3.6 3.55 3.69 3.62 3.61 3.6 3.61 3.58 3.6 ...
#>  $ Al  : num  1.1 1.36 1.54 1.29 1.24 1.62 1.14 1.05 1.37 1.36 ...
#>  $ Si  : num  71.8 72.7 73 72.6 73.1 ...
#>  $ K   : num  0.06 0.48 0.39 0.57 0.55 0.64 0.58 0.57 0.56 0.57 ...
#>  $ Ca  : num  8.75 7.83 7.78 8.22 8.07 8.07 8.17 8.24 8.3 8.4 ...
#>  $ Ba  : num  0 0 0 0 0 0 0 0 0 0 ...
#>  $ Fe  : num  0 0 0 0 0 0.26 0 0 0 0.11 ...
#>  $ Type: Factor w/ 6 levels "1","2","3","5",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
## split data into a train and test set
index <- 1:nrow(Glass)
testindex <- sample(index, trunc(length(index)/3))
testset  <- Glass[testindex,]
trainset <- Glass[-testindex,]
```

Both for the SVM and the partitioning tree (via `rpart()`), we fit the model and try to predict the test set values:

```
## svm
svm.model <- svm(Type ~ ., data = trainset, cost = 100, gamma = 1)
svm.pred  <- predict(svm.model, testset[,-10])
```

(The dependent variable, Type, has column number 10. cost is a general penalizing parameter for C-classification and `gamma` is the radial basis function-specific `kernel` parameter.)

```
## rpart
rpart.model <- rpart(Type ~ ., data = trainset)
rpart.pred <- predict(rpart.model, testset[,-10], type = "class")
```

A cross-tabulation of the true versus the predicted values yields:

```
## compute svm confusion matrix
table(pred = svm.pred, true = testset[,10])
```

```
#>     true
#> pred  1  2  3  5  6  7
#>    1 17  7  5  0  0  0
#>    2  4 13  4  3  0  0
#>    3  3  0  2  0  0  0
#>    5  0  0  0  1  0  0
#>    6  0  0  0  0  2  0
#>    7  0  1  0  0  0  9
```

```
## compute rpart confusion matrix
table(pred = rpart.pred, true = testset[,10])
```

```
#>     true
#> pred  1  2  3  5  6  7
#>    1 20  7  8  0  0  0
#>    2  4 14  3  0  1  0
#>    3  0  0  0  0  0  0
#>    5  0  0  0  3  1  0
#>    6  0  0  0  0  0  0
#>    7  0  0  0  1  0  9
```

19.0.1 Comparison test sets

```
confusionMatrix(svm.pred, testset$Type)
```

```
#> Confusion Matrix and Statistics
#>
#>           Reference
#> Prediction  1  2  3  5  6  7
#>          1 17  7  5  0  0  0
#>          2  4 13  4  3  0  0
#>          3  3  0  2  0  0  0
#>          5  0  0  0  1  0  0
#>          6  0  0  0  0  2  0
#>          7  0  1  0  0  0  9
#>
#> Overall Statistics
#>
#>                Accuracy : 0.6197
#>                  95% CI : (0.4967, 0.7324)
#>     No Information Rate : 0.338
#>     P-Value [Acc > NIR] : 1.146e-06
#>
#>                   Kappa : 0.4802
#>
```

```
#>  Mcnemar's Test P-Value : NA
#>
#> Statistics by Class:
#>
#>                      Class: 1 Class: 2 Class: 3 Class: 5 Class: 6 Class: 7
#> Sensitivity            0.7083   0.6190  0.18182  0.25000  1.00000   1.0000
#> Specificity            0.7447   0.7800  0.95000  1.00000  1.00000   0.9839
#> Pos Pred Value         0.5862   0.5417  0.40000  1.00000  1.00000   0.9000
#> Neg Pred Value         0.8333   0.8298  0.86364  0.95714  1.00000   1.0000
#> Prevalence             0.3380   0.2958  0.15493  0.05634  0.02817   0.1268
#> Detection Rate         0.2394   0.1831  0.02817  0.01408  0.02817   0.1268
#> Detection Prevalence   0.4085   0.3380  0.07042  0.01408  0.02817   0.1408
#> Balanced Accuracy      0.7265   0.6995  0.56591  0.62500  1.00000   0.9919
```

```
confusionMatrix(rpart.pred, testset$Type)
```

```
#> Confusion Matrix and Statistics
#>
#>           Reference
#> Prediction  1  2  3  5  6  7
#>          1 20  7  8  0  0  0
#>          2  4 14  3  0  1  0
#>          3  0  0  0  0  0  0
#>          5  0  0  0  3  1  0
#>          6  0  0  0  0  0  0
#>          7  0  0  0  1  0  9
#>
#> Overall Statistics
#>
#>                Accuracy : 0.6479
#>                  95% CI : (0.5254, 0.7576)
#>     No Information Rate : 0.338
#>     P-Value [Acc > NIR] : 9.673e-08
#>
#>                   Kappa : 0.5114
#>
#>  Mcnemar's Test P-Value : NA
#>
#> Statistics by Class:
#>
#>                      Class: 1 Class: 2 Class: 3 Class: 5 Class: 6 Class: 7
#> Sensitivity            0.8333   0.6667   0.0000  0.75000  0.00000   1.0000
#> Specificity            0.6809   0.8400   1.0000  0.98507  1.00000   0.9839
#> Pos Pred Value         0.5714   0.6364      NaN  0.75000      NaN   0.9000
#> Neg Pred Value         0.8889   0.8571   0.8451  0.98507  0.97183   1.0000
#> Prevalence             0.3380   0.2958   0.1549  0.05634  0.02817   0.1268
#> Detection Rate         0.2817   0.1972   0.0000  0.04225  0.00000   0.1268
#> Detection Prevalence   0.4930   0.3099   0.0000  0.05634  0.00000   0.1408
#> Balanced Accuracy      0.7571   0.7533   0.5000  0.86754  0.50000   0.9919
```

19.0.2 Comparison with resamples

Finally, we compare the performance of the two methods by computing the respective accuracy rates and the kappa indices (as computed by `classAgreement()` also contained in package `e1071`). In Table 1, we

summarize the results of 10 replications—Support Vector Machines show better results.

```
set.seed(1234567)

# SVM
fit.svm <- train(Type ~., data = trainset,
                 method = "svmRadial")

# Random Forest
fit.rpart <- train(Type ~., data = trainset,
                method="rpart")

# collect resamples
results <- resamples(list(svm = fit.svm,
                          rpart  = fit.rpart))

summary(results)
```

```
#>
#> Call:
#> summary.resamples(object = results)
#>
#> Models: svm, rpart
#> Number of resamples: 25
#>
#> Accuracy
#>            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
#> svm   0.5454545 0.6296296 0.6666667 0.6754137 0.7254902 0.8269231    0
#> rpart 0.5000000 0.5740741 0.6181818 0.6172868 0.6666667 0.7031250    0
#>
#> Kappa
#>            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
#> svm   0.3325243 0.4482759 0.5134680 0.5247176 0.5802469 0.7432803    0
#> rpart 0.2921857 0.3934837 0.4425428 0.4448108 0.5079455 0.5690999    0
```

Chapter 20

Ozone SVM

https://cran.r-project.org/web/packages/e1071/vignettes/svmdoc.pdf

```
library(e1071)
library(rpart)

data(Ozone, package="mlbench")
## split data into a train and test set
index <- 1:nrow(Ozone)
testindex <- sample(index, trunc(length(index)/3))
testset <- na.omit(Ozone[testindex,-3])
trainset <- na.omit(Ozone[-testindex,-3])
```

```
## svm
svm.model <- svm(V4 ~ ., data = trainset, cost = 1000, gamma = 0.0001)
svm.pred  <- predict(svm.model, testset[,-3])
crossprod(svm.pred - testset[,3]) / length(testindex)
```

```
##          [,1]
## [1,] 9.965266
```

```
## rpart
rpart.model <- rpart(V4 ~ ., data = trainset)
rpart.pred  <- predict(rpart.model, testset[,-3])
crossprod(rpart.pred - testset[,3]) / length(testindex)
```

```
##          [,1]
## [1,] 21.15639
```

Chapter 21

A gentle introduction to support vector machines using R

https://eight2late.wordpress.com/2017/02/07/a-gentle-introduction-to-support-vector-machines-using-r/

21.1 Support vector machines in R

In this demo we'll use the svm interface that is implemented in the `e1071` R package. This interface provides R programmers access to the comprehensive `libsvm` library written by Chang and Lin. I'll use two toy datasets: the famous iris dataset available with the base R package and the sonar dataset from the mlbench package. I won't describe details of the datasets as they are discussed at length in the documentation that I have linked to. However, it is worth mentioning the reasons why I chose these datasets:

As mentioned earlier, no real life dataset is linearly separable, but the iris dataset is almost so. Consequently, it is a good illustration of using linear SVMs. Although one almost never uses these in practice, I have illustrated their use primarily for pedagogical reasons. The sonar dataset is a good illustration of the benefits of using RBF kernels in cases where the dataset is hard to visualise (60 variables in this case!). In general, one would almost always use RBF (or other nonlinear) kernels in practice.

With that said, let's get right to it. I assume you have R and RStudio installed. For instructions on how to do this, have a look at the first article in this series. The processing preliminaries – loading libraries, data and creating training and test datasets are much the same as in my previous articles so I won't dwell on these here. For completeness, however, I'll list all the code so you can run it directly in R or R studio (a complete listing of the code can be found here):

21.2 SVM on `iris` dataset

21.2.1 Training and test datasets

```
#load required library
library(e1071)

#load built-in iris dataset
data(iris)
```

```
#set seed to ensure reproducible results
set.seed(42)

#split into training and test sets
iris[, "train"] <- ifelse(runif(nrow(iris)) < 0.8, 1, 0)

#separate training and test sets
trainset <- iris[iris$train == 1,]
testset <- iris[iris$train == 0,]

#get column index of train flag
trainColNum <- grep("train", names(trainset))

#remove train flag column from train and test sets
trainset <- trainset[,-trainColNum]
testset <- testset[,-trainColNum]

dim(trainset)
```

```
## [1] 115   5
```

```
dim(testset)
```

```
## [1] 35  5
```

21.2.2 Build the SVM model

```
#get column index of predicted variable in dataset
typeColNum <- grep("Species", names(iris))

#build model - linear kernel and C-classification (soft margin) with default cost (C=1)
svm_model <- svm(Species~ ., data = trainset,
                 method = "C-classification",
                 kernel = "linear")
svm_model
```

```
##
## Call:
## svm(formula = Species ~ ., data = trainset, method = "C-classification",
##     kernel = "linear")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  1
##       gamma:  0.25
##
## Number of Support Vectors:  24
```

The output from the SVM model show that there are 24 support vectors. If desired, these can be examined using the SV variable in the model – i.e via svm_model$SV.

21.2.3 Support Vectors

```
# support vectors
svm_model$SV
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width
## 19   -0.25639203  1.76683447   -1.3228618  -1.3054201
## 42   -1.70055936 -1.70445193   -1.5591789  -1.3054201
## 45   -0.97847569  1.76683447   -1.2047033  -1.1709010
## 53    1.18777530  0.14690082    0.5676747   0.3088091
## 55    0.70638619 -0.54735646    0.3904369   0.3088091
## 57    0.46569164  0.60973900    0.4495161   0.4433282
## 58   -1.21917025 -1.47303284   -0.3775936  -0.3637864
## 69    0.34534436 -1.93587102    0.3313576   0.3088091
## 71   -0.01569747  0.37831991    0.5085954   0.7123664
## 73    0.46569164 -1.24161374    0.5676747   0.3088091
## 78    0.94708075 -0.08451828    0.6267539   0.5778473
## 84    0.10464981 -0.77877556    0.6858332   0.4433282
## 85   -0.61743386 -0.08451828    0.3313576   0.3088091
## 86    0.10464981  0.84115809    0.3313576   0.4433282
## 99   -0.97847569 -1.24161374   -0.5548314  -0.2292673
## 107  -1.21917025 -1.24161374    0.3313576   0.5778473
## 111   0.70638619  0.37831991    0.6858332   0.9814046
## 117   0.70638619 -0.08451828    0.9221503   0.7123664
## 124   0.46569164 -0.77877556    0.5676747   0.7123664
## 130   1.54881714 -0.08451828    1.0993881   0.4433282
## 138   0.58603892  0.14690082    0.9221503   0.7123664
## 139   0.10464981 -0.08451828    0.5085954   0.7123664
## 147   0.46569164 -1.24161374    0.6267539   0.8468855
## 150  -0.01569747 -0.08451828    0.6858332   0.7123664
```

The test prediction accuracy indicates that the linear performs quite well on this dataset, confirming that it is indeed near linearly separable. To check performance by class, one can create a confusion matrix as described in my post on random forests. I'll leave this as an exercise for you. Another point is that we have used a soft-margin classification scheme with a cost C=1. You can experiment with this by explicitly changing the value of C. Again, I'll leave this for you an exercise.

21.2.4 Predictions on training model

```
# training set predictions
pred_train <- predict(svm_model, trainset)
mean(pred_train == trainset$Species)
```

```
## [1] 0.9826087
```

```
# [1] 0.9826087
```

21.2.5 Predictions on test model

```
# test set predictions
pred_test <-predict(svm_model, testset)
mean(pred_test == testset$Species)
```

```
## [1] 0.9142857
```

```
# [1] 0.9142857
```

21.2.6 Confusion matrix and Accuracy

```
# confusion matrix
cm <- table(pred_test, testset$Species)
cm
```

```
##
## pred_test    setosa versicolor virginica
##   setosa         18          0         0
##   versicolor      0          5         3
##   virginica       0          0         9
```

```
# accuracy
sum(diag(cm)) / sum(cm)
```

```
## [1] 0.9142857
```

21.3 SVM with Radial Basis Function kernel. Linear

21.3.1 Training and test sets

```
#load required library (assuming e1071 is already loaded)
library(mlbench)

#load Sonar dataset
data(Sonar)
#set seed to ensure reproducible results
set.seed(42)
#split into training and test sets
Sonar[, "train"] <- ifelse(runif(nrow(Sonar))<0.8,1,0)

#separate training and test sets
trainset <- Sonar[Sonar$train==1,]
testset <- Sonar[Sonar$train==0,]

#get column index of train flag
trainColNum <- grep("train",names(trainset))
#remove train flag column from train and test sets
trainset <- trainset[,-trainColNum]
testset <- testset[,-trainColNum]

#get column index of predicted variable in dataset
typeColNum <- grep("Class",names(Sonar))
```

21.3.2 Predictions on Training model

```
#build model - linear kernel and C-classification with default cost (C=1)
svm_model <- svm(Class~ ., data=trainset,
                 method="C-classification",
                 kernel="linear")

#training set predictions
pred_train <-predict(svm_model,trainset)
mean(pred_train==trainset$Class)
```

```
## [1] 0.969697
```

21.3.3 Predictions on test model

```
#test set predictions
pred_test <-predict(svm_model,testset)
mean(pred_test==testset$Class)
```

```
## [1] 0.6046512
```

I'll leave you to examine the contents of the model. The important point to note here is that the performance of the model with the test set is quite dismal compared to the previous case. This simply indicates that the linear kernel is not appropriate here. Let's take a look at what happens if we use the RBF kernel with default values for the parameters:

21.4 SVM with Radial Basis Function kernel. Non-linear

21.4.1 Predictions on training model

```
#build model: radial kernel, default params
svm_model <- svm(Class~ ., data=trainset,
                 method="C-classification",
                 kernel="radial")
# print params
svm_model$cost
```

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

```
svm_model$gamma
```

```
## [1] 0.01666667
```

```
#training set predictions
pred_train <-predict(svm_model,trainset)
mean(pred_train==trainset$Class)
```

```
## [1] 0.9878788
```

21.4.2 Predictions on test model

```
#test set predictions
pred_test <-predict(svm_model,testset)
mean(pred_test==testset$Class)
```

```
## [1] 0.7674419
```

That's a pretty decent improvement from the linear kernel. Let's see if we can do better by doing some parameter tuning. To do this we first invoke tune.svm and use the parameters it gives us in the call to svm:

21.4.3 Tuning of parameters

```
# find optimal parameters in a specified range
tune_out <- tune.svm(x = trainset[,-typeColNum],
                     y = trainset[, typeColNum],
                     gamma = 10^(-3:3),
                     cost = c(0.01, 0.1, 1, 10, 100, 1000),
                     kernel = "radial")

#print best values of cost and gamma
tune_out$best.parameters$cost
```

```
## [1] 100
```

```
tune_out$best.parameters$gamma
```

```
## [1] 0.01
```

```
#build model
svm_model <- svm(Class~ ., data = trainset,
                 method = "C-classification",
                 kernel = "radial",
                 cost = tune_out$best.parameters$cost,
                 gamma = tune_out$best.parameters$gamma)
```

21.4.4 Prediction on training model with new parameters

```
# training set predictions
pred_train <-predict(svm_model,trainset)
mean(pred_train==trainset$Class)
```

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

21.4.5 Prediction on test model with new parameters

```
# test set predictions
pred_test <-predict(svm_model,testset)
mean(pred_test==testset$Class)
```

```
## [1] 0.8139535
```

Which is fairly decent improvement on the un-optimised case.

21.5 Wrapping up

This bring us to the end of this introductory exploration of SVMs in R. To recap, the distinguishing feature of SVMs in contrast to most other techniques is that they attempt to construct optimal separation boundaries between different categories.

SVMs are quite versatile and have been applied to a wide variety of domains ranging from chemistry to pattern recognition. They are best used in binary classification scenarios. This brings up a question as to where SVMs are to be preferred to other binary classification techniques such as logistic regression. The honest response is, "it depends" – but here are some points to keep in mind when choosing between the two. A general point to keep in mind is that SVM algorithms tend to be expensive both in terms of memory and computation, issues that can start to hurt as the size of the dataset increases.

Given all the above caveats and considerations, the best way to figure out whether an SVM approach will work for your problem may be to do what most machine learning practitioners do: try it out!

Chapter 22

Multiclass classification. iris

```
# load the caret package
library(caret)
# attach the iris dataset to the environment
data(iris)
# rename the dataset
dataset <- iris
```

We will split the loaded dataset into two, 80% of which we will use to train our models and 20% that we will hold back as a validation dataset.

```
# create a list of 80% of the rows in the original dataset we can use for training
validationIndex <- createDataPartition(dataset$Species, p=0.80, list=FALSE)
# select 20% of the data for validation
validation <- dataset[-validationIndex,]

# use the remaining 80% of data to training and testing the models
dataset <- dataset[validationIndex,]

# dimensions of dataset
dim(dataset)
```

```
#> [1] 120   5
```

```
# list types for each attribute
sapply(dataset, class)
```

```
#> Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
#>    "numeric"    "numeric"    "numeric"    "numeric"     "factor"
```

22.1 Peek at the dataset

```
# take a peek at the first 5 rows of the data
head(dataset)
```

```
#>   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
```

```
#> 4          4.6         3.1          1.5         0.2  setosa
#> 6          5.4         3.9          1.7         0.4  setosa
#> 8          5.0         3.4          1.5         0.2  setosa
```

```
library(dplyr)

glimpse(dataset)
```

```
#> Observations: 120
#> Variables: 5
#> $ Sepal.Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.4, 5.0, 4.4, 4.9, 5.4, 4.8,...
#> $ Sepal.Width  <dbl> 3.5, 3.0, 3.2, 3.1, 3.9, 3.4, 2.9, 3.1, 3.7, 3.4,...
#> $ Petal.Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.7, 1.5, 1.4, 1.5, 1.5, 1.6,...
#> $ Petal.Width  <dbl> 0.2, 0.2, 0.2, 0.2, 0.4, 0.2, 0.2, 0.1, 0.2, 0.2,...
#> $ Species      <fct> setosa, setosa, setosa, setosa, setosa, setosa, s...
```

```
library(skimr)

skim(dataset)
```

```
#> Skim summary statistics
#>  n obs: 120
#>  n variables: 5
#>
#> -- Variable type:factor -------------------------------------------------------
#>  variable missing complete   n n_unique                       top_counts
#>   Species       0      120 120        3 set: 40, ver: 40, vir: 40, NA: 0
#>  ordered
#>    FALSE
#>
#> -- Variable type:numeric ------------------------------------------------------
#>      variable missing complete   n mean   sd  p0 p25 p50 p75 p100     hist
#>  Petal.Length       0      120 120 3.77 1.78 1   1.6 4.4 5.1  6.9
#>   Petal.Width       0      120 120 1.2  0.76 0.1 0.3 1.3 1.8  2.5
#>  Sepal.Length       0      120 120 5.85 0.82 4.3 5.1 5.8 6.4  7.7
#>   Sepal.Width       0      120 120 3.06 0.43 2   2.8 3   3.3  4.4
```

22.2 Levels of the class

```
# list the levels for the class
levels(dataset$Species)
```

```
#> [1] "setosa"     "versicolor" "virginica"
```

22.3 class distribution

```
# summarize the class distribution
percentage <- prop.table(table(dataset$Species)) * 100
cbind(freq=table(dataset$Species), percentage=percentage)
```

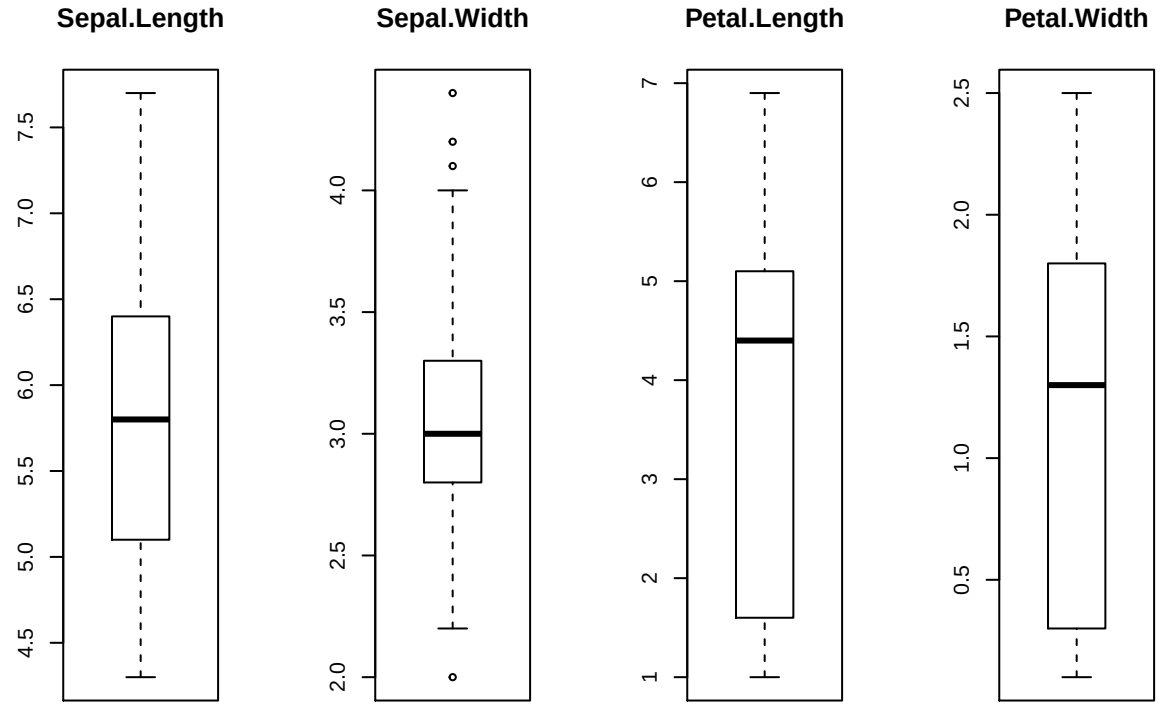
```
#>            freq percentage
#> setosa       40   33.33333
```

```
#> versicolor   40   33.33333
#> virginica    40   33.33333
```

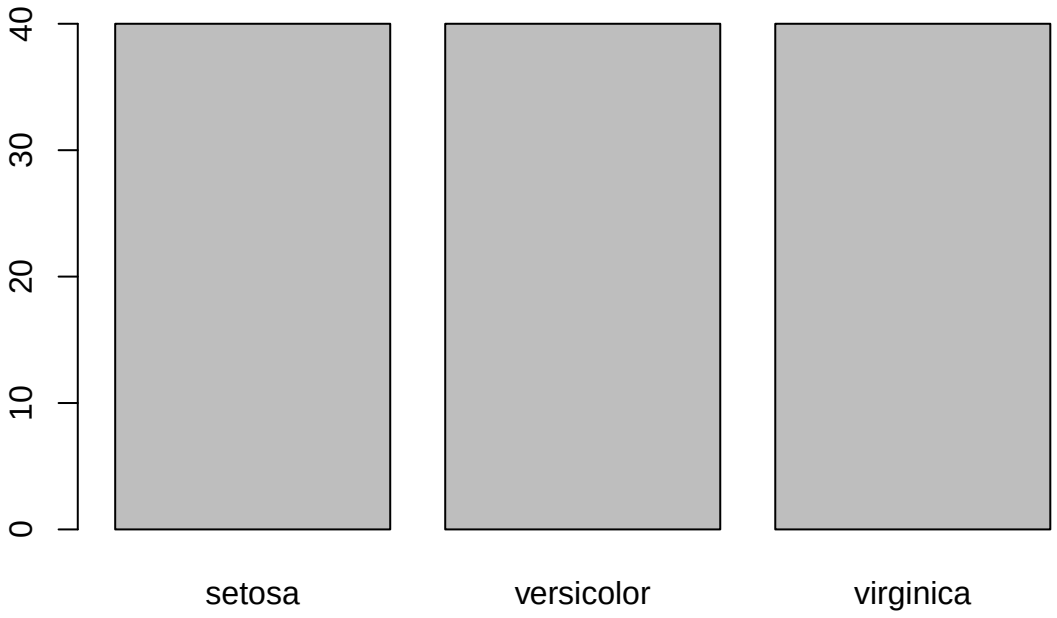
22.4 Visualize the dataset

```
# split input and output
x <- dataset[,1:4]
y <- dataset[,5]
```

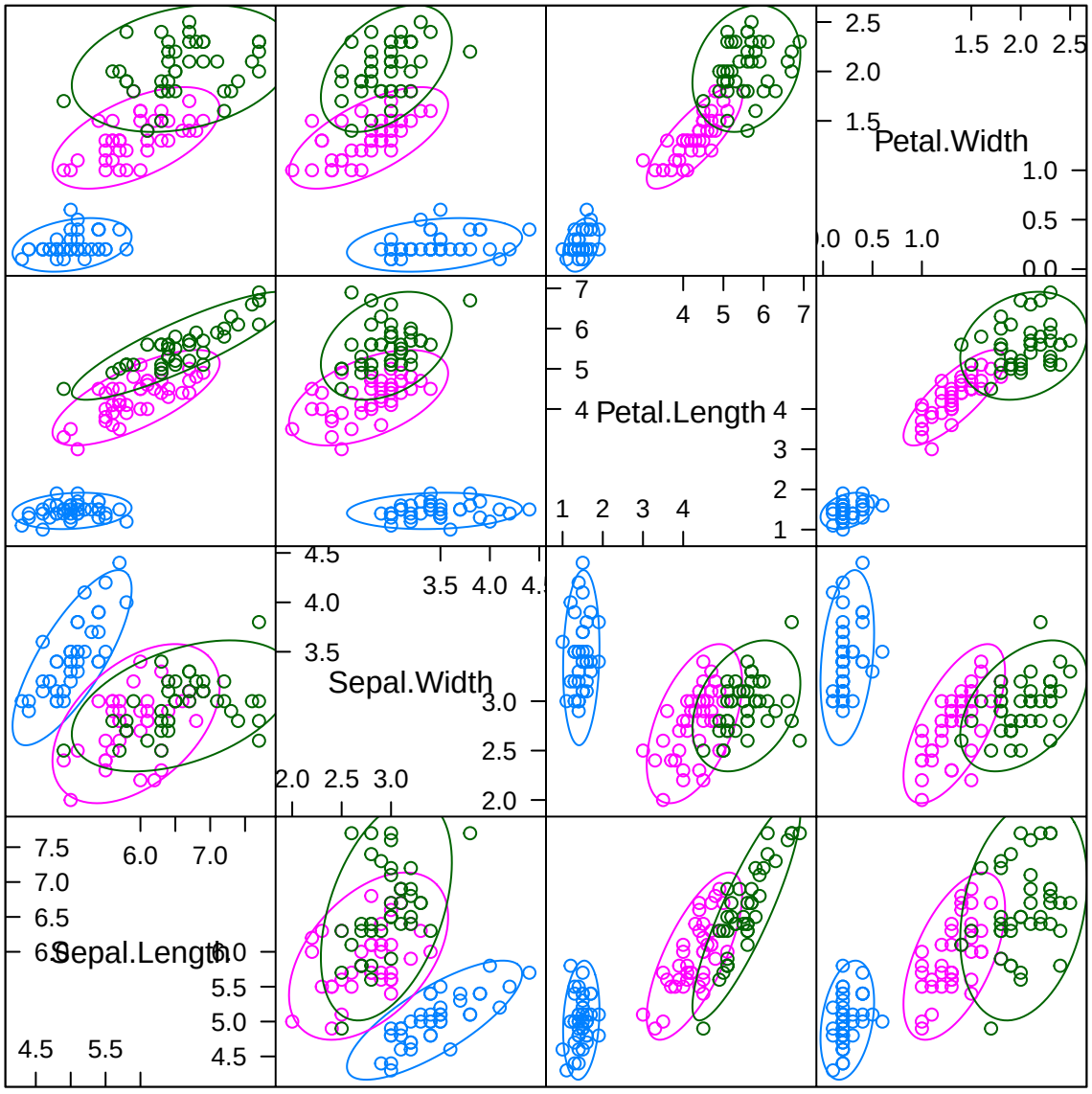
```
# boxplot for each attribute on one image
par(mfrow=c(1,4))
for(i in 1:4) {
    boxplot(x[,i], main=names(dataset)[i])
}
```




```
# barplot for class breakdown
plot(y)
```

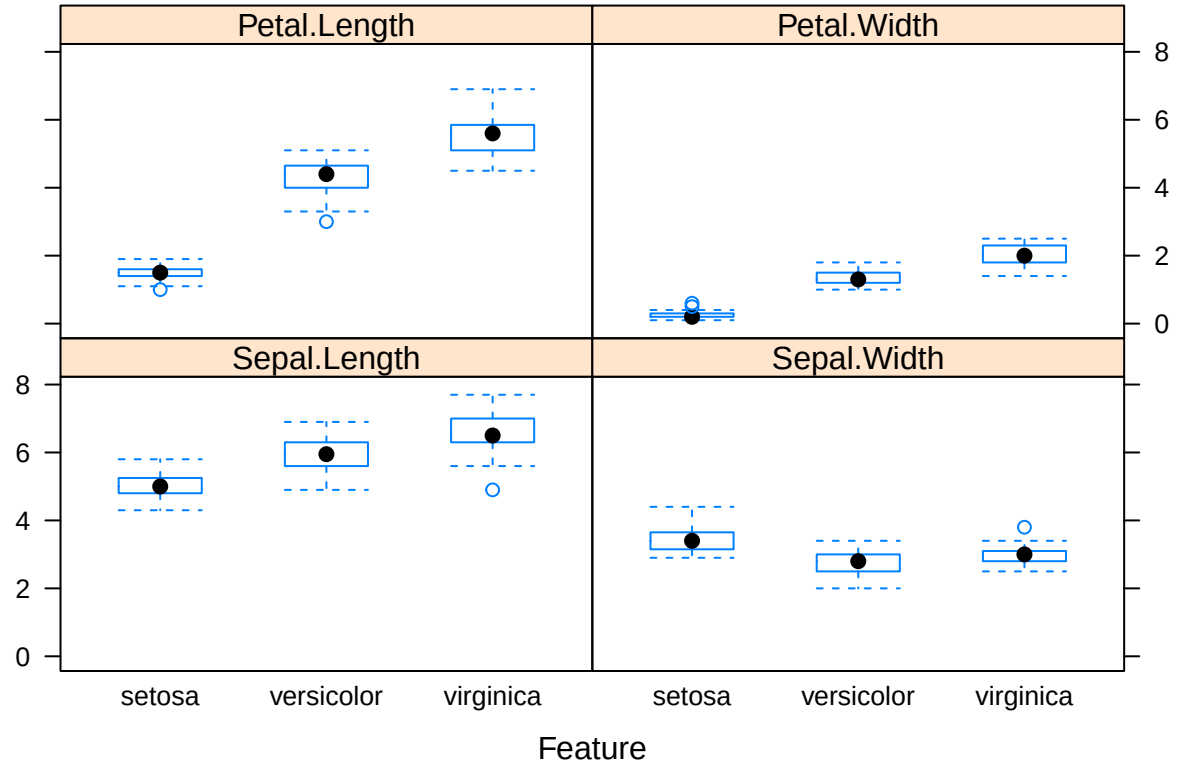



```
# scatter plot matrix
featurePlot(x=x, y=y, plot="ellipse")
```

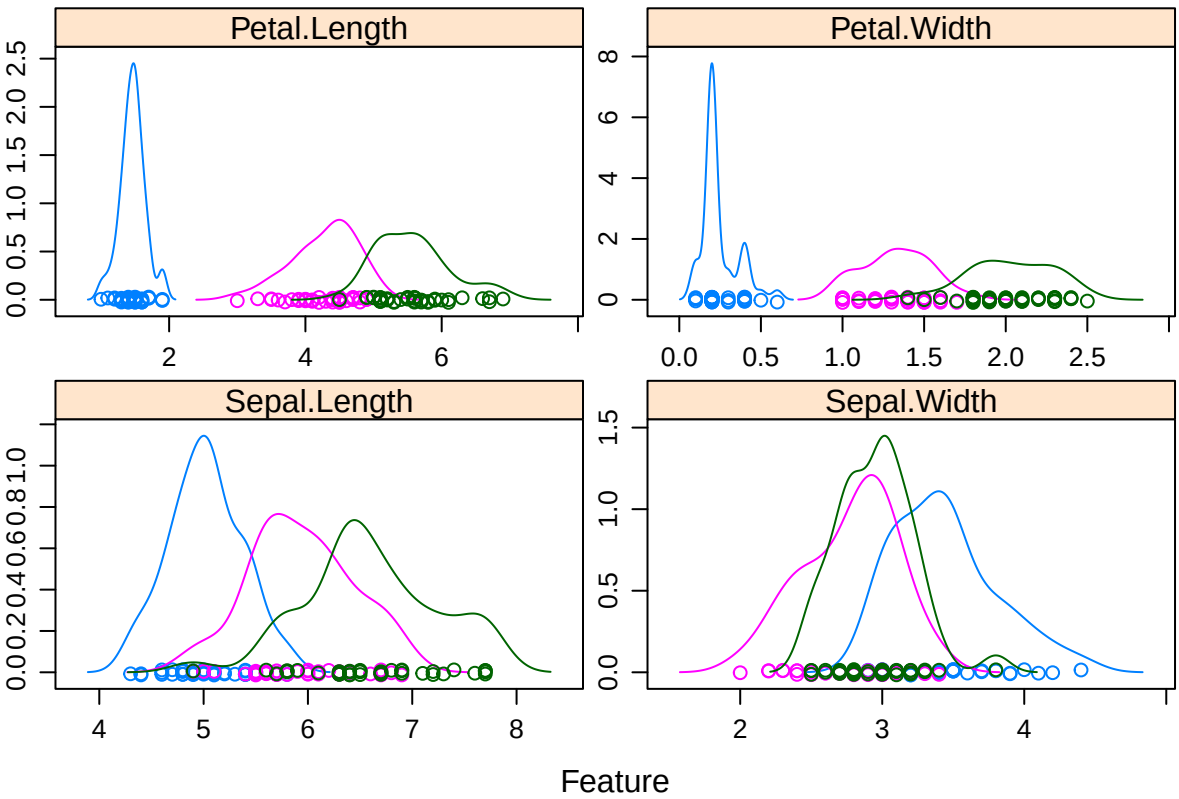


Scatter Plot Matrix

```
# box and whisker plots for each attribute
featurePlot(x=x, y=y, plot="box")
```




```
# density plots for each attribute by class value
scales <- list(x=list(relation="free"), y=list(relation="free"))
featurePlot(x=x, y=y, plot="density", scales=scales)
```



22.5 Evaluate algorithms

22.5.1 split and metrics

```
# Run algorithms using 10-fold cross-validation
trainControl <- trainControl(method="cv", number=10)
metric <- "Accuracy"
```

22.5.2 build models

```
# LDA
set.seed(7)
fit.lda <- train(Species~., data=dataset, method = "lda",
                 metric=metric, trControl=trainControl)
# CART
set.seed(7)
fit.cart <- train(Species~., data=dataset, method = "rpart",
                  metric=metric, trControl=trainControl)
# KNN
set.seed(7)
fit.knn <- train(Species~., data=dataset, method = "knn",
                 metric=metric, trControl=trainControl)
# SVM
set.seed(7)
fit.svm <- train(Species~., data=dataset, method = "svmRadial",
                 metric=metric, trControl=trainControl)
# Random Forest
set.seed(7)
fit.rf <- train(Species~., data=dataset, method = "rf",
                metric=metric, trControl=trainControl)
```

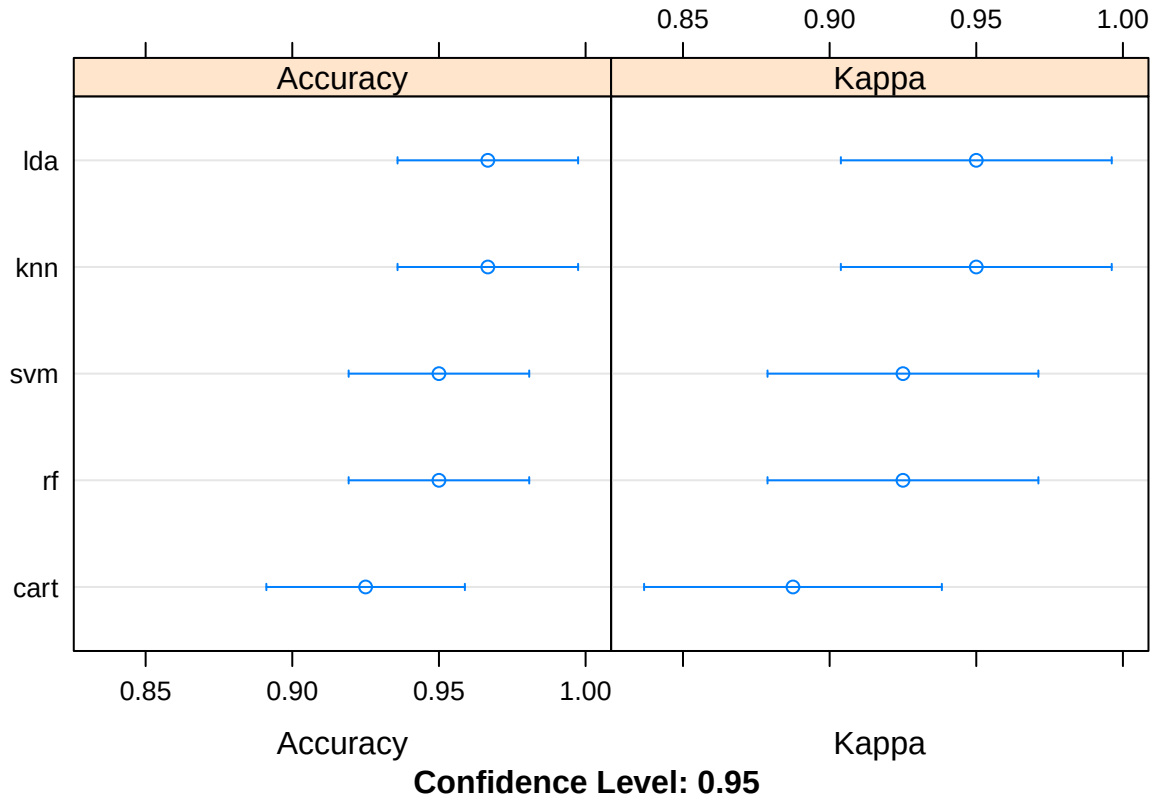
22.5.3 compare

```
#summarize accuracy of models
results <- resamples(list(lda  = fit.lda,
                          cart = fit.cart,
                          knn  = fit.knn,
                          svm  = fit.svm,
                          rf   = fit.rf))
summary(results)
```

```
#>
#> Call:
#> summary.resamples(object = results)
#>
#> Models: lda, cart, knn, svm, rf
#> Number of resamples: 10
#>
#> Accuracy
#>           Min.   1st Qu.    Median      Mean   3rd Qu. Max. NA's
```

```
#> lda  0.9166667 0.9166667 1.0000000 0.9666667 1.0000000    1    0
#> cart 0.8333333 0.9166667 0.9166667 0.9250000 0.9166667    1    0
#> knn  0.9166667 0.9166667 1.0000000 0.9666667 1.0000000    1    0
#> svm  0.9166667 0.9166667 0.9166667 0.9500000 1.0000000    1    0
#> rf   0.9166667 0.9166667 0.9166667 0.9500000 1.0000000    1    0
#>
#> Kappa
#>       Min. 1st Qu. Median   Mean 3rd Qu. Max. NA's
#> lda  0.875   0.875  1.000 0.9500   1.000    1    0
#> cart 0.750   0.875  0.875 0.8875   0.875    1    0
#> knn  0.875   0.875  1.000 0.9500   1.000    1    0
#> svm  0.875   0.875  0.875 0.9250   1.000    1    0
#> rf   0.875   0.875  0.875 0.9250   1.000    1    0
```

```
# compare accuracy of models
dotplot(results)
```




```
# summarize Best Model
print(fit.lda)
```

```
#> Linear Discriminant Analysis
#>
#> 120 samples
#>   4 predictor
#>   3 classes: 'setosa', 'versicolor', 'virginica'
#>
#> No pre-processing
#> Resampling: Cross-Validated (10 fold)
#> Summary of sample sizes: 108, 108, 108, 108, 108, 108, ...
```

```
#> Resampling results:
#>
#>   Accuracy   Kappa
#>   0.9666667  0.95
```

22.6 Make predictions

```
# estimate skill of LDA on the validation dataset
predictions <- predict(fit.lda, validation)
confusionMatrix(predictions, validation$Species)
```

```
#> Confusion Matrix and Statistics
#>
#>             Reference
#> Prediction   setosa versicolor virginica
#>   setosa         10          0         0
#>   versicolor      0         10         0
#>   virginica       0          0        10
#>
#> Overall Statistics
#>
#>                Accuracy : 1
#>                  95% CI : (0.8843, 1)
#>     No Information Rate : 0.3333
#>     P-Value [Acc > NIR] : 4.857e-15
#>
#>                   Kappa : 1
#>
#>  Mcnemar's Test P-Value : NA
#>
#> Statistics by Class:
#>
#>                      Class: setosa Class: versicolor Class: virginica
#> Sensitivity                 1.0000            1.0000           1.0000
#> Specificity                 1.0000            1.0000           1.0000
#> Pos Pred Value              1.0000            1.0000           1.0000
#> Neg Pred Value              1.0000            1.0000           1.0000
#> Prevalence                  0.3333            0.3333           0.3333
#> Detection Rate              0.3333            0.3333           0.3333
#> Detection Prevalence        0.3333            0.3333           0.3333
#> Balanced Accuracy           1.0000            1.0000           1.0000
```

Bibliography