MAE 5905: Introdução à Ciência de Dados - Prova 2

Leonardo Makoto - 7180679

11/07/2023

```
# carregando pacotes gerais
library(tidyverse)
```

Questão 1 (3,0 pontos)

Considere o conjunto de dados `Boston` do pacote `MASS`, contendo n = 506 amostras e p = 14 variáveis. Considere o conjunto de treinamento contendo as primeiras 253 amostras e o conjunto teste contendo as amostras restantes. Ajuste uma árvore de regressão, considerando a variável `medv` como resposta.

```
# carregando bibliotecas e dados

library(MASS)

# carregando boston
data("Boston")
boston <-  Boston
attach(boston)

# visualizando uma amostrade boston, com características da base
glimpse(boston)
```

```
## Rows: 506
## Columns: 14
## $ crim    <dbl> 0.00632, 0.02731, 0.02729, 0.03237, 0.0690~
## $ zn      <dbl> 18.0, 0.0, 0.0, 0.0, 0.0, 0.0, 12.5, 12.5,~
## $ indus   <dbl> 2.31, 7.07, 7.07, 2.18, 2.18, 2.18, 7.87, ~
## $ chas    <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ~
## $ nox     <dbl> 0.538, 0.469, 0.469, 0.458, 0.458, 0.458, ~
## $ rm      <dbl> 6.575, 6.421, 7.185, 6.998, 7.147, 6.430, ~
## $ age     <dbl> 65.2, 78.9, 61.1, 45.8, 54.2, 58.7, 66.6, ~
## $ dis     <dbl> 4.0900, 4.9671, 4.9671, 6.0622, 6.0622, 6.~
## $ rad     <int> 1, 2, 2, 3, 3, 3, 5, 5, 5, 5, 5, 5, 5, 4, ~
## $ tax     <dbl> 296, 242, 242, 222, 222, 222, 311, 311, 31~
## $ ptratio <dbl> 15.3, 17.8, 17.8, 18.7, 18.7, 18.7, 15.2, ~
## $ black   <dbl> 396.90, 396.90, 392.83, 394.63, 396.90, 39~
## $ lstat   <dbl> 4.98, 9.14, 4.03, 2.94, 5.33, 5.21, 12.43,~
## $ medv    <dbl> 24.0, 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, ~
```

```
# algumas estatísticas básicas de boston
summary(boston)
```

```
##       crim               zn             indus       
##  Min.   : 0.00632   Min.   :  0.00   Min.   : 0.46  
##  1st Qu.: 0.08205   1st Qu.:  0.00   1st Qu.: 5.19  
##  Median : 0.25651   Median :  0.00   Median : 9.69  
##  Mean   : 3.61352   Mean   : 11.36   Mean   :11.14  
##  3rd Qu.: 3.67708   3rd Qu.: 12.50   3rd Qu.:18.10  
##  Max.   :88.97620   Max.   :100.00   Max.   :27.74  
##       chas              nox               rm        
##  Min.   :0.00000   Min.   :0.3850   Min.   :3.561  
##  1st Qu.:0.00000   1st Qu.:0.4490   1st Qu.:5.886  
##  Median :0.00000   Median :0.5380   Median :6.208  
##  Mean   :0.06917   Mean   :0.5547   Mean   :6.285  
##  3rd Qu.:0.00000   3rd Qu.:0.6240   3rd Qu.:6.623  
##  Max.   :1.00000   Max.   :0.8710   Max.   :8.780  
##       age              dis              rad        
##  Min.   :  2.90   Min.   : 1.130   Min.   : 1.000  
##  1st Qu.: 45.02   1st Qu.: 2.100   1st Qu.: 4.000  
##  Median : 77.50   Median : 3.207   Median : 5.000  
##  Mean   : 68.57   Mean   : 3.795   Mean   : 9.549  
##  3rd Qu.: 94.08   3rd Qu.: 5.188   3rd Qu.:24.000  
##  Max.   :100.00   Max.   :12.127   Max.   :24.000  
##       tax             ptratio          black       
##  Min.   :187.0   Min.   :12.60   Min.   :  0.32  
##  1st Qu.:279.0   1st Qu.:17.40   1st Qu.:375.38  
##  Median :330.0   Median :19.05   Median :391.44  
##  Mean   :408.2   Mean   :18.46   Mean   :356.67  
##  3rd Qu.:666.0   3rd Qu.:20.20   3rd Qu.:396.23  
##  Max.   :711.0   Max.   :22.00   Max.   :396.90  
##      lstat            medv      
##  Min.   : 1.73   Min.   : 5.00  
##  1st Qu.: 6.95   1st Qu.:17.02  
##  Median :11.36   Median :21.20  
##  Mean   :12.65   Mean   :22.53  
##  3rd Qu.:16.95   3rd Qu.:25.00  
##  Max.   :37.97   Max.   :50.00
```

```
# definindo amostra de treinamento e de teste
boston_treinamento <- boston[1:253,]
boston_teste <- boston[254:nrow(boston),]
```

(a) Ajuste um modelo de árvore aos dados de treinamento. Verifique se é necessário podar a árvore.

```
# carregando bibliotecas
library(tree)

# ajudando modelo aos dados de treinamento
set.seed(123)

boston_treinamento_tree <- tree(medv ~ crim + zn + indus + chas + nox + rm + age + dis + rad + tax + ptratio + black + lsta
t, data = boston_treinamento )
boston_treinamento_tree
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 253 17520.00 24.31
##    2) rm < 6.92 210  4283.00 21.35
##      4) lstat < 10.14 81   957.80 25.13
##        8) rm < 6.5285 54   366.20 23.54 *
##        9) rm > 6.5285 27   180.70 28.32 *
##      5) lstat > 10.14 129  1434.00 18.97
##       10) lstat < 16.085 78   515.20 20.37 *
##       11) lstat > 16.085 51   530.50 16.82
##         22) crim < 0.65402 29   255.70 18.70 *
##         23) crim > 0.65402 22    36.63 14.34 *
##    3) rm > 6.92 43  2402.00 38.77
##      6) rm < 7.4525 25   338.00 33.40 *
##      7) rm > 7.4525 18   339.80 46.23 *
```

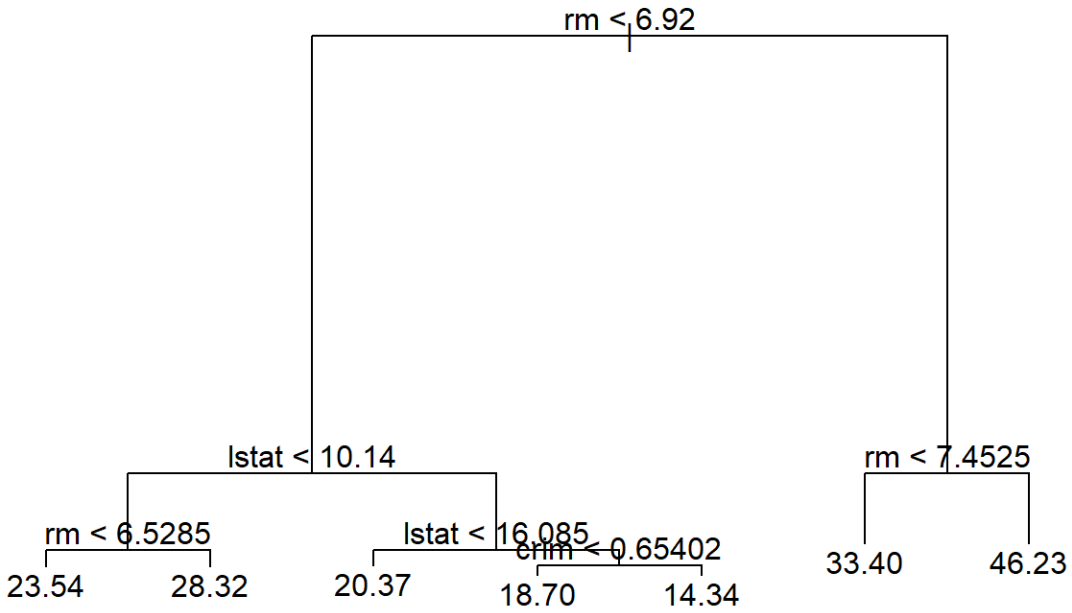
```
summary(boston_treinamento_tree)
```

```
##
## Regression tree:
## tree(formula = medv ~ crim + zn + indus + chas + nox + rm + age +
##     dis + rad + tax + ptratio + black + lstat, data = boston_treinamento)
## Variables actually used in tree construction:
## [1] "rm"    "lstat" "crim"
## Number of terminal nodes:  7
## Residual mean deviance:  8.261 = 2032 / 246
## Distribution of residuals:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -9.796000 -1.672000 -0.003448  0.000000  1.504000 12.660000
```

Note que o output do **summary()** mostra que somente 3 variáveis (rm (número médio de quartos por habitação),lstat (% de status social mais baixo da população) e crim (taxa de criminalidade per capita por cidade)) foram usadas para construir a árvore. **Deviance** mostra a soma do erros quadrados para uma árvore.

Vamos plotar a árvore:

```
plot(boston_treinamento_tree)
text(boston_treinamento_tree , pretty = 0)
```



A árvore indica que para valores mais altos de número médio de quartos por habitação, correspondem para valores mais altos de medv (a variável resposta: Valor médio de casas ocupadas pelos proprietários em US$ 1.000.). Ainda, no grupo de valores menos altos de número médio de quartos por habitação (rm < 6,92), aqueles que tem valores menores de lstat (% de status social mais baixo da população), tem status valroes mais altos de medv.

Para testar se podar a árvore é necessário, vamos usar *cross-validation*.

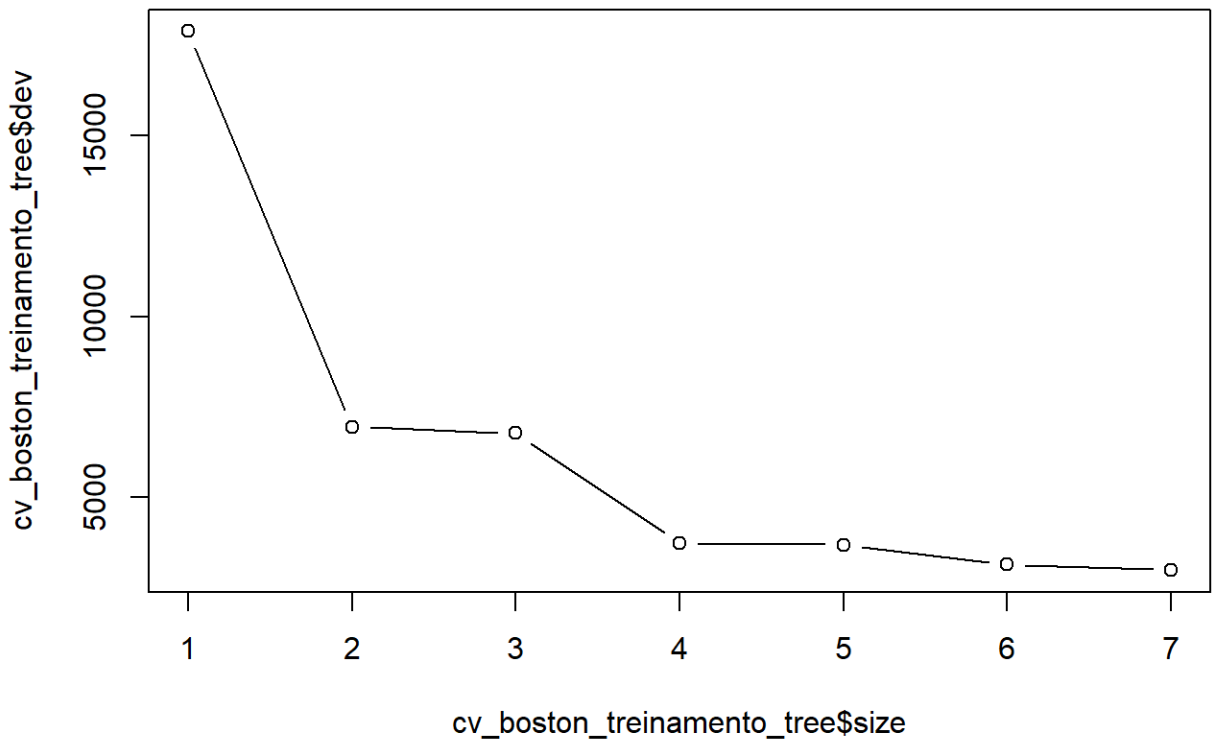
```
cv_boston_treinamento_tree <-  cv.tree(boston_treinamento_tree)
cv_boston_treinamento_tree
```

```
## $size
## [1] 7 6 5 4 3 2 1
##
## $dev
## [1]  2988.875  3157.296  3694.577  3730.507  6776.745
## [6]  6954.849 17876.599
##
## $k
## [1]       -Inf   238.0834   388.6755   410.8889  1724.6211
## [6]  1891.2023 10835.3231
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

```
summary(cv_boston_treinamento_tree)
```

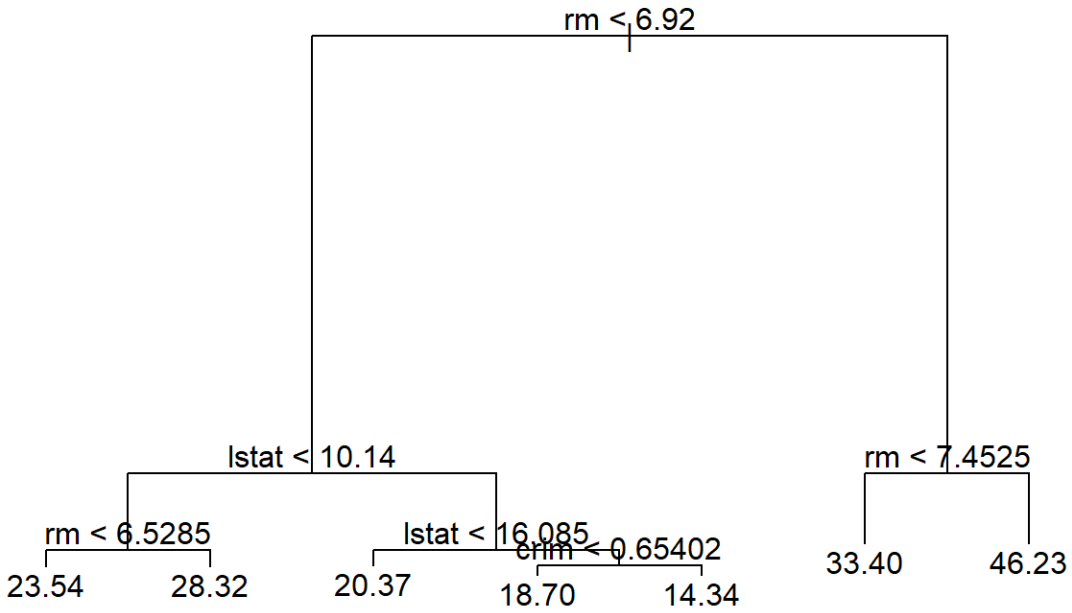
```
##        Length Class  Mode
## size   7      -none- numeric
## dev    7      -none- numeric
## k      7      -none- numeric
## method 1      -none- character
```

```
plot(cv_boston_treinamento_tree$size , cv_boston_treinamento_tree$dev, type = "b")
```



Uma árvore de 7 nódulos é selecionada por **cross-validation**. Vamos podar a árvore:

```
boston_prune <- prune.tree(boston_treinamento_tree , best = 7)
plot(boston_prune)
text(boston_prune , pretty = 0)
```

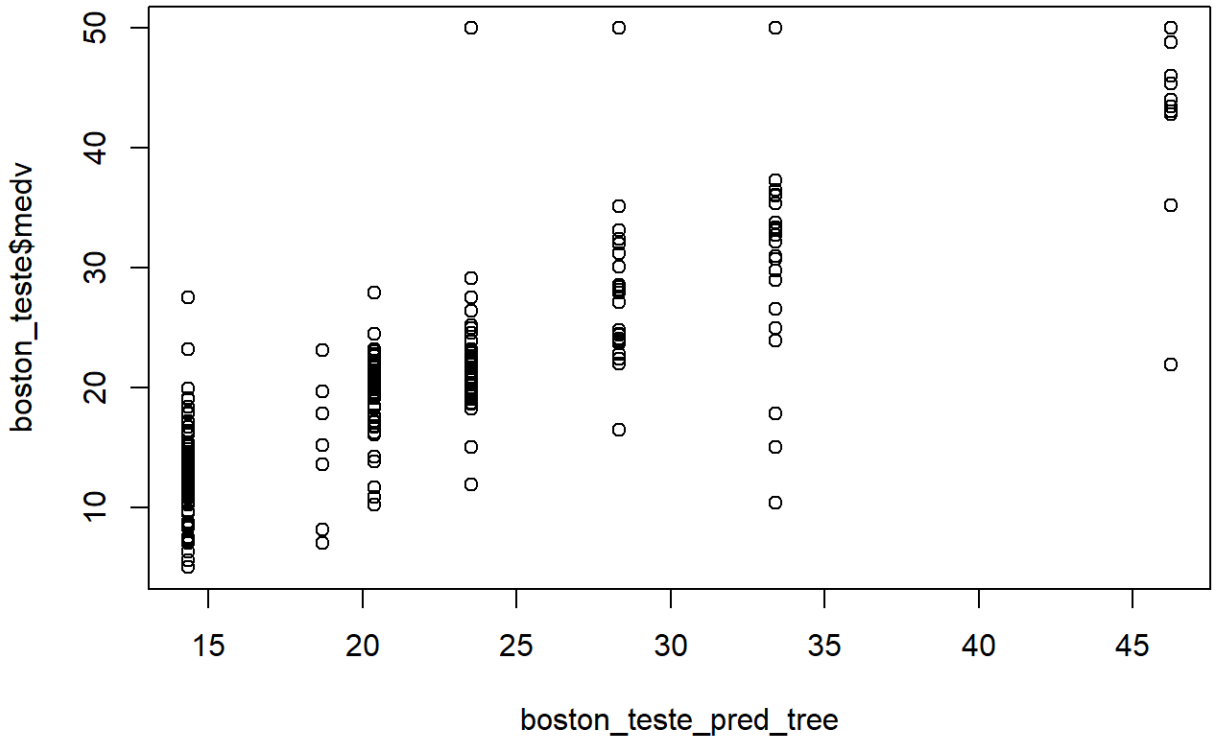


Por meio dde **cross-validation**, não parece ser necessário podar a árvore. A árvore não podada já leva a 7 nódulos e para exatamente a mesma árvore.

(b) Use a árvore não podada para fazer previsões para o conjunto teste. Calcule o EQM.

```
# previsões para o conjunto teste

boston_teste_pred_tree <- predict(boston_treinamento_tree, newdata = boston_teste)
plot(boston_teste_pred_tree , boston_teste$medv)
```




```
# Calcular o EQM
eqm_teste_tree <- mean((boston_teste$medv - boston_teste_pred_tree)^2)
eqm_teste_tree
```

```
## [1] 34.2754
```

```
sqrt(eqm_teste_tree)
```

```
## [1] 5.854519
```

Isto é, o erro quadratíco médio de teste associado a árvore de regressão é 34,2754. A raiz quadrada do EQM de teste é aproximadamente 5,854519. Isto é, o modelo leva a previsões de teste que são, em média, entre aproximadamente $5.854519, do valor verdadeiro de medv.

(c) Use bagging, florestas e boosting e comente sobre o melhor ajuste.

Estimando o bagging:

```
# Carregar bibliotecas
library(randomForest)
library(gbm)
set.seed(123)

# Bagging
boston_treinamento_bagging <- randomForest(
  medv ~ .,
  data = boston_treinamento,
  mtry = 12,
  importance = TRUE
  )

boston_treinamento_bagging
```

```
##
## Call:
##  randomForest(formula = medv ~ ., data = boston_treinamento, mtry = 12,      importance = TRUE)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 12
##
##           Mean of squared residuals: 7.753894
##                     % Var explained: 88.8
```

```
boston_teste_pred_bagging <- predict(boston_treinamento_bagging, newdata = boston_teste)

eqm_bagging <- mean((boston_teste$medv - boston_teste_pred_bagging)^2)
eqm_bagging
```

```
## [1] 33.57203
```

O erro quadrático médio de teste associado ao bagging é `r eqm_bagging`.

```
# Random Forests
boston_treinamento_rf <- randomForest(
  medv ~ .,
  data = boston_treinamento,
  mtry = 6,
  ntree = 25
  )

boston_treinamento_rf
```

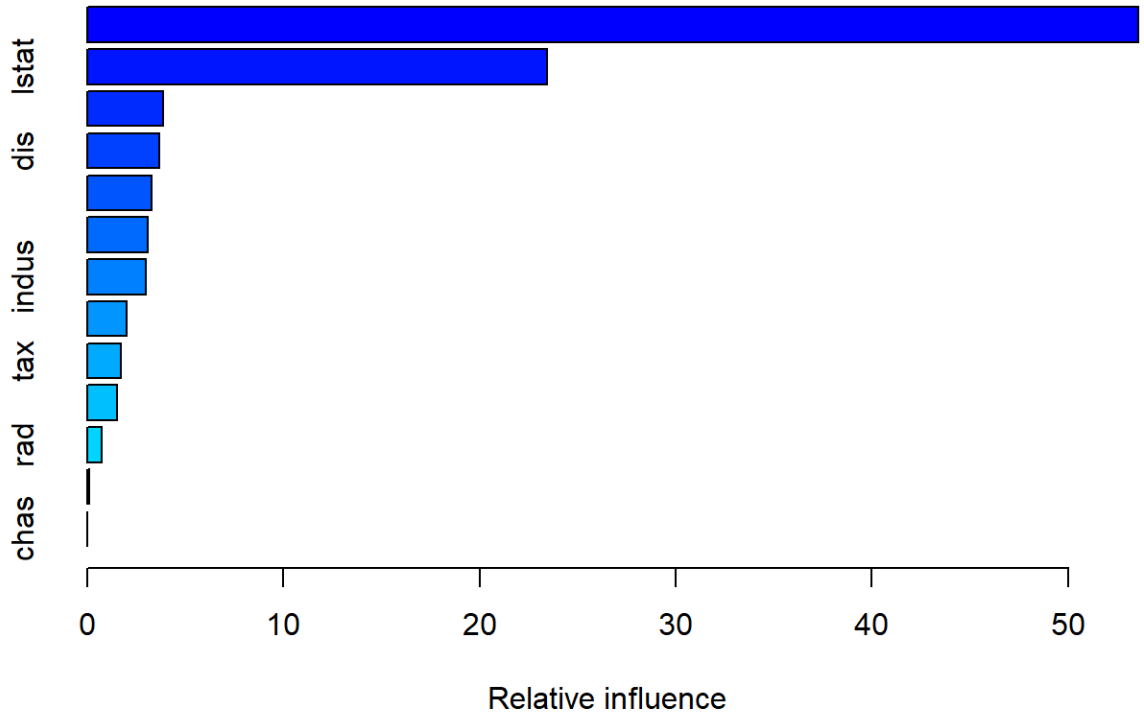
```
##
## Call:
##  randomForest(formula = medv ~ ., data = boston_treinamento, mtry = 6,      ntree = 25)
##                Type of random forest: regression
##                      Number of trees: 25
## No. of variables tried at each split: 6
##
##           Mean of squared residuals: 8.488808
##                     % Var explained: 87.74
```

```
boston_teste_pred_rf <- predict(boston_treinamento_rf, newdata = boston_teste)
eqm_rf <- mean((boston_teste$medv - boston_teste_pred_rf)^2)
eqm_rf
```

```
## [1] 34.51443
```

O erro quadrático de teste para florestas aleatórias é 34.5144311.

```
# Boosting
boston_treinamento_boosting <- gbm(
  medv ~ .,
  data = boston_treinamento,
  distribution = "gaussian",
  n.trees = 5000,
  interaction.depth = 4
  )
summary(boston_treinamento_boosting)
```




```
##             var     rel.inf
## rm           rm 53.58677871
## lstat     lstat 23.46451134
## crim       crim  3.85720354
## dis         dis  3.66612246
## age         age  3.27216775
## ptratio ptratio  3.08302311
## indus     indus  2.96776746
## black     black  1.99536794
## tax         tax  1.71215053
## nox         nox  1.53425772
## rad         rad  0.73301027
## zn           zn  0.10797449
## chas       chas  0.01966469
```

```
boston_teste_pred_boosting <- predict(
  boston_treinamento_boosting,
  newdata = boston_teste,
  n.trees = 5000)

eqm_boosting <- mean((boston_teste$medv - boston_teste_pred_boosting)^2)

eqm_boosting
```

```
## [1] 32.48648
```

O erro quadrático de teste para boosting é 32.4864792.

Usando como medida de comparação o erro quadrático de teste, o modelo com melhor ajuste é o boosting.

Questão 2 (4,0 pontos)

Considere o conjunto de dados `OJ` do pacote `ISLR`.

```
# carregando bibliotecas e dados
library(ISLR)

#carregando oj
data("OJ")
oj <- OJ
attach(oj)

# visualizando uma amostra de oj, com características da base
glimpse(oj)
```

```
## Rows: 1,070
## Columns: 18
## $ Purchase       <fct> CH, CH, CH, MM, CH, CH, CH, CH, CH,~
## $ WeekofPurchase <dbl> 237, 239, 245, 227, 228, 230, 232, ~
## $ StoreID        <dbl> 1, 1, 1, 1, 7, 7, 7, 7, 7, 7, 7, 7,~
## $ PriceCH        <dbl> 1.75, 1.75, 1.86, 1.69, 1.69, 1.69,~
## $ PriceMM        <dbl> 1.99, 1.99, 2.09, 1.69, 1.69, 1.99,~
## $ DiscCH         <dbl> 0.00, 0.00, 0.17, 0.00, 0.00, 0.00,~
## $ DiscMM         <dbl> 0.00, 0.30, 0.00, 0.00, 0.00, 0.00,~
## $ SpecialCH      <dbl> 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0,~
## $ SpecialMM      <dbl> 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0,~
## $ LoyalCH        <dbl> 0.500000, 0.600000, 0.680000, 0.400~
## $ SalePriceMM    <dbl> 1.99, 1.69, 2.09, 1.69, 1.69, 1.99,~
## $ SalePriceCH    <dbl> 1.75, 1.75, 1.69, 1.69, 1.69, 1.69,~
## $ PriceDiff      <dbl> 0.24, -0.06, 0.40, 0.00, 0.00, 0.30~
## $ Store7         <fct> No, No, No, No, Yes, Yes, Yes, Yes,~
## $ PctDiscMM      <dbl> 0.000000, 0.150754, 0.000000, 0.000~
## $ PctDiscCH      <dbl> 0.000000, 0.000000, 0.091398, 0.000~
## $ ListPriceDiff  <dbl> 0.24, 0.24, 0.23, 0.00, 0.00, 0.30,~
## $ STORE          <dbl> 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0,~
```

```
# algumas estatísticas básicas de oj
summary(oj)
```

```
##  Purchase WeekofPurchase     StoreID        PriceCH
##  CH:653   Min.   :227.0   Min.   :1.00   Min.   :1.690
##  MM:417   1st Qu.:240.0   1st Qu.:2.00   1st Qu.:1.790
##           Median :257.0   Median :3.00   Median :1.860
##           Mean   :254.4   Mean   :3.96   Mean   :1.867
##           3rd Qu.:268.0   3rd Qu.:7.00   3rd Qu.:1.990
##           Max.   :278.0   Max.   :7.00   Max.   :2.090
##     PriceMM          DiscCH            DiscMM
##  Min.   :1.690   Min.   :0.00000   Min.   :0.0000
##  1st Qu.:1.990   1st Qu.:0.00000   1st Qu.:0.0000
##  Median :2.090   Median :0.00000   Median :0.0000
##  Mean   :2.085   Mean   :0.05186   Mean   :0.1234
##  3rd Qu.:2.180   3rd Qu.:0.00000   3rd Qu.:0.2300
##  Max.   :2.290   Max.   :0.50000   Max.   :0.8000
##    SpecialCH        SpecialMM         LoyalCH
##  Min.   :0.0000   Min.   :0.0000   Min.   :0.000011
##  1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:0.325257
##  Median :0.0000   Median :0.0000   Median :0.600000
##  Mean   :0.1477   Mean   :0.1617   Mean   :0.565782
##  3rd Qu.:0.0000   3rd Qu.:0.0000   3rd Qu.:0.850873
##  Max.   :1.0000   Max.   :1.0000   Max.   :0.999947
##   SalePriceMM     SalePriceCH      PriceDiff
##  Min.   :1.190   Min.   :1.390   Min.   :-0.6700
##  1st Qu.:1.690   1st Qu.:1.750   1st Qu.: 0.0000
##  Median :2.090   Median :1.860   Median : 0.2300
##  Mean   :1.962   Mean   :1.816   Mean   : 0.1465
##  3rd Qu.:2.130   3rd Qu.:1.890   3rd Qu.: 0.3200
##  Max.   :2.290   Max.   :2.090   Max.   : 0.6400
##  Store7      PctDiscMM        PctDiscCH
##  No :714   Min.   :0.0000   Min.   :0.00000
##  Yes:356   1st Qu.:0.0000   1st Qu.:0.00000
##            Median :0.0000   Median :0.00000
##            Mean   :0.0593   Mean   :0.02731
##            3rd Qu.:0.1127   3rd Qu.:0.00000
##            Max.   :0.4020   Max.   :0.25269
##  ListPriceDiff       STORE
##  Min.   :0.000   Min.   :0.000
##  1st Qu.:0.140   1st Qu.:0.000
##  Median :0.240   Median :2.000
##  Mean   :0.218   Mean   :1.631
##  3rd Qu.:0.300   3rd Qu.:3.000
##  Max.   :0.440   Max.   :4.000
```

(a) Crie um conjunto de treinamento contendo uma amostra de 800 observações e um conjunto teste contendo as observações restantes.

```
# criando conjuntos de treinamento e de teste
treino <- sample (1: nrow (oj), 800)

oj_treino <- oj[treino,]

oj_teste <-  oj[-treino,]
```

(b) Ajuste um classificador SVM ao conjunto de treinamento usando `cost=0.01`, tendo `Purchase` como resposta e as outras variáveis como preditoras. Use a função `summary()` e descreva os resultados obtidos.

```
library(e1071)
# ajustando um svm no conjunto de treinamento
oj_treino_svm <- svm(Purchase ~ ., kernel = "linear", data = oj_treino, cost = 0.01)
summary(oj_treino_svm)
```

```
##
## Call:
## svm(formula = Purchase ~ ., data = oj_treino, kernel = "linear",
##     cost = 0.01)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.01
##
## Number of Support Vectors:  448
##
##  ( 226 222 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

O SVM desse modelo criou 436 *support vectors* dentro de 800 observações, nas quais 219 pertencem ao conjunto CH e 217 pertencem ao conjunto de MM.

(c) Quais são as taxas de erros de treinamento e de teste?

```
# calculando a taxa de erro de treinamento
oj_treino_svm_pred <- predict(oj_treino_svm, oj_treino)

oj_treino_svm_table <- table(oj_treino$Purchase, oj_treino_svm_pred)
oj_treino_svm_table
```

```
##     oj_treino_svm_pred
##       CH  MM
##   CH 440  55
##   MM  87 218
```

```
oj_treino_svm_taxa_erro <- (oj_treino_svm_table[2,1]+oj_treino_svm_table[1,2]) /
  (oj_treino_svm_table[1,1]+oj_treino_svm_table[2,1]+oj_treino_svm_table[1,2]+oj_treino_svm_table[2,2])
oj_treino_svm_taxa_erro
```

```
## [1] 0.1775
```

A taxa de erro de treino é 0.1775.

```
# calculando a taxa de erro de teste
oj_teste_svm_pred <- predict(oj_treino_svm, oj_teste)

oj_teste_svm_table <- table(oj_teste$Purchase, oj_teste_svm_pred)
oj_teste_svm_table
```

```
##     oj_teste_svm_pred
##       CH  MM
##   CH 146  12
##   MM  25  87
```

```
oj_teste_svm_taxa_erro <- (oj_teste_svm_table[2,1]+oj_teste_svm_table[1,2]) /
  (oj_teste_svm_table[1,1]+oj_teste_svm_table[2,1]+oj_teste_svm_table[1,2]+oj_teste_svm_table[2,2])
oj_teste_svm_taxa_erro
```

```
## [1] 0.137037
```

A taxa de erro de teste é 0.137037.

(d) Use a função `tune()` para selecionar um `cost` ótimo. Considere valores no intervalo 0.01 a 10.

```
oj_treino_svm_tune <- tune(
  svm,
  Purchase ~ .,
  kernel = "linear",
  data = oj_treino,
  ranges = list(
    cost = c(0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1, 2,  5, 10)
  )
)
summary(oj_treino_svm_tune)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##     5
##
## - best performance: 0.17375
##
## - Detailed performance results:
##     cost   error dispersion
## 1   0.01 0.18250 0.03343734
## 2   0.05 0.17750 0.03216710
## 3   0.10 0.17625 0.02913689
## 4   0.25 0.17625 0.02913689
## 5   0.50 0.18250 0.03291403
## 6   0.75 0.18125 0.03240906
## 7   1.00 0.17875 0.03438447
## 8   2.00 0.17625 0.02972676
## 9   5.00 0.17375 0.03251602
## 10 10.00 0.17625 0.03143004
```

```
oj_treino_svm_tune$best.parameters$cost
```

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

```
oj_treino_svm_tune$performances$error
```

```
##  [1] 0.18250 0.17750 0.17625 0.17625 0.18250 0.18125 0.17875
##  [8] 0.17625 0.17375 0.17625
```

O **cost** ótimo é 5.

(e) Calcule as taxas de erro para este novo valor de `cost`.

Estimando o modelo com novo *cost*:

```
# para calcular as taxas de erro para este novo valor de cost, vamos usar o parâmetro calculado no item anterior
oj_treino_svm_best_cost <- svm(
  Purchase ~ .,
  kernel = "linear",
  data = oj_treino,
  cost = oj_treino_svm_tune$best.parameters$cost
  )
```

```
oj_treino_svm_best_cost_pred <- predict(oj_treino_svm_best_cost, oj_treino)

oj_treino_svm_best_cost_table <- table(oj_treino$Purchase, oj_treino_svm_best_cost_pred)
oj_treino_svm_best_cost_table
```

```
##     oj_treino_svm_best_cost_pred
##       CH  MM
##   CH 439  56
##   MM  82 223
```

```
oj_treino_svm_best_cost_taxa_erro <- (oj_treino_svm_best_cost_table[2,1]+oj_treino_svm_best_cost_table[1,2]) /
  (oj_treino_svm_best_cost_table[1,1]+oj_treino_svm_best_cost_table[2,1]+oj_treino_svm_best_cost_table[1,2]+oj_treino_svm_be
st_cost_table[2,2])
oj_treino_svm_best_cost_taxa_erro
```

```
## [1] 0.1725
```

A taxa de erro de treino, para *cost* ótimo de 5, é 0.1725.

```
oj_teste_svm_best_cost_pred <- predict(oj_treino_svm_best_cost, oj_teste)

oj_teste_svm_best_cost_table <- table(oj_teste$Purchase, oj_teste_svm_best_cost_pred)
oj_teste_svm_best_cost_table
```

```
##     oj_teste_svm_best_cost_pred
##       CH  MM
##   CH 145  13
##   MM  24  88
```

```
oj_teste_svm_best_cost_taxa_erro <- (oj_teste_svm_best_cost_table[2,1]+oj_teste_svm_best_cost_table[1,2]) /
  (oj_teste_svm_best_cost_table[1,1]+oj_teste_svm_best_cost_table[2,1]+oj_teste_svm_best_cost_table[1,2]+oj_teste_svm_best_c
ost_table[2,2])
oj_teste_svm_best_cost_taxa_erro
```

```
## [1] 0.137037
```

A taxa de erro de teste, para *cost* ótimo de 5, é 0.137037.

(f) Repita (b)-(e) usando SVM com kernel radial. Use o valor default para `gamma`.

```
oj_treino_svm_radial <- svm(
  Purchase ~ .,
  kernel = "radial",
  data = oj_treino,
  cost = oj_treino_svm_tune$best.parameters$cost
  )

summary(oj_treino_svm_radial)
```

```
##
## Call:
## svm(formula = Purchase ~ ., data = oj_treino, kernel = "radial",
##     cost = oj_treino_svm_tune$best.parameters$cost)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  5
##
## Number of Support Vectors:  357
##
##  ( 184 173 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

O SVM desse modelo, com **kernel radial** criou 357 *support vectors* dentro de 800 observações, nas quais 184 pertencem ao conjunto CH e 173 pertencem ao conjunto de MM.

```
oj_treino_svm_radial_pred <- predict(oj_treino_svm_radial, oj_treino)

oj_treino_svm_radial_table <- table(oj_treino$Purchase, oj_treino_svm_radial_pred)
oj_treino_svm_radial_table
```

```
##     oj_treino_svm_radial_pred
##       CH  MM
##   CH 457  38
##   MM  84 221
```

```
oj_treino_svm_radial_taxa_erro <- (oj_treino_svm_radial_table[2,1]+oj_treino_svm_radial_table[1,2]) /
  (oj_treino_svm_radial_table[1,1]+oj_treino_svm_radial_table[2,1]+oj_treino_svm_radial_table[1,2]+oj_treino_svm_radial_tabl
e[2,2])
oj_treino_svm_radial_taxa_erro
```

```
## [1] 0.1525
```

A taxa de erro de treino, com a especificação de **kernel radial** é 0.1525.

```
oj_teste_svm_radial_pred <- predict(oj_treino_svm_radial, oj_teste)

oj_teste_svm_radial_table <- table(oj_teste$Purchase, oj_teste_svm_radial_pred)
oj_teste_svm_radial_table
```

```
##     oj_teste_svm_radial_pred
##       CH  MM
##   CH 146  12
##   MM  25  87
```

```
oj_teste_svm_radial_taxa_erro <- (oj_teste_svm_radial_table[2,1]+oj_teste_svm_radial_table[1,2]) /
  (oj_teste_svm_radial_table[1,1]+oj_teste_svm_radial_table[2,1]+oj_teste_svm_radial_table[1,2]+oj_teste_svm_radial_table[2,
2])
oj_teste_svm_radial_taxa_erro
```

```
## [1] 0.137037
```

A taxa de erro de teste, com a especificação de **kernel radial**, é 0.137037.

```
oj_treino_svm_radial_tune <- tune(
  svm,
  Purchase ~ .,
  kernel = "linear",
  data = oj_treino,
  ranges = list(
    cost = c(0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1, 2,  5, 10)
  )
)
summary(oj_treino_svm_radial_tune)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##  0.25
##
## - best performance: 0.17625
##
## - Detailed performance results:
##     cost   error dispersion
## 1   0.01 0.18500 0.03574602
## 2   0.05 0.18000 0.03496029
## 3   0.10 0.17750 0.03374743
## 4   0.25 0.17625 0.02972676
## 5   0.50 0.17750 0.03322900
## 6   0.75 0.17750 0.03322900
## 7   1.00 0.17750 0.03476109
## 8   2.00 0.17625 0.03701070
## 9   5.00 0.18000 0.03736085
## 10 10.00 0.18000 0.03689324
```

```
oj_treino_svm_radial_tune$best.parameters$cost
```

```
## [1] 0.25
```

```
oj_treino_svm_radial_tune$performances$error
```

```
##  [1] 0.18500 0.18000 0.17750 0.17625 0.17750 0.17750 0.17750
##  [8] 0.17625 0.18000 0.18000
```

O **cost** ótimo com a especificação de **kernel radial** é 0.25.

Estimando o modelo com novo *cost*:

```
oj_treino_svm_radial_best_cost <- svm(
  Purchase ~ .,
  kernel = "linear",
  data = oj_treino,
  cost = oj_treino_svm_radial_tune$best.parameters$cost
  )

  oj_treino_svm_radial_best_cost_pred <- predict(oj_treino_svm_radial_best_cost, oj_treino)

oj_treino_svm_radial_best_cost_table <- table(oj_treino$Purchase, oj_treino_svm_radial_best_cost_pred)
oj_treino_svm_radial_best_cost_table
```

```
##     oj_treino_svm_radial_best_cost_pred
##       CH  MM
##   CH 437  58
##   MM  82 223
```

```
oj_treino_svm_radial_best_cost_taxa_erro <- (oj_treino_svm_radial_best_cost_table[2,1]+oj_treino_svm_radial_best_cost_table
[1,2]) /
  (oj_treino_svm_radial_best_cost_table[1,1]+oj_treino_svm_radial_best_cost_table[2,1]+oj_treino_svm_radial_best_cost_table
[1,2]+oj_treino_svm_radial_best_cost_table[2,2])
oj_treino_svm_radial_best_cost_taxa_erro
```

```
## [1] 0.175
```

A taxa de erro de treino, para *cost* ótimo de 0.25, é 0.175.

```
oj_teste_svm_radial_best_cost_pred <- predict(oj_treino_svm_radial_best_cost, oj_teste)

oj_teste_svm_radial_best_cost_table <- table(oj_teste$Purchase, oj_teste_svm_radial_best_cost_pred)
oj_teste_svm_radial_best_cost_table
```

```
##     oj_teste_svm_radial_best_cost_pred
##       CH  MM
##   CH 143  15
##   MM  23  89
```

```
oj_teste_svm_radial_best_cost_taxa_erro <- (oj_teste_svm_radial_best_cost_table[2,1]+oj_teste_svm_radial_best_cost_table[1,
2]) /
  (oj_teste_svm_radial_best_cost_table[1,1]+oj_teste_svm_radial_best_cost_table[2,1]+oj_teste_svm_radial_best_cost_table[1,
2]+oj_teste_svm_radial_best_cost_table[2,2])
oj_teste_svm_radial_best_cost_taxa_erro
```

```
## [1] 0.1407407
```

A taxa de erro de teste, para *cost* ótimo de 0.25, é 0.175.

(g) Repita (b)-(e) com um kernel polinomial com `degree=2`.

```
oj_treino_svm_poly_2 <- svm(
  Purchase ~ .,
  kernel = "poly",
  degree = 2,
  data = oj_treino,
  cost = oj_treino_svm_tune$best.parameters$cost)

summary(oj_treino_svm_poly_2)
```

```
##
## Call:
## svm(formula = Purchase ~ ., data = oj_treino, kernel = "poly",
##     degree = 2, cost = oj_treino_svm_tune$best.parameters$cost)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  polynomial
##        cost:  5
##      degree:  2
##      coef.0:  0
##
## Number of Support Vectors:  383
##
##  ( 199 184 )
##
##
## Number of Classes:  2
##
## Levels:
##  CH MM
```

O SVM desse modelo com a especificação de **kernel polinomial com `degree=2** criou 383 *support vectors* dentro de 800 observações, nas quais 199 pertencem ao conjunto CH e 184 pertencem ao conjunto de MM.

```
oj_treino_svm_poly_2_pred <- predict(oj_treino_svm_poly_2, oj_treino)

oj_treino_svm_poly_2_table <- table(oj_treino$Purchase, oj_treino_svm_poly_2_pred)
oj_treino_svm_poly_2_table
```

```
##     oj_treino_svm_poly_2_pred
##       CH  MM
##   CH 457  38
##   MM  90 215
```

```
oj_treino_svm_poly_2_taxa_erro <- (oj_treino_svm_poly_2_table[2,1]+oj_treino_svm_poly_2_table[1,2]) /
  (oj_treino_svm_poly_2_table[1,1]+oj_treino_svm_poly_2_table[2,1]+oj_treino_svm_poly_2_table[1,2]+oj_treino_svm_poly_2_tabl
e[2,2])
oj_treino_svm_poly_2_taxa_erro
```

```
## [1] 0.16
```

A taxa de erro de treino com a especificação de **kernel polinomial com `degree=2** é 0.16.

```
oj_teste_svm_poly_2_pred <- predict(oj_treino_svm_poly_2, oj_teste)

oj_teste_svm_poly_2_table <- table(oj_teste$Purchase, oj_teste_svm_poly_2_pred)
oj_teste_svm_poly_2_table
```

```
##     oj_teste_svm_poly_2_pred
##       CH  MM
##   CH 147  11
##   MM  28  84
```

```
oj_teste_svm_poly_2_taxa_erro <- (oj_teste_svm_poly_2_table[2,1]+oj_teste_svm_poly_2_table[1,2]) /
  (oj_teste_svm_poly_2_table[1,1]+oj_teste_svm_poly_2_table[2,1]+oj_teste_svm_poly_2_table[1,2]+oj_teste_svm_poly_2_table[2,
2])
oj_teste_svm_poly_2_taxa_erro
```

```
## [1] 0.1444444
```

A taxa de erro de teste com a especificação de **kernel polinomial com `degree=2** é 0.1444444.

```
oj_treino_svm_poly_2_tune <- tune(
  svm,
  Purchase ~ .,
  kernel = "linear",
  data = oj_treino,
  ranges = list(
    cost = c(0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1, 2,  5, 10)
  )
)
summary(oj_treino_svm_poly_2_tune)
```

```
##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
##
## - best parameters:
##  cost
##  0.25
##
## - best performance: 0.1825
##
## - Detailed performance results:
##     cost   error dispersion
## 1   0.01 0.19000 0.03162278
## 2   0.05 0.18750 0.03280837
## 3   0.10 0.18375 0.03729108
## 4   0.25 0.18250 0.03782269
## 5   0.50 0.18375 0.03821086
## 6   0.75 0.18500 0.03763863
## 7   1.00 0.18375 0.03866254
## 8   2.00 0.18500 0.03855011
## 9   5.00 0.18625 0.03793727
## 10 10.00 0.18750 0.03773077
```

```
oj_treino_svm_poly_2_tune$best.parameters$cost
```

```
## [1] 0.25
```

```
oj_treino_svm_poly_2_tune$performances$error
```

```
##  [1] 0.19000 0.18750 0.18375 0.18250 0.18375 0.18500 0.18375
##  [8] 0.18500 0.18625 0.18750
```

O **cost** ótimo com a especificação de **kernel polinomial com `degree=2** é 0.25.

Estimando o modelo com novo *cost*:

```
oj_treino_svm_poly_2_best_cost <- svm(
  Purchase ~ .,
  kernel = "linear",
  data = oj_treino,
  cost = oj_treino_svm_poly_2_tune$best.parameters$cost
  )
```

```
oj_treino_svm_poly_2_best_cost_pred <- predict(oj_treino_svm_poly_2_best_cost, oj_treino)

oj_treino_svm_poly_2_best_cost_table <- table(oj_treino$Purchase, oj_treino_svm_poly_2_best_cost_pred)
oj_treino_svm_poly_2_best_cost_table
```

```
##     oj_treino_svm_poly_2_best_cost_pred
##       CH  MM
##   CH 437  58
##   MM  82 223
```

```
oj_treino_svm_poly_2_best_cost_taxa_erro <- (oj_treino_svm_poly_2_best_cost_table[2,1]+oj_treino_svm_poly_2_best_cost_table
[1,2]) /
  (oj_treino_svm_poly_2_best_cost_table[1,1]+oj_treino_svm_poly_2_best_cost_table[2,1]+oj_treino_svm_poly_2_best_cost_table
[1,2]+oj_treino_svm_poly_2_best_cost_table[2,2])
oj_treino_svm_poly_2_best_cost_taxa_erro
```

```
## [1] 0.175
```

A taxa de erro de treino com a especificação de **kernel polinomial com `degree=2**, para *cost* ótimo de 0.25, é 0.175.

```
oj_teste_svm_poly_2_best_cost_pred <- predict(oj_treino_svm_poly_2_best_cost, oj_teste)

oj_teste_svm_poly_2_best_cost_table <- table(oj_teste$Purchase, oj_teste_svm_poly_2_best_cost_pred)
oj_teste_svm_poly_2_best_cost_table
```

```
##     oj_teste_svm_poly_2_best_cost_pred
##       CH  MM
##   CH 143  15
##   MM  23  89
```

```
oj_teste_svm_poly_2_best_cost_taxa_erro <- (oj_teste_svm_poly_2_best_cost_table[2,1]+oj_teste_svm_poly_2_best_cost_table[1,
2]) /
  (oj_teste_svm_poly_2_best_cost_table[1,1]+oj_teste_svm_poly_2_best_cost_table[2,1]+oj_teste_svm_poly_2_best_cost_table[1,
2]+oj_teste_svm_poly_2_best_cost_table[2,2])
oj_teste_svm_poly_2_best_cost_taxa_erro
```

```
## [1] 0.1407407
```

A taxa de erro de teste com a especificação de **kernel polinomial com `degree=2**, para *cost* ótimo de 0.25, é 0.1407407.

(h) Qual procedimento parece dar os melhores resultados para esses dados?

A taxa de erro de teste com **kernel linear** e com **cost** ótimo é 0.137037.

A taxa de erro de teste com **kernel radial** e com **cost** ótimo é 0.1407407.

A taxa de erro de teste com **kernel polinomial com `degree=2** e com **cost** ótimo é 0.1407407.

O procedimento que parece dar os melhroes resultados para esses dados é o SVM com **kernel linear**.

Questão 3 (3,0 pontos)

Considere o conjunto de dados `food-texture` , que pode ser encontrado em openmv.net/info/food-texture (openmv.net/info/food-texture). Os dados estão no formato CSV. Leia com atenção o significado de cada variável.

```
# carregando bibliotecas e dados
library(corrplot)
library(psych)


food_texture <-  read.csv(file='food-texture.csv')
head(food_texture)
```

```
##      X  Oil Density Crispy Fracture Hardness
## 1 B110 16.5    2955     10       23       97
## 2 B136 17.7    2660     14        9      139
## 3 B171 16.2    2870     12       17      143
## 4 B192 16.7    2920     10       31       95
## 5 B225 16.3    2975     11       26      143
## 6 B237 19.1    2790     13       16      189
```

```
glimpse(food_texture)
```

```
## Rows: 50
## Columns: 6
## $ X        <chr> "B110", "B136", "B171", "B192", "B225", "~
## $ Oil      <dbl> 16.5, 17.7, 16.2, 16.7, 16.3, 19.1, 18.4,~
## $ Density  <int> 2955, 2660, 2870, 2920, 2975, 2790, 2750,~
## $ Crispy   <int> 10, 14, 12, 10, 11, 13, 13, 10, 11, 11, 1~
## $ Fracture <int> 23, 9, 17, 31, 26, 16, 17, 26, 23, 24, 15~
## $ Hardness <int> 97, 139, 143, 95, 143, 189, 114, 63, 123,~
```

```
summary(food_texture)
```

```
##       X                  Oil          Density     
##  Length:50          Min.   :13.7   Min.   :2570  
##  Class :character   1st Qu.:16.3   1st Qu.:2772  
##  Mode  :character   Median :16.9   Median :2868  
##                     Mean   :17.2   Mean   :2858  
##                     3rd Qu.:18.1   3rd Qu.:2945  
##                     Max.   :21.2   Max.   :3125  
##      Crispy         Fracture        Hardness    
##  Min.   : 7.00   Min.   : 9.00   Min.   : 63.0  
##  1st Qu.:10.00   1st Qu.:17.00   1st Qu.:107.2  
##  Median :12.00   Median :21.00   Median :126.0  
##  Mean   :11.52   Mean   :20.86   Mean   :128.2  
##  3rd Qu.:13.00   3rd Qu.:25.00   3rd Qu.:143.8  
##  Max.   :15.00   Max.   :33.00   Max.   :192.0
```

A base de dados simulada **food_texture** contém medidas de comidas de confeitaria.

Oil: porcentagem de óleo na massa;

Density: a densidade do produto (quanto maior o número, mais denso o produto);

Crispy: uma medida de crocância, em uma escala de 7 a 15, sendo 15 mais crocante;

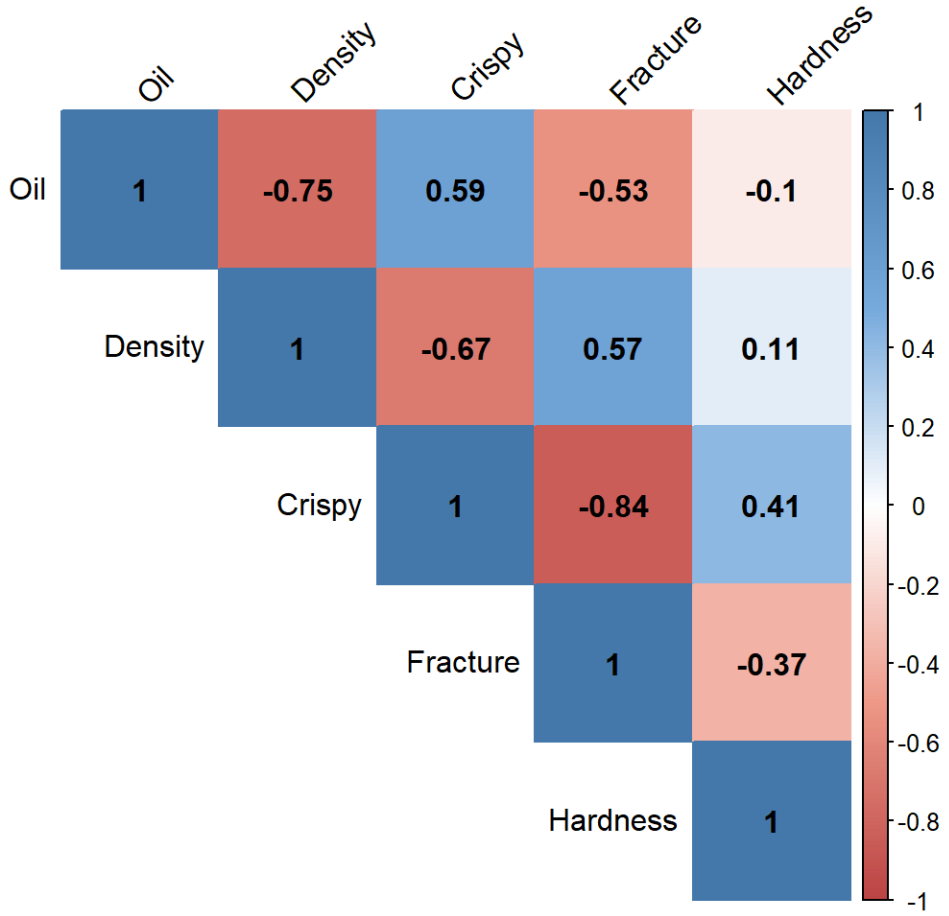
Fracture: o ângulo, em graus, pelo qual a massa pode ser dobrada lentamente antes de se romper;

Hardness: uma ponta afiada é usada para medir a quantidade de força necessária antes que ocorra a quebra.

(a) Faça uma análise de componentes principais (ACP). Escreva cada CP como função das variáveis originais. Tente interpretar cada componente que você vai reter. Faça os gráficos apropriados.

Inicialmente, vamos analisar a correlação entre as variáveis do modelo:

```
# criando a correlação entre as variáveis
correlacao <- cor(food_texture[,2:6], method = "pearson")
# paleta de cores pasteis para usar no gráfico de correlação
col <- colorRampPalette(c("#BB4444", "#EE9988", "#FFFFFF", "#77AADD", "#4477AA"))
# produzindo um gráfico para visualização
corrplot(correlacao, method = "color",
type = "upper", col = col(200),
addCoef.col = "black",
tl.col="black", tl.srt=45)
```

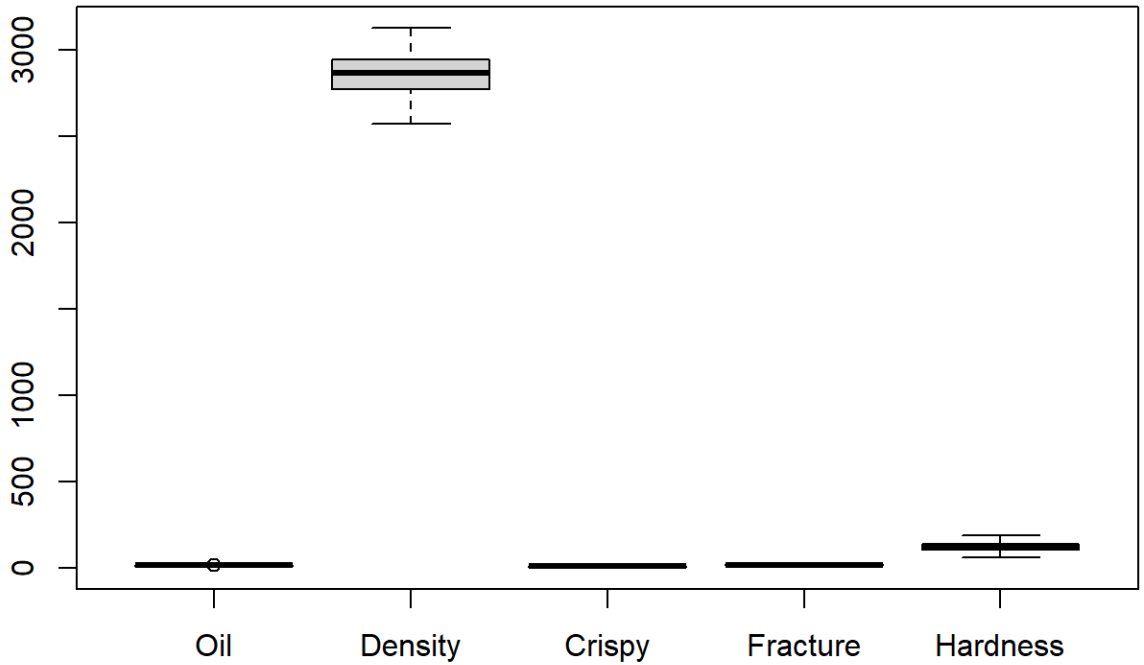


No correlograma, destacam-se os seguintes pontos:

- **Crispy** e **Fracture** são altamente correlacionados negativamente (-0.84);
- **Hardness** tem pouca correlação com as outras variáveis (menor que 0.5, em termos absolutos, com todas as outras), e, em especial, com **Oil** (-0.1);
- **Oil** e **Density** são altamente correlacionados negativamente (-0.75).

É preciso padronizar esses dados? Vamos avaliar com um *bloxplot*

```
boxplot(food_texture[,2:6])
```



Como a distribuições de valores das variáveis tem disperções distintas, vamos padronizar cada variável para facilitar a interpretação dos coeficientes.

Vamos prosseguir com a análise de componentes principais:

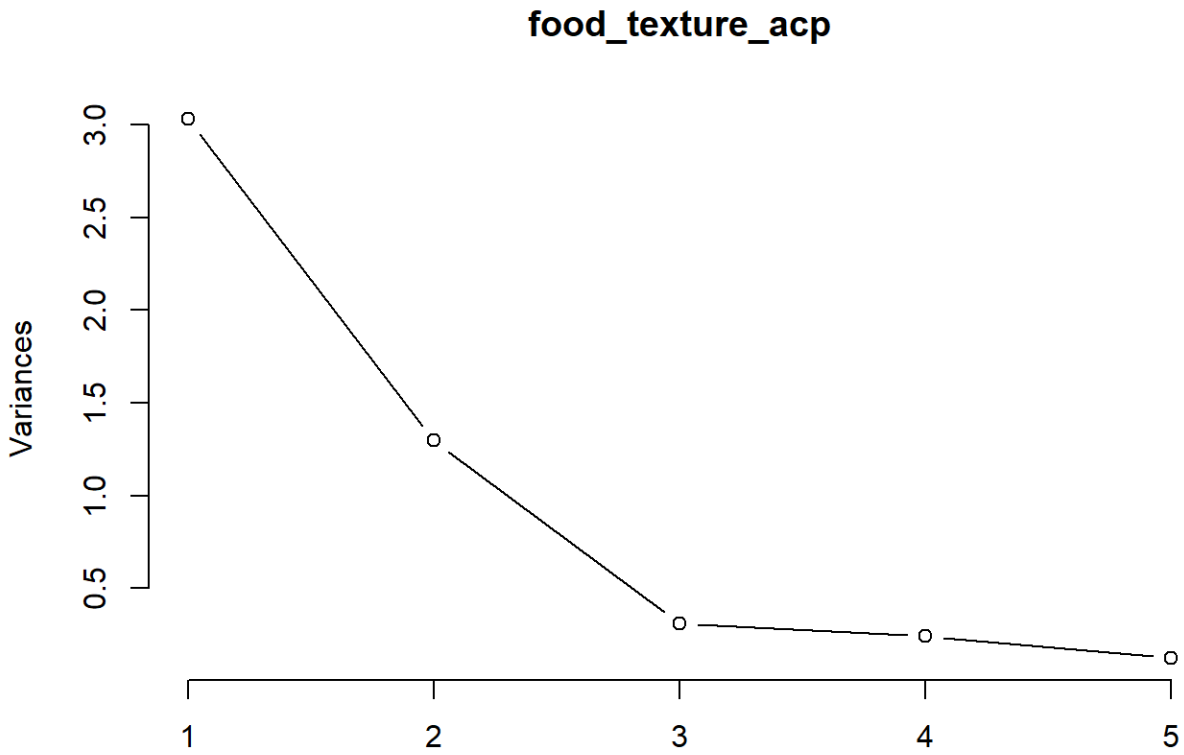
```
food_texture_acp <- prcomp(
  food_texture[,2:6],
  center = TRUE,
  scale. = TRUE)
food_texture_acp
```

```
## Standard deviations (1, .., p=5):
## [1] 1.7410380 1.1382907 0.5568207 0.4918537 0.3480110
##
## Rotation (n x k) = (5 x 5):
##                 PC1        PC2         PC3        PC4
## Oil       0.4575334 -0.3704389  0.65903020 -0.4679449
## Density  -0.4787455  0.3567500  0.01623973 -0.7184632
## Crispy    0.5323877  0.1976610 -0.17888443  0.1325269
## Fracture -0.5044769 -0.2212399  0.54227938  0.4569317
## Hardness  0.1534026  0.8046661  0.48923298  0.1961843
##                  PC5
## Oil       0.01204121
## Density   0.35648161
## Crispy    0.79242064
## Fracture  0.44011646
## Hardness -0.22614798
```

```
summary(food_texture_acp)
```

```
## Importance of components:
##                            PC1    PC2     PC3     PC4
## Standard deviation     1.7410 1.1383 0.55682 0.49185
## Proportion of Variance 0.6062 0.2591 0.06201 0.04838
## Cumulative Proportion  0.6062 0.8654 0.92739 0.97578
##                            PC5
## Standard deviation     0.34801
## Proportion of Variance 0.02422
## Cumulative Proportion  1.00000
```

```
screeplot(food_texture_acp, type = "lines")
```

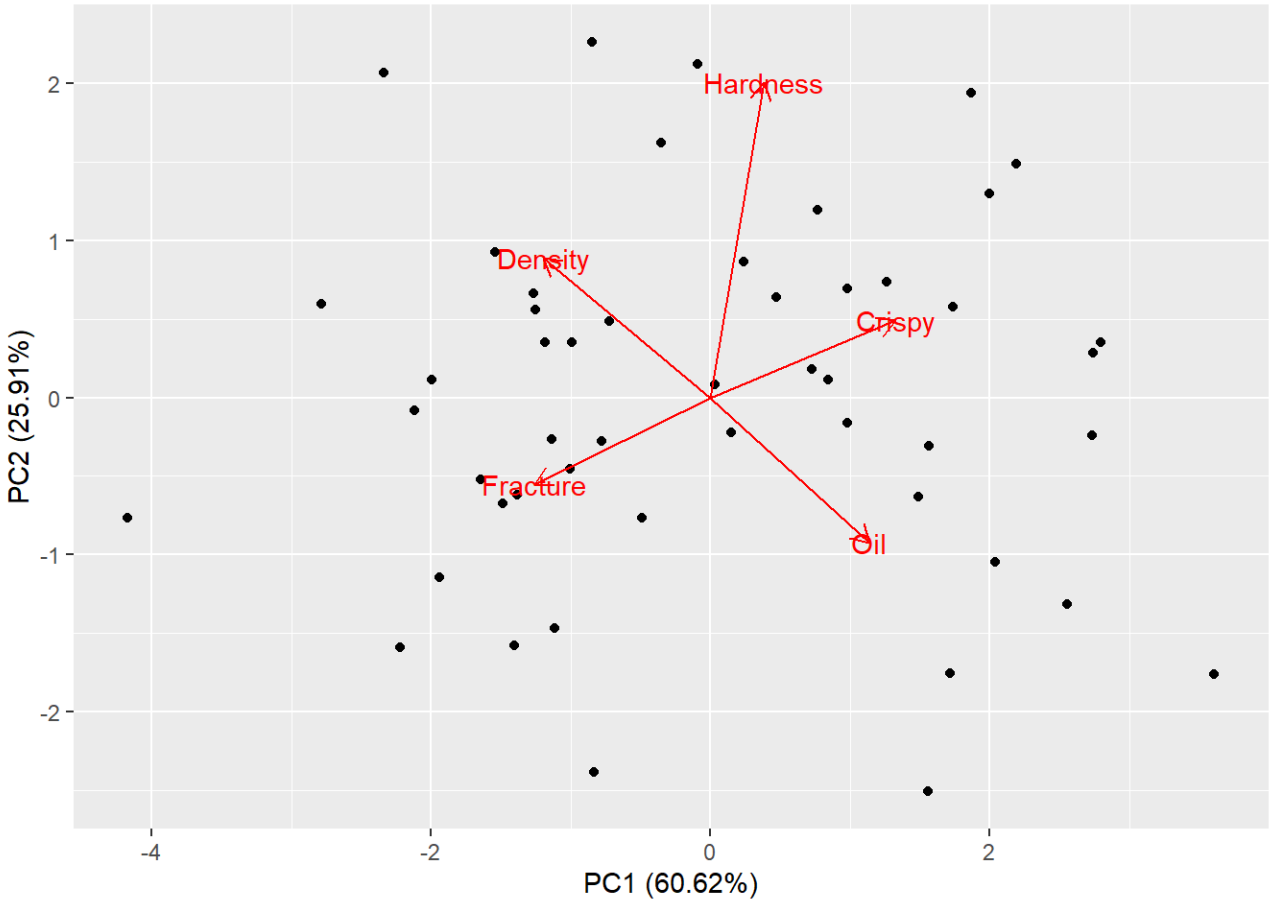


Os resultados mostram 5 componentes principais. Algumas características desses componentes podem ser destacados, como o peso de **Hardness** em PC2.

Analisando a variância explicada por cada um dos componentes, temos que o primeiro componente explica cerca de 60% da variância dsa variáveis em análise; já o segundo explica cerca de 26%. Conjuntamente, os dois explicam cerca de 86% da variância dos dados. Junto com o screeplot, temos que os dois primeiros componentes são suficientes para o modelo.

No gráfico abaixo, temos o vetores de cada variável relevante dentro do espaço dos componentes. Podemos ver que **Hardness**

```
library(ggfortify)
# gráfico com os autovetores e os componentes principais.
autoplot(food_texture_acp, data = food_texture,
loadings = TRUE, loadings.label = TRUE,
scale = 0
)
```



Interpretando os componentes:

1. O primeiro componente possui cargas positivas para **Oil** e para **Cripsy** e negativas para **Density** e **Fracture**. Assim, podemos interpretar o primeiro componente como um componente de crocância e leveza.
2. O segundo componente possui uma carga positiva com alto valor para **Hardness**, e uma carga positiva para **Density**. Aqui, então, temos que o segundo componente pode ser interpretado como um componente de robustez e dureza.

(b) Faça uma análise fatorial (AF) com dois fatores. Para isso, considere AF em três situações: sem rotação, com rotação varimax e com rotação promax.

Análise fatorial com dois fatores sem rotação:

```
food_texture_af_sem_rotacao <- fa(
 food_texture[,2:6],
 nfactors=2,
 rotate="none"
)


summary(food_texture_af_sem_rotacao)
```

```
##
## Factor analysis with Call: fa(r = food_texture[, 2:6], nfactors = 2, rotate = "none")
##
## Test of the hypothesis that 2 factors are sufficient.
## The degrees of freedom for the model is 1  and the objective function was  0.01
## The number of observations was  50  with Chi Square =  0.32  with prob <  0.57
##
## The root mean square of the residuals (RMSA) is  0
## The df corrected root mean square of the residuals is  0.02
##
## Tucker Lewis Index of factoring reliability =  1.048
## RMSEA index =  0  and the 10 % confidence intervals are  0 0.312
## BIC =  -3.59
```

```
food_texture_af_sem_rotacao$loadings
```

```
##
## Loadings:
##          MR1    MR2
## Oil       0.735 -0.371
## Density  -0.813  0.413
## Crispy    0.943  0.249
## Fracture -0.833 -0.234
## Hardness  0.241  0.737
##
##                  MR1   MR2
## SS loadings    2.842 0.967
## Proportion Var 0.568 0.193
## Cumulative Var 0.568 0.762
```

```
food_texture_af_sem_rotacao$communality
```

```
##       Oil   Density    Crispy  Fracture  Hardness
## 0.6770224 0.8309053 0.9521949 0.7482314 0.6010060
```

Análise fatorial com dois fatores, com rotação Varimax

```
food_texture_af_varimax <- fa(
 food_texture[,2:6],
 nfactors=2,
 rotate="varimax"
)

summary(food_texture_af_varimax)
```

```
##
## Factor analysis with Call: fa(r = food_texture[, 2:6], nfactors = 2, rotate = "varimax")
##
## Test of the hypothesis that 2 factors are sufficient.
## The degrees of freedom for the model is 1  and the objective function was  0.01
## The number of observations was  50  with Chi Square =  0.32  with prob <  0.57
##
## The root mean square of the residuals (RMSA) is  0
## The df corrected root mean square of the residuals is  0.02
##
## Tucker Lewis Index of factoring reliability =  1.048
## RMSEA index =  0  and the 10 % confidence intervals are  0 0.312
## BIC =  -3.59
```

```
food_texture_af_varimax$loadings
```

```
##
## Loadings:
##          MR1    MR2
## Oil      -0.822
## Density   0.911
## Crispy   -0.748  0.627
## Fracture  0.654 -0.566
## Hardness         0.769
##
##                  MR1   MR2
## SS loadings    2.502 1.307
## Proportion Var 0.500 0.261
## Cumulative Var 0.500 0.762
```

```
food_texture_af_varimax$communality
```

```
##       Oil   Density    Crispy  Fracture  Hardness
## 0.6770224 0.8309053 0.9521949 0.7482314 0.6010060
```

Análise fatorial com dois fatores, com rotação Promax

```
food_texture_af_promax <- fa(
 food_texture[,2:6],
 nfactors=2,
 rotate="promax"
)
summary(food_texture_af_promax)
```

```
##
## Factor analysis with Call: fa(r = food_texture[, 2:6], nfactors = 2, rotate = "promax")
##
## Test of the hypothesis that 2 factors are sufficient.
## The degrees of freedom for the model is 1  and the objective function was  0.01
## The number of observations was  50  with Chi Square =  0.32  with prob <  0.57
##
## The root mean square of the residuals (RMSA) is  0
## The df corrected root mean square of the residuals is  0.02
##
## Tucker Lewis Index of factoring reliability =  1.048
## RMSEA index =  0  and the 10 % confidence intervals are  0 0.312
## BIC =  -3.59
##  With factor correlations of
##       MR1   MR2
## MR1  1.00 -0.37
## MR2 -0.37  1.00
```

```
food_texture_af_promax$loadings
```

```
##
## Loadings:
##          MR1    MR2
## Oil      -0.873 -0.187
## Density   0.968  0.210
## Crispy   -0.662  0.514
## Fracture  0.575 -0.469
## Hardness  0.258  0.832
##
##                  MR1   MR2
## SS loadings    2.533 1.256
## Proportion Var 0.507 0.251
## Cumulative Var 0.507 0.758
```

```
food_texture_af_promax$communality
```

```
##       Oil   Density    Crispy  Fracture  Hardness
## 0.6770224 0.8309053 0.9521949 0.7482314 0.6010060
```

```
# food_texture_af_promax$scores
```

(c) Faça os gráficos apropriados e comente sobre qual rotação é mais apropriada para melhor interpretar os fatores.

```
# carregando bibliotecas de plots
library(ggplot2)
library(ggrepel)
```

Gráfico de AF sem rotação:

```
# Convertenedo a tabela scores em um data frame
scores_df <- as.data.frame(food_texture_af_sem_rotacao$scores)
# Convertenedo a tabela scores em um data frame

loadings_df <-  data.frame(matrix(
  as.numeric(food_texture_af_sem_rotacao$l),
  attributes(food_texture_af_sem_rotacao$l)$dim,
  dimnames=attributes(food_texture_af_sem_rotacao$l)$dimnames))


# Definir o tema do gráfico
theme_set(theme_minimal())

# Criar o gráfico de dispersão
food_texture_af_sem_rotacao_plot <- ggplot(scores_df, aes(x = MR1, y = MR2)) +
  geom_point() +
  geom_segment(
    aes(x = 0, y = 0, xend = MR1, yend = MR2),
    data = loadings_df,
    arrow = arrow(length = unit(0.3, "cm")),
    color = "blue") +
  geom_label_repel(aes(label = rownames(loadings_df)),
                   data = loadings_df, color = "blue")
food_texture_af_sem_rotacao_plot
```

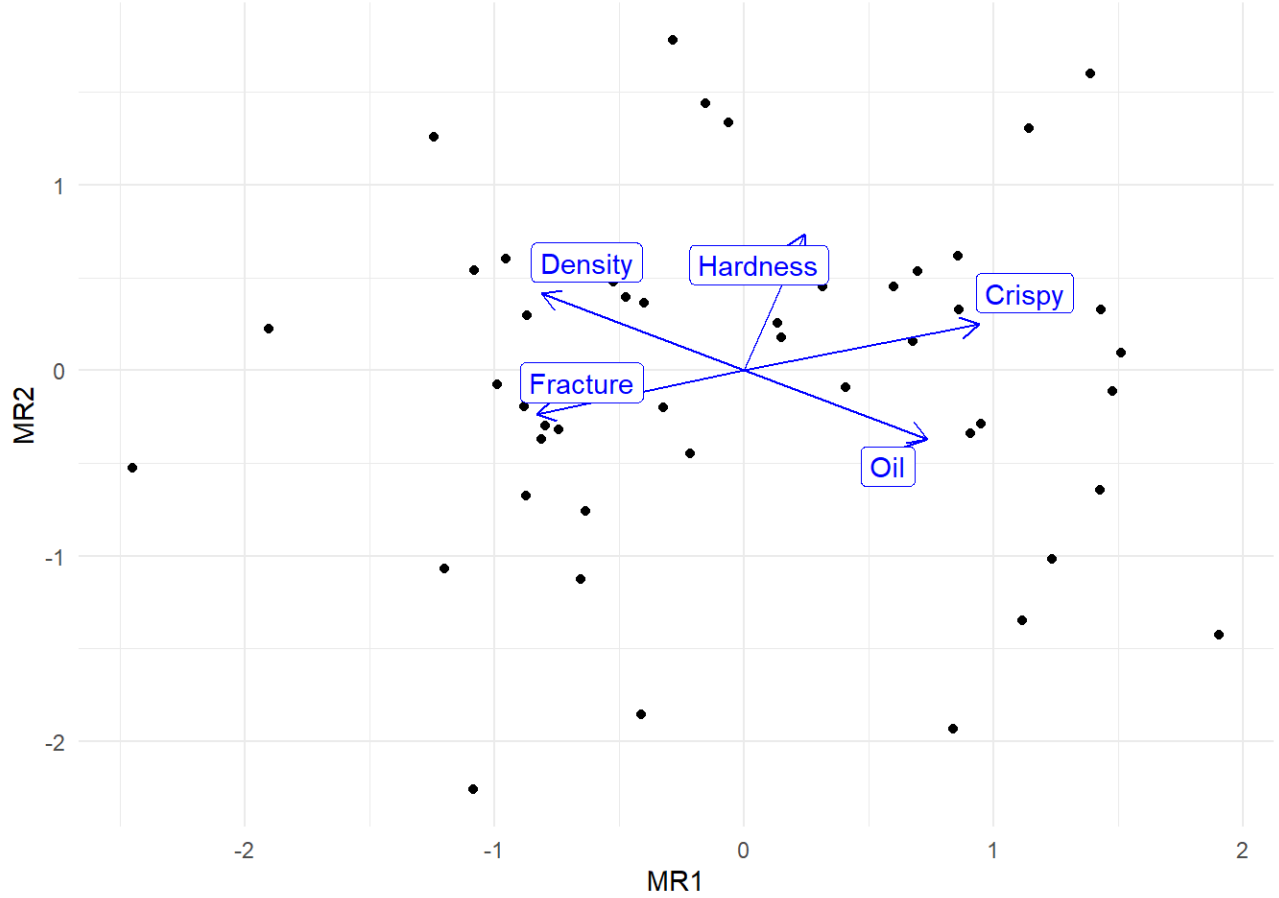


Gráfico de AF com rotação **Varimax**:

```r
# Convertenedo a tabela scores em um data frame
scores_df <- as.data.frame(food_texture_af_varimax$scores)
# Convertenedo a tabela scores em um data frame

loadings_df <-  data.frame(matrix(
  as.numeric(food_texture_af_varimax$l),
  attributes(food_texture_af_varimax$l)$dim,
  dimnames=attributes(food_texture_af_varimax$l)$dimnames))


# Definir o tema do gráfico
theme_set(theme_minimal())

# Criar o gráfico de dispersão
food_texture_af_varimax_plot <- ggplot(scores_df, aes(x = MR1, y = MR2)) +
  geom_point() +
  geom_segment(
    aes(x = 0, y = 0, xend = MR1, yend = MR2),
    data = loadings_df,
    arrow = arrow(length = unit(0.3, "cm")),
    color = "red") +
  geom_label_repel(aes(label = rownames(loadings_df)),
                   data = loadings_df, color = "red")
food_texture_af_varimax_plot
```

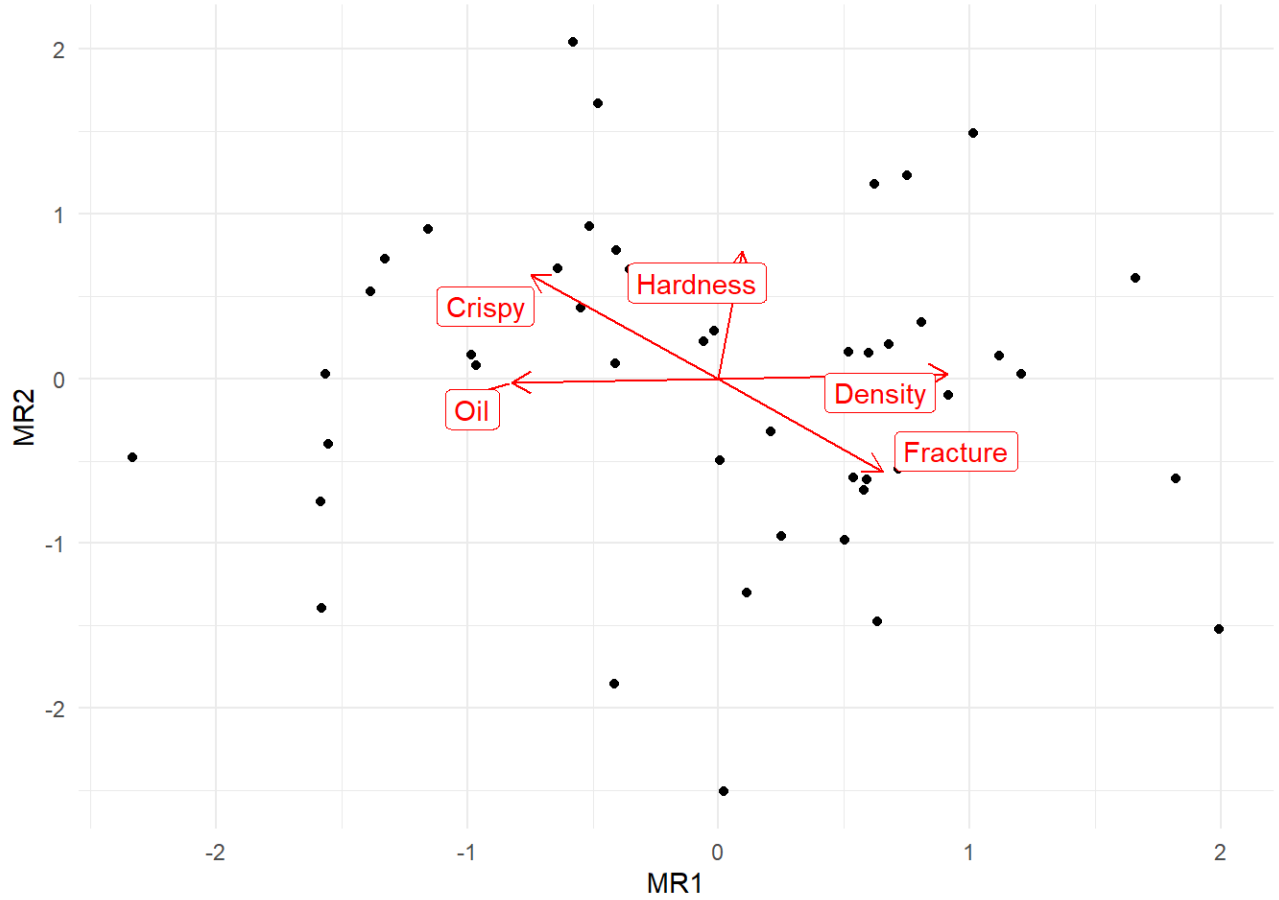


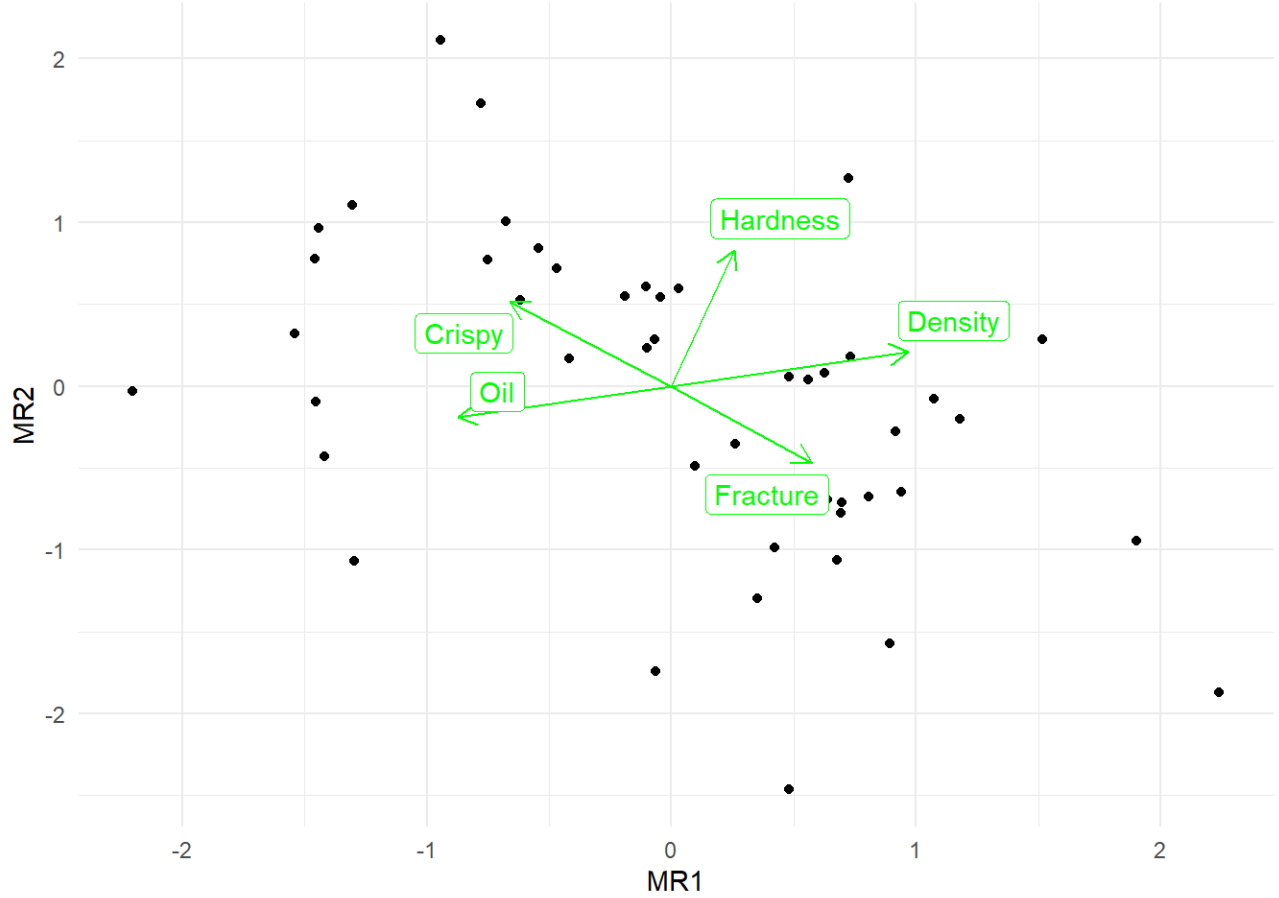
Gráfico de AF com rotação **Promax**:

```
# Convertenedo a tabela scores em um data frame
scores_df <- as.data.frame(food_texture_af_promax$scores)
# Convertenedo a tabela scores em um data frame

loadings_df <-  data.frame(matrix(
  as.numeric(food_texture_af_promax$l),
  attributes(food_texture_af_promax$l)$dim,
  dimnames=attributes(food_texture_af_promax$l)$dimnames))


# Definir o tema do gráfico
theme_set(theme_minimal())

# Criar o gráfico de dispersão
food_texture_af_promax_plot <- ggplot(scores_df, aes(x = MR1, y = MR2)) +
  geom_point() +
  geom_segment(
    aes(x = 0, y = 0, xend = MR1, yend = MR2),
    data = loadings_df,
    arrow = arrow(length = unit(0.3, "cm")),
    color = "green") +
  geom_label_repel(aes(label = rownames(loadings_df)),
                   data = loadings_df, color = "green")
food_texture_af_promax_plot
```



A interpetação mais apropriada para melhor interpretar os fatores é a rotação **Varimax**.

(d) Faça uma análise de componentes independentes (ACI). Escreva cada CI como função das variáveis originais. Tente interpretar cada componente que você vai reter.

```
# carregando as bibliotecas

library(fastICA)
library(ica)
```

Para realizar a ACI, vamos usar o pacote do R fastICA, com dois fatores, como indicado no item anterior.

```
# realizando a ACI
food_texture_ica_fast <- fastICA(food_texture[,2:6], 2)
```

Relembrando a teoria: A matriz de dados X é considerada ser a combinação linear de componentes não-gaussianos independentes. Isto é,

$$X = SA$$

onde S contém os componentes independentes e A é uma matrix de “mistura”. Para obter S,é necessário “desmisturar” os dados (X), estimando uma matriz W, onde

$$S = XW$$

O algoritmo de fast ICA estima W tal que

$$XKW = S$$

Temos que a matriz A de mistura é:

```
food_texture_ica_fast$A
```

```
##            [,1]      [,2]       [,3]      [,4]      [,5]
## [1,] -0.5855092  59.08933  0.1905578 -0.626026  28.47772
## [2,] -1.0274236 108.16043 -1.4486533  3.873568 -11.76736
```

A matriz S de componentes independetes, por sua vez, é:

```
food_texture_ica_fast$S
```

```
##              [,1]       [,2]
##  [1,] -0.58303223  1.21823705
##  [2,] -0.29365011 -1.66778746
##  [3,]  0.46992228 -0.14277933
##  [4,] -0.76854678  0.99823374
##  [5,]  0.78400647  0.65784200
##  [6,]  1.52711847 -1.45886629
##  [7,] -0.73663032 -0.59298556
##  [8,] -2.14591021  0.36313848
##  [9,]  0.15767401  0.81413496
## [10,]  0.37978051  0.60081522
## [11,] -0.27925119 -0.10412113
## [12,]  1.18158577 -0.85407100
## [13,]  1.22076785 -0.64662802
## [14,]  1.50800654  0.16888867
## [15,] -1.58042590  1.53239803
## [16,]  1.29438720 -1.51654434
## [17,] -1.06061630 -1.33906481
## [18,]  0.50506584  0.02263936
## [19,] -0.98837775 -0.96439812
## [20,]  0.03495639 -0.96965706
## [21,]  1.82835051  0.31955794
## [22,]  0.52754982  0.84264487
## [23,]  0.38979051 -0.93117894
## [24,] -0.93358114 -0.89798949
## [25,]  0.97541654 -0.83365021
## [26,]  0.68924202 -0.21692442
## [27,] -0.53379020  1.09952488
## [28,]  0.04113967  0.55497948
## [29,] -0.01164528 -0.11062752
## [30,] -2.38284424 -0.66427883
## [31,]  0.07096920  2.01581154
## [32,] -0.04001371 -0.27726062
## [33,] -0.20798175  2.58637469
## [34,]  0.11650142 -0.78004190
## [35,]  1.95125062 -0.67282285
## [36,] -1.55643445 -1.80929827
## [37,] -1.48340324 -1.24639180
## [38,]  0.06879673 -1.26376274
## [39,] -0.78122185  0.49486888
## [40,] -0.29817125  1.24826235
## [41,]  0.83188305  0.67744348
## [42,] -0.10570690  0.17194592
```

```
## [43,]  0.41834316  0.34842365
## [44,]  1.09203939  1.64464649
## [45,] -0.06598104  0.51978673
## [46,] -0.21849559  0.18867561
## [47,]  1.48738630  0.45720009
## [48,] -0.09372920  0.67591042
## [49,] -0.83684288 -1.00052664
## [50,] -1.56564681  0.73927281
```

A matriz W é dada por:

```
food_texture_ica_fast$W
```

```
##           [,1]       [,2]
## [1,] 0.4851685  0.8744207
## [2,] 0.8744207 -0.4851685
```

Por fim, a matriz K é dada por :

```
food_texture_ica_fast$K
```

```
##               [,1]          [,2]
## [1,] -0.00007772104 -0.00001430439
## [2,]  0.00810066862 -0.00085470114
## [3,] -0.00007718264  0.00092074597
## [4,]  0.00020266457 -0.00256986131
## [5,]  0.00023181268  0.03241593345
```

Vamos calcluar KW:

```
food_texture_ica_fast_KW <- food_texture_ica_fast$K %*% food_texture_ica_fast$W
food_texture_ica_fast_KW
```

```
##               [,1]          [,2]
## [1,] -0.00005021586 -0.00006102084
## [2,]  0.00318282098  0.00749806620
## [3,]  0.00076767273 -0.00051420704
## [4,] -0.00214881339  0.00142402987
## [5,]  0.02845763061 -0.01552448836
```

Com a matriz KW, podemos criar a equação de componentens independentes em função de X Isto é,

$S_1 = -0.00005021585651868 32X_1 + 0.0031828209786998 2X_2 + 0.0007676727265024 78X_3 + -0.0021488133881232 1X_4 + 0.028457630605348 4X$

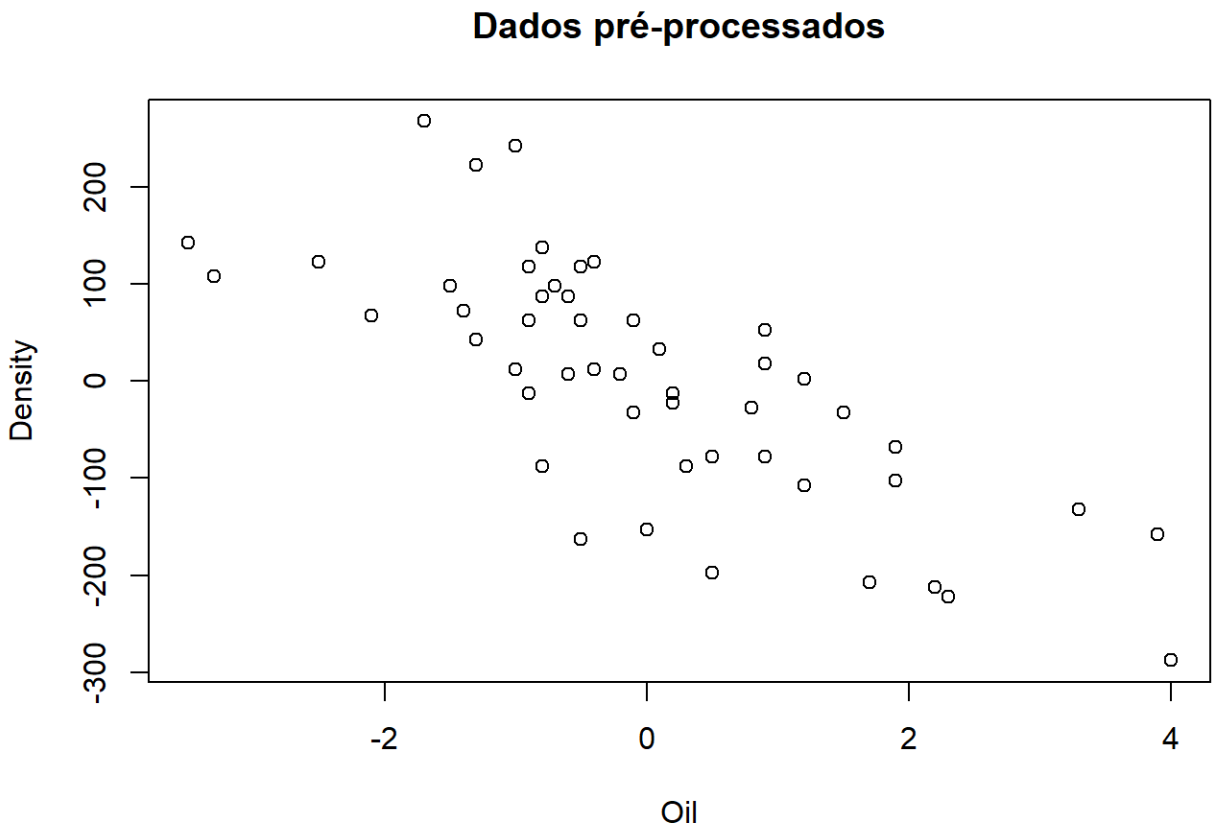
e

$S_2 = -0.00006102083849605 89X_1 + 0.0074980661960945 9X_2 + -0.0005142070426624 85X_3 + 0.0014240298739698 4X_4 + -0.015524488362216 6$

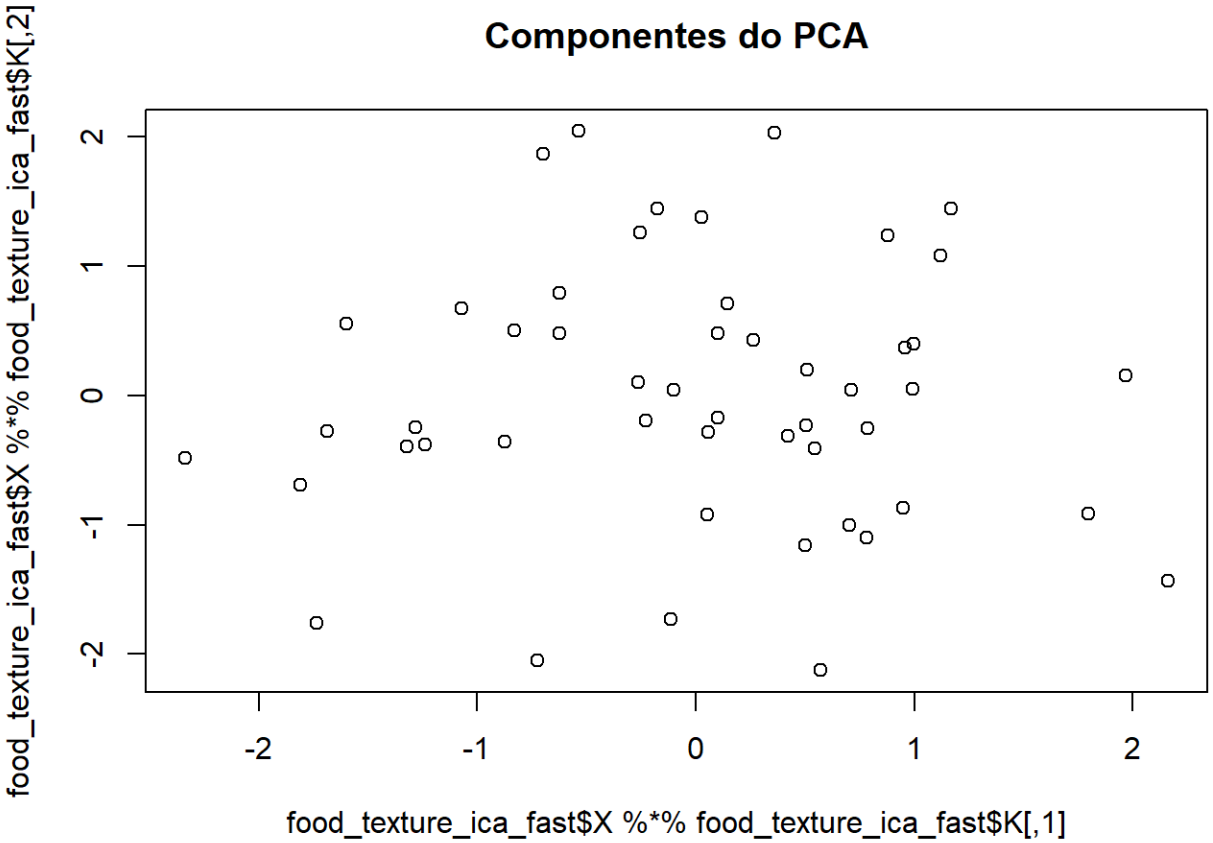
onde cada X_i é referente as variáveis de **food-texture** e cada S_j é referente ao componente independente.

Abaixo, podemos observar os gráficos dos dados pré-processados; dos componentes principais (que podem ser obitida pela matriz pré-branquamento (pre-whitening matrix), que projeta a matriz de pré-processados nos componentes principais); e por fim, um gráfico dos componentes independentes.

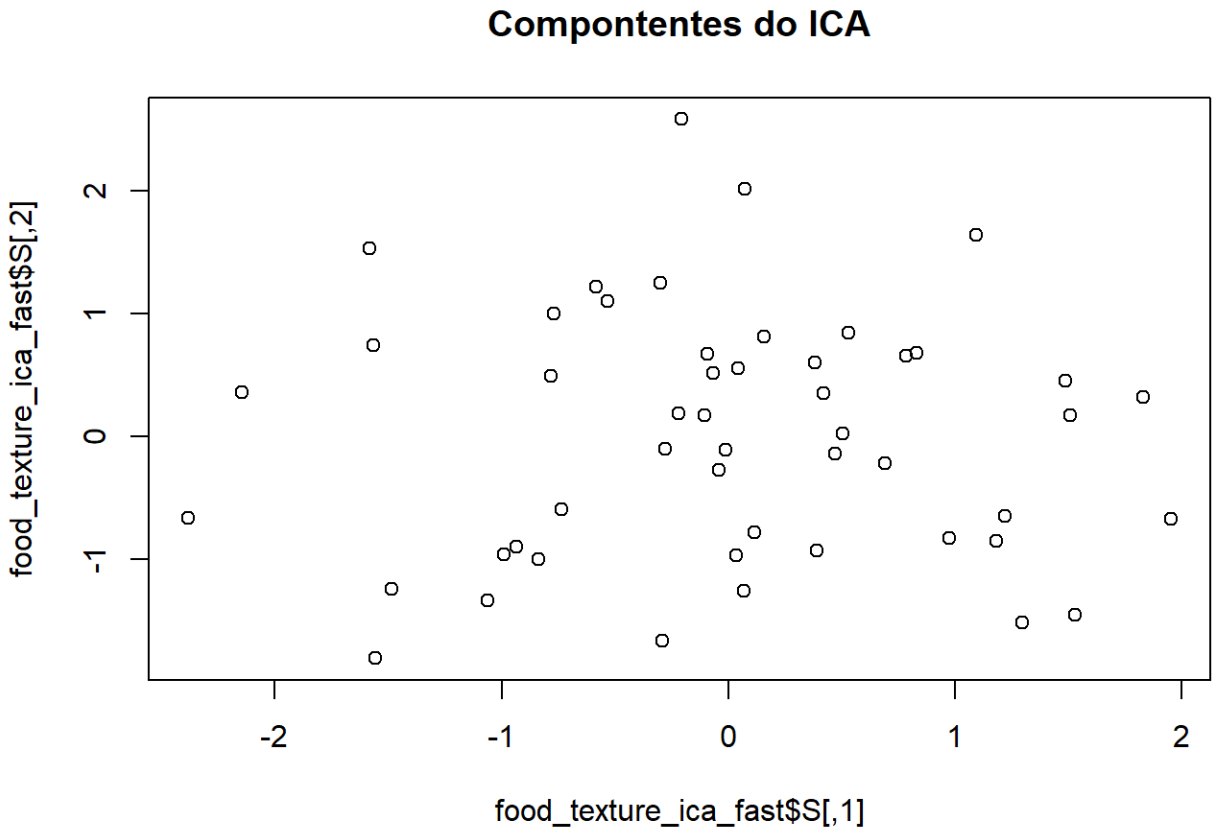
```
plot(food_texture_ica_fast$X, main = "Dados pré-processados")
```




```
plot(food_texture_ica_fast$X %*% food_texture_ica_fast$K, main = "Componentes do PCA")
```




```
plot(food_texture_ica_fast$S, main = "Compontentes do ICA")
```



Analisando a tabela KW e as equações de S em função de X, podemos ver as cargas para cada uma das colunas de X, e interpretar cada um dos dois componentes independentes. No caso do primeiro componente, as cargas relativas a dureza e densidade tem maior número absoluto, e são positivas. As cargas relativas a óleo e a fratura são negativas, e a relativa a crocâcnia é positiva, mas seus valores são relativamente menores. Para o segundo componente, as cargas relativas a dureza e desindade tem maior número absoluto, mas dureza tem valor negativo. As cargas relativas a óleo e a crocância são negativas, e a relativa a fatura é positiva, e, analogamente ao primeiro componente, seus valores são relativamente menores.

Em suma, o primeiro componente está relacionada com dureza, crocância e o negativo de fratura, em contrapondo com o segundo componente, que está relacionado com o negativo de dureza e crocância, e posititvamente com fratura. Podemos interpretar o primeiro componente como um componente de dureza e crocância e o segundo como um componente de moleza e não crocância.