Applied computational intelligence

Homework II

Student Name

Alef Carneiro de Sousa

Observações:

- Este arquivo contém apenas a solução das questões.
- Para este trabalho foi escolhida a Alternativa 1.
- Para todos os exercícios foi utilizado o conjunto de dados `solubility`. A seguir temos o cabeçalho que carrega o conjunto de dados:

```
# Importando a biblioteca que possui o dataset
library("AppliedPredictiveModeling")

# Carregando o dataset
data("solubility")
```

Exercise 0

Para este exercício serão utilzadas as bibliotecas: `caret`.

O primeiro passo utilizado no pré-processamento foi o tratamento da *skewness* dos dados. Esta parte do pré-processamento (e análise dos dados) foi realizada em três passos:

1. Cálculo da *skewness* dos dados originais (`solTrainX`)

```
previousSkewness = vector("list", length(solTrainX))

for(i in 1:length(solTrainX)) {
  previousSkewness[[i]] = skewness(solTrainX[[i]])
}

previousSkewness = data.frame(previousSkewness)
colnames(previousSkewness) = names(solTrainX)
```

2. Aplicação do método de Yeo-Johnson para redução da *skewness* dos dados, gerando um novo conjunto de dados (`solTrainXtrans`)

```
ppParams = preProcess(solTrainX, method = "YeoJohnson")
solTrainXtrans = predict(ppParams, solTrainX)

# O mesmo é feito para o conjunto de testes
ppParams = preProcess(solTestX, method = "YeoJohnson")
solTestXtrans = predict(ppParams, solTestX)
```

3. Os valores de *skewness* foram modificados para os dados contínuos e mostrados lado a lado com os valores de *skewness* originais

```
# 3.1. Cálculo da skewness dos dados transformados
transSkewness = vector("list", length(solTrainXtrans))

for(i in 1:length(solTrainXtrans)) {
  transSkewness[[i]] = skewness(solTrainXtrans[[i]])
}

transSkewness = data.frame(transSkewness)
colnames(transSkewness) = names(solTrainXtrans)

# 3.2. Mostrar a diferença entre previousSkewness e transSkewness no LaTeX
comp = previousSkewness != transSkewness
toConvertTex = as.data.frame(previousSkewness[comp])
toConvertTex$transSkewness = transSkewness[comp]
colnames(toConvertTex) = c("previousSkewness", "transSkewness")
rownames(toConvertTex) = names(previousSkewness)[which(comp)]

xtable(toConvertTex)
```

3. (cont.) A tabela gerada pelo código encontra-se a seguir:

Tabela 1: Comparação da skewness

	previousSkewness	transSkewness
MolWeight	0.99	-0.00
NumAtoms	1.36	0.00
NumNonHAtoms	0.99	-0.00
NumBonds	1.36	0.01
NumNonHBonds	0.97	-0.01
NumMultBonds	0.67	-0.14
NumRotBonds	1.57	0.02
NumDblBonds	1.36	0.16
NumAromaticBonds	0.79	-0.08
NumHydrogen	1.26	0.03
NumCarbon	0.93	-0.00
NumNitrogen	1.55	0.43
NumOxygen	1.77	0.04
NumHalogen	2.69	1.01
NumRings	1.03	0.00
HydrophilicFactor	3.40	0.25
SurfaceArea1	1.71	-0.23
SurfaceArea2	1.47	-0.25

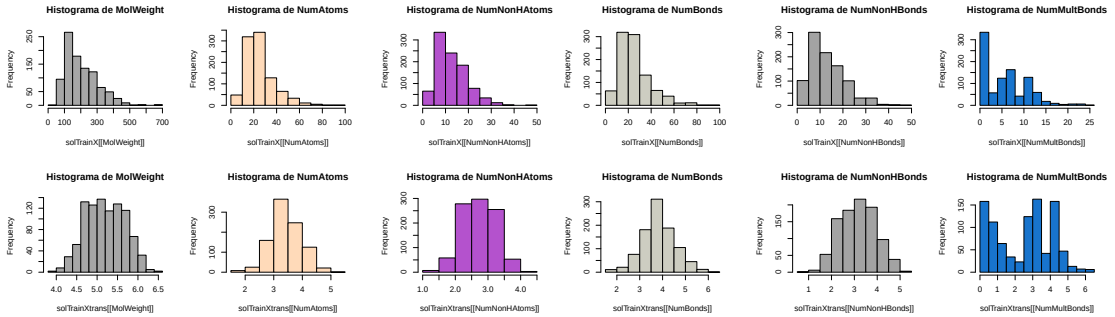
4. É feita uma média dos valores absolutos dos conjuntos de *skewness* da Tabela 1, para efeitos de comparação. Que nos retornou como resultado os valores 1,4373 e 0,1483, respectivamente

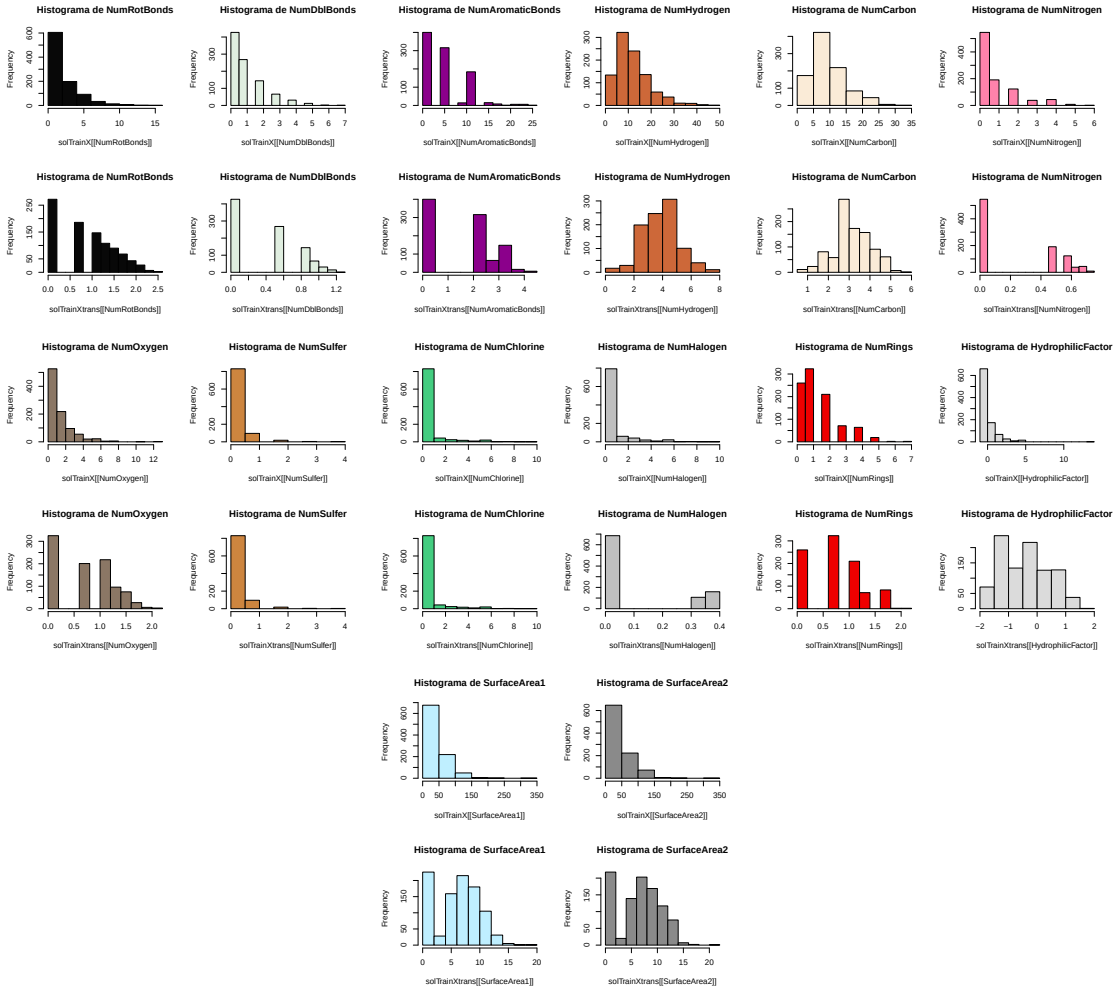
```
mean(abs(previousSkewness[comp]))
mean(abs(transSkewness[comp]))
```

5. Por último é feito o *plot* dos histogramas originais e transformados, para que se possa visualizar, na prática, o efeito da transformação

```
# Define que 4 plots ficaram
# juntos em um grid 2x2
par(mfrow = c(2, 2))
nomes = names(solTrainX)

for (i in seq(209, 228, 2)) {
  hist(solTrainX[[i]], main=stringi::stri_join("Histograma de ",
      nomes[i]), xlab=stringi::stri_join("solTrainX[[", nomes[i], "
      ]]"))
  hist(solTrainX[[i+1]], main=stringi::stri_join("Histograma de ",
      nomes[i+1]), xlab=stringi::stri_join("solTrainX[[", nomes[i
      +1], "]]"))
  hist(solTrainXtrans[[i]], main=stringi::stri_join("Histograma de "
      , nomes[i]), xlab=stringi::stri_join("solTrainXtrans[[", nomes
      [i], "]]"))
  hist(solTrainXtrans[[i+1]], main=stringi::stri_join("Histograma de
      ", nomes[i+1]), xlab=stringi::stri_join("solTrainXtrans[[",
      nomes[i+1], "]]"))

  invisible(readline(prompt="Tecle [Enter] para continuar"))
}
```

Segue os plots dos histogramas:





É possível notar que os histogramas dos dados transformados estão mais centralizados, seguindo uma distribuição mais próxima da Gaussiana.

Até este momento o único pré-processamento realizado nos dados foi a redução de *skewness*. Agora vamos aplicar os métodos *center* e *scale*, que será realizado com o código abaixo:

```
ppParams = preProcess(solTrainXtrans, method = c("center", "scale"))
solTrainXtrans = predict(ppParams, solTrainXtrans)

# O mesmo é feito para o conjunto de testes
ppParams = preProcess(solTestXtrans, method = c("center", "scale"))
solTestXtrans = predict(ppParams, solTestXtrans)
```

Agora que os dados já estão devidamente transformados, será feita uma análise da correlação dos *predictors* entre si e com a saída. Para isso será construído um conjunto de dados contendo os dados dos *predictors* transformados e a saída. Em seguida, será utilizado as funções `corrplot` e `cor` para que seja obtida a matriz de correlação.

```
trainingData = solTrainXtrans
trainingData$Solubility = solTrainY

corrplot(cor(trainingData[209:229]), order = "hclust", type = "lower",
         method = "number", addCoefasPercent = TRUE)
```

A Figura 1 é o resultado do código acima. Nela podemos identificar correlação entre *predictors*. Entretanto não existe muita correlação entre os *predictors* e a saída.

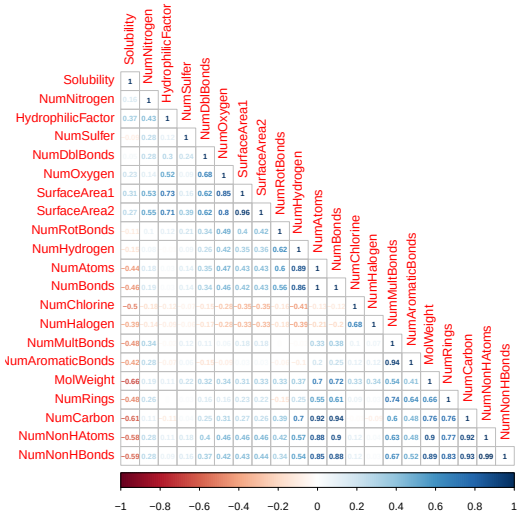


Figura 1: Matriz de Correlação

Após esta simples análise, será feito a última parte do pré-processamento. Redução do número de *predictors* que possuem alta correlação, lembrando que os mesmo *predictors* devem ser retirados do conjunto de testes. Para isso, foi utilizado o código abaixo:

```
altaCorr = findCorrelation(cor(trainingData), 0.9)
trainingDataFiltered = trainingData[-altaCorr]
solTestXtransFiltered = solTestXtrans[-altaCorr]
```

Exercise 1

Para este exercício serão utilzadas as bibliotecas: `e1071`, `caret`, `xtable` e `corrplot`.

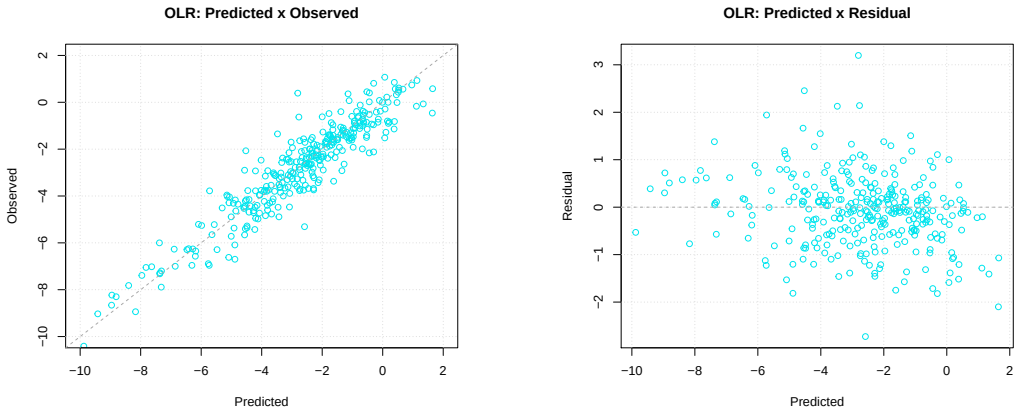
Agora que os nossos conjuntos de dados (tanto de treino como de testes) já foram pré-processados, iremos criar nossos modelos de predição.

Primeiramente será feito uma simples regressão linear a partir do nosso conjunto de treino e então tentaremos predizer o conjunto de testes. Para isso utilizaremos o conjunto `trainingData`, que foi criado anteriormente. Tal conjunto possui nossos *predictors* com seus valores transformados e filtrados e também possui os valores de `solTrainY`.

A seguir encontra-se o código que cria o modelo em questão e calcula os valores de $RMSE$ e R^2, além de criar os *plots* mostrados em Figura 2a e em Figura 2b.

```
## Regressão linear comum
# Treino
# Criação do modelo a partir do conjunto de dados
lmFit1 = lm(Solubility ~ ., data = trainingDataFiltered)

# Predição
# A partir do modelo e de um conjunto de entradas
# observadas, iremos tentar predizer qual a saída.
lmPred1 = predict(lmFit1, solTestXtransFiltered)

# Criando conjunto predicted + observed
lmPred1.values = data.frame(obs = solTestY, pred = lmPred1)

# Comparação dos valores dos observados e preditos
defaultSummary(lmPred1.values)[c("RMSE", "Rsquared")]

# Plot - Predicted x Observed
plot(obs ~ pred,
     data = lmPred1.values,
     col = "turquoise2",
     xlim = c(-10, 2),
     ylim = c(-10, 2),
     xlab = "Predicted", ylab = "Observed",
     main = "OLR: Predicted x Observed")
grid()
abline(0, 1, col = "darkgrey", lty = 2)

# Plot - Predicted x Residual
plot((obs - pred) ~ pred,
     data = lmPred1.values,
     col = "turquoise2",
     xlab = "Predicted", ylab = "Residual",
     main = "OLR: Predicted x Residual")
grid()
abline(h = 0, col = "darkgrey", lty = 2)
```

Como resultado encontramos valores de $RMSE$ e R^2 iguais a 0,7664 e 0,8711, respectivamente, e os seguintes *plots*:



(a) Comparação com observado

(b) Resíduos

Figura 2: Verificação dos resultados

Agora será feita uma regressão linear e, para validação dos dados, ao invés do conjunto de testes será utilizada a 10-*fold cross validation*. Este método de validação, cria, a partir do nosso conjunto de treino, diferentes conjuntos de teste e validação, com o objetivo de estimar os valores de $RMSE$ e R^2 do modelo. Isto pode ser feito com o código abaixo.

Uma pequena observação quanto ao código, nele é utilizada a função `set.seed`, responsável por dizer qual será a "semente"(fonte) utilizada para calcular determinados valores aleatórios. A técnica *k-fold cross validation* utiliza aleatoriade para realizar a reamostragem dos dados, a função `set.seed(100)` não é necessária, a mesma só foi utilizada caso o leitor queira replicar os resultados obtidos, tanto neste caso como em outros que aparecerão posteriormente.

```
## OLR com 10-fold cross validation
# Cross-Validation
# Criação do objeto de controle que será utilizado:
# 10-fold cross validation
ctrl = trainControl(method = "cv", number = 10)

# Treino
set.seed(100) # Necessário para se chegar aos mesmos resultados
lmFitCV = train(Solubility ~ .,
                data = trainingDataFiltered,
                method = "lm",
                trControl = ctrl)

lmFitCV$results[c("RMSE", "Rsquared")]
```

Os resultados de ambos (*test set* e *cross validation*) podem ser vistos na Tabela 2

Tabela 2: Comparação dos erros

	RMSE	R^2
Test set	0,7664	0,8711
10-CV	0,7011	0,8805

Exercise 2

Para este exercício serão utilzadas as bibliotecas: `caret`, `xtable` e `lmridge`.

Neste exercício, teremos os nossos modelos penalizados, técnica utilizada para evitar o *overfitting*. Será utilizada a técnica Ridge de penalização.

Para escolher a melhor forma de penalização, será calculado os valores de $RMSE$ para valores de λ de 0,01 até 0,1, variando esses valores em 0,01.

Para calcular esses valores de $RMSE$ é necessário criar todos os modelos para os diferentes valores de λ, calcular os dados de saída a partir dos modelos e das saídas geradas, será verificado qual valor de λ cria um modelo com menor valor de $RMSE$.

Por fim, os dados criados por este valor de λ são os que serão utilizados. As operações citadas a cima podem ser realizadas com o código a seguir:

```
## Regressão linear penalizada
# Treino
# Criando modelos penalizados
# Valores de lambda variando de 0.01 a 0.1 com passo 0.01
lmFitRidge = lmridge(Solubility ~ ., data = trainingData, K = seq(0.01,
    0.1, 0.01))

# Predição
# Criando predições para cada um dos modelos que foram
# criados, para os diferentes valores de lambda
lmPredRidge = predict.lmridge(lmFitRidge, solTestXtrans)

# Inicializando uma lista para receber
# os valores de RMSE que serão calculados
rmse = vector("numeric", length(lmFitRidge$K))

# Cálculo dos valores de RMSE
for (i in 1:10) {
  lmRidge.values = data.frame(obs = solTestY, pred = lmPredRidge[, i])
  rmse[i] = defaultSummary(lmRidge.values)["RMSE"]
}

# Pegando índice do menor RMSE, para encontrar
# o modelo que melhor representa os dados
indexMin = which.min(rmse)

# Atualizando os valores que existiam para que
# contenham apenas os valores do melhor modelo
lmRidge.values = data.frame(obs = solTestY, pred = lmPredRidge[,
    indexMin])
lmFitRidge$coef = lmFitRidge$coef[, indexMin]
```

```
lmFitRidge$rfit = lmFitRidge$rfit[, indexMin]
lmPredRidge = lmPredRidge[, indexMin]
defaultSummary(lmRidge.values)[c("RMSE", "Rsquared")]

# Plot - RMSE x lambda
plot(lmFitRidge$K, rmse,
     xlab = expression(lambda),
     ylab = "RMSE",
     col = "darkred",
     pch = 17)
lines(lmFitRidge$K, rmse, col = "darkred")
grid()

# Plot - Predicted x Observed
plot(obs ~ pred,
     data = lmRidge.values,
     col = "turquoise2",
     xlim = c(-10, 2),
     ylim = c(-10, 2),
     xlab = "Predicted", ylab = "Observed")
grid()
abline(0, 1, col = "darkgrey", lty = 2)

# Plot - Predicted x Residual
plot((obs - pred) ~ pred,
     data = lmRidge.values,
     col = "turquoise2",
     xlab = "Predicted", ylab = "Residual")
grid()
abline(h = 0, col = "darkgrey", lty = 2)
```

Como resultado podemos verificar valores de $RMSE$ e de R^2 iguais a 0,7104 e 0,8845, respectivamente.

Além do que já havia sido citado, o código também gera os *plots* mostrados na Figura 3, na Figura 4a e na Figura 4b.

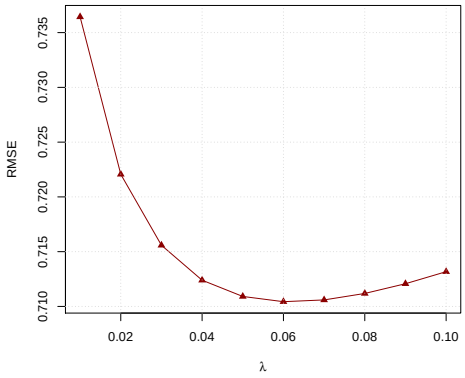
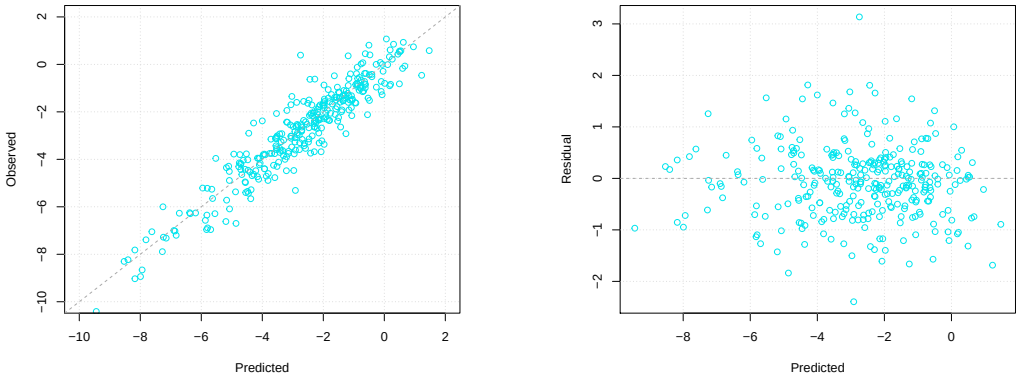


Figura 3: λ x $RMSE$



(a) Predicted x Observed

(b) Predicted x Residual

Figura 4: Resultado final - modelo com menor $RMSE$

As duas últimas figuras mostram o resultado do modelo com menor valor de $RMSE$.

Agora o mesmo será feito utilizando 10-*fold cross validation*. Será criado o modelo de regressão com penalização Ridge para os mesmos valores de λ utilizados anteriormente. Como resultado disso é obtido os valores de $RMSE$ e de R^2 mostrados na Tabela 3, onde é possível verificar que o valor de λ que minimiza o $RMSE$ é 0,03 ($RMSE$ = 0,6736). Algo que também pode ser verificado pelo gráfico contido na Figura 5.

Tabela 3: Erros para cada valor de λ

	λ	$RMSE$	R^2
Fold 01	0.01	0.6835	0.8863
Fold 02	0.02	0.6753	0.8891
Fold 03	0.03	0.6736	0.8899
Fold 04	0.04	0.6744	0.8901
Fold 05	0.05	0.6766	0.8898
Fold 06	0.06	0.6797	0.8894
Fold 07	0.07	0.6835	0.8889
Fold 08	0.08	0.6876	0.8883
Fold 09	0.09	0.6922	0.8876
Fold 10	0.10	0.6972	0.8870

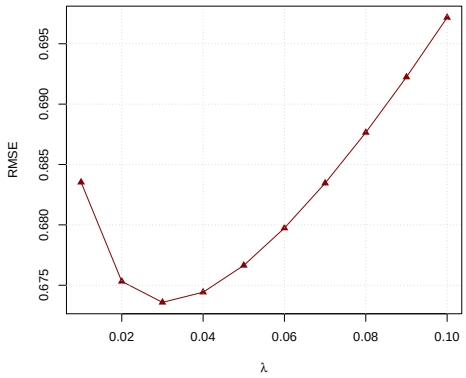


Figura 5: λ x $RMSE$ - 10-*fold cross validation*

Abaixo está o código utilizado para gerar estes resultados.

```
## Regressão linear penalizada com cross validation
# Treino
ridgeGrid = data.frame(.lambda = seq(.01, .1, length.out = 10))
set.seed(100) # Necessário para se chegar aos mesmos resultados
ridgeRegFit = train(Solubility ~ .,
                    method = "ridge",
                    data = trainingData,
                    # Valores de penalidade a ser utilizado
                    tuneGrid = ridgeGrid,
                    trControl = ctrl)
# Visualização dos resultados
ridgeRegFit

# Plot - RMSE x lambda
plot(ridgeGrid$.lambda,
     ridgeRegFit$results$RMSE,
```

```
     xlab = expression(lambda),
     ylab = "RMSE",
     col = "darkred",
     pch = 17)
lines(ridgeGrid$.lambda, ridgeRegFit$results$RMSE, col = "darkred")
grid()

# Gerar tabela, em LaTeX, com diversos valores
# de RMSE e R^2 para cada valor de lambda
xtable(ridgeRegFit$results[c("lambda","RMSE", "Rsquared")],
       digits = c(1, 2, 4, 4))
```

Exercise 3

Para este exercício serão utilzadas as bibliotecas: `caret` e `pls`.

Neste último caso é utilizado PLSR, que possui o objetivo de diminuir o número de *predictors* retirando aqueles que possuem uma maior correlação. Diferente do que foi feito anteriormente, o PLSR não procura correlação aos pares e sim de cada *predictor* com todos os outros.

Como no conjunto de dados `solubility` não há necessidade de redução do número de *predictors*, será feita a análise para todos os casos do número de componentes e escolhido o melhor modelo com base no $RMSE$.

Primeiramente isto é feito utilizando o conjunto de testes para validação. Segue abaixo o código capaz de realizar este cálculo:

```
## PLS
# Treino
plsFit = plsr(Solubility ~ ., data = trainingData)

# Predição
plsPred = predict(plsFit, solTestXtrans, ncomp = 1:plsFit$ncomp)

# Inicializando uma lista para receber
# os valores de RMSE que serão calculados
rmse = vector("numeric", plsFit$ncomp)

# Cálculo dos valores de RMSE
for (i in 1:plsFit$ncomp) {
  plsPred.values = data.frame(obs = solTestY, pred = plsPred[, , i])
  rmse[i] = defaultSummary(plsPred.values)["RMSE"]
}

# Pegando  ndice  do menor RMSE, para encontrar
# o modelo que melhor representa os dados
indexMin = which.min(rmse)

# Atualizando os valores que existiam para que
# contenham apenas os valores do melhor modelo
plsPred = predict(plsFit, solTestXtrans, ncomp = indexMin)
plsPred.values = data.frame(obs = solTestY, pred = plsPred[, 1, 1])
defaultSummary(plsPred.values)[c("RMSE", "Rsquared")]

# Plot - RMSE x #Componentes
```

```
plot(1:plsFit$ncomp,
     rmse,
     xlab = "#Componentes",
     ylab = "RMSE",
     col = "darkred",
     pch = 17)
lines(1:plsFit$ncomp, rmse, col = "darkred")
grid()
```

Com isso podemos verificar que o número de componentes que minimiza o $RMSE$ (para o valor de 0,7156) é 11. A Figura 6 mostra o gráfico com a relação entre o $RMSE$ e o número de componentes:

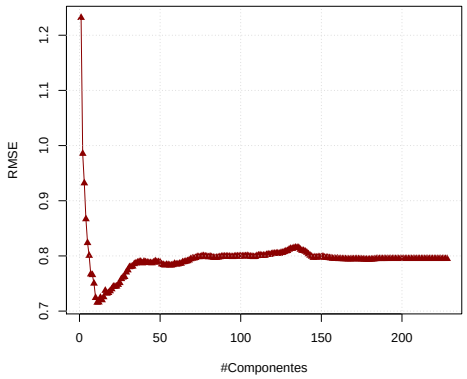


Figura 6: Componentes x $RMSE$