UNIVERSIDADE FEDERAL DO CEARÁ

Centro de Tecnologia

Departamento de Teleinformática

HOMEWORK II

Regressão Linear

Aluno: Caio Cid Santiago Barbosa
Matrícula: 378596

Disciplina: Inteligência Computacional Aplicada
Professor: Michela Mulas
Data: 6 de Novembro de 2017

Fortaleza, Ceará
2017

Parte 0

Sempre que começamos um tratamento de um *dataset*, o primeiro passo é o pré-processamento de seus dados. Desse modo, analisaremos os *predictors* para podermos fazer operações de pré-processamento. Primeiramente, adicionaremos as bibliotecas que utilizaremos no trabalho. Depois, analisaremos a *skewness* dos *predictors*. Para tal, removeremos os *predictors* binários, dado que eles não possuem *skewness* :

```
library(AppliedPredictiveModeling);
library(e1071);
library(corrplot);
library(caret);
library(elasticnet);
library(pls);

data(solubility);

skew <- apply(solTrainX[209:228],2,skewness);

print(paste("Média da skewness: ",mean(skew)))
print(paste("Máximo valor da skewness: ",max(skew)))
print(paste("Minimo valor da skewness: ",min(skew)))
```

	Média	Máximo	Mínimo
Skewness	1.64394822809875	3.83547356438057	0.66918164642201

Com isso, conseguimos chegar em algumas conclusões. A média da *skewness* é 1.64. Isso indica que os *predictors* tem uma tendência a serem deslocados para esquerda (*right skewed*). É interessante retirarmos a *skewness* pois as variações dos valores do preditores interessam mais do que a magnitude deles. Deixando os preditores deslocados prejudicaria pois levaria em conta seu valor absoluto, enquanto o mais interessante para analisarmos na regressão é como esses valores variam. Para corrigir essa *skewness*, utilizamos Box&Cox:

```
## Find the predictors that are not fingerprints
contVars <- names(solTrainX)[!grepl("FP",
↪  names(solTrainX))]

## Some have zero values, so we need to add one to them
↪  so that
## we can use the Box-Cox transformation.
contPredTrain <- solTrainX[,contVars] + 1
contPredTest <- solTestX[,contVars] + 1

pp <- preProcess(contPredTrain, method = "BoxCox")
contPredTrain <- predict(pp, contPredTrain)
contPredTest <- predict(pp, contPredTest)

## Reassemble the fingerprint data with the transformed
↪  values.
trainXtrans <- cbind(solTrainX[,grep("FP",
↪  names(solTrainX))], contPredTrain)
testXtrans <- cbind( solTestX[,grep("FP",
↪  names(solTestX))], contPredTest)
```

Esse código foi retirado direto da documentação do *dataset*, onde ele cria duas estruturas chamadas *solTestXtrans* e *solTrainXtrans*, que correspondem ao conjunto de teste transformado e ao conjunto de treino transformado, respectivamente. Agora, analisaremos a correlação entre os *predictors*:

```
correlation <-
↪  cor(cbind(solTrainXtrans[209:228],solTrainY))
corrplot(correlation, order = "hclust", mar = c(1,1,1,1))
```

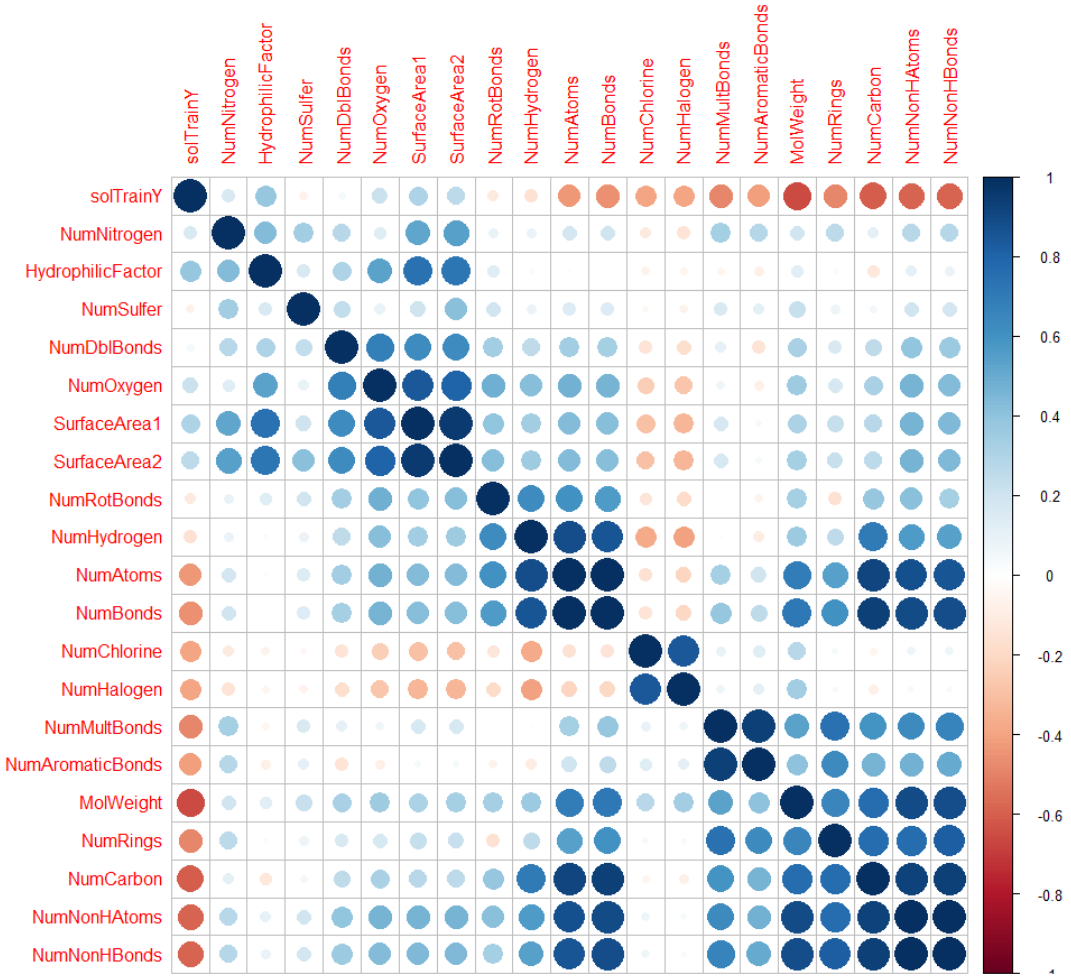

Figura 1: Matriz de Correlação entre os *predictors* e a saída representada com círculos.

Podemos verificar que existem muitos pares de *predictors* fortemente correlacionados, que deverão e serão retirados posteriormente para a regressão linear. Além disso, incluímos a saída para verificar se possui algum *predictor* correlacionado a saída e podemos verificar que existem alguns preditores inversamente proporcionais a saída, como o peso molecular. Essa observação é coerente, tendo em vista que quanto maior a molécula, mais difícil ela é de se dissolver.

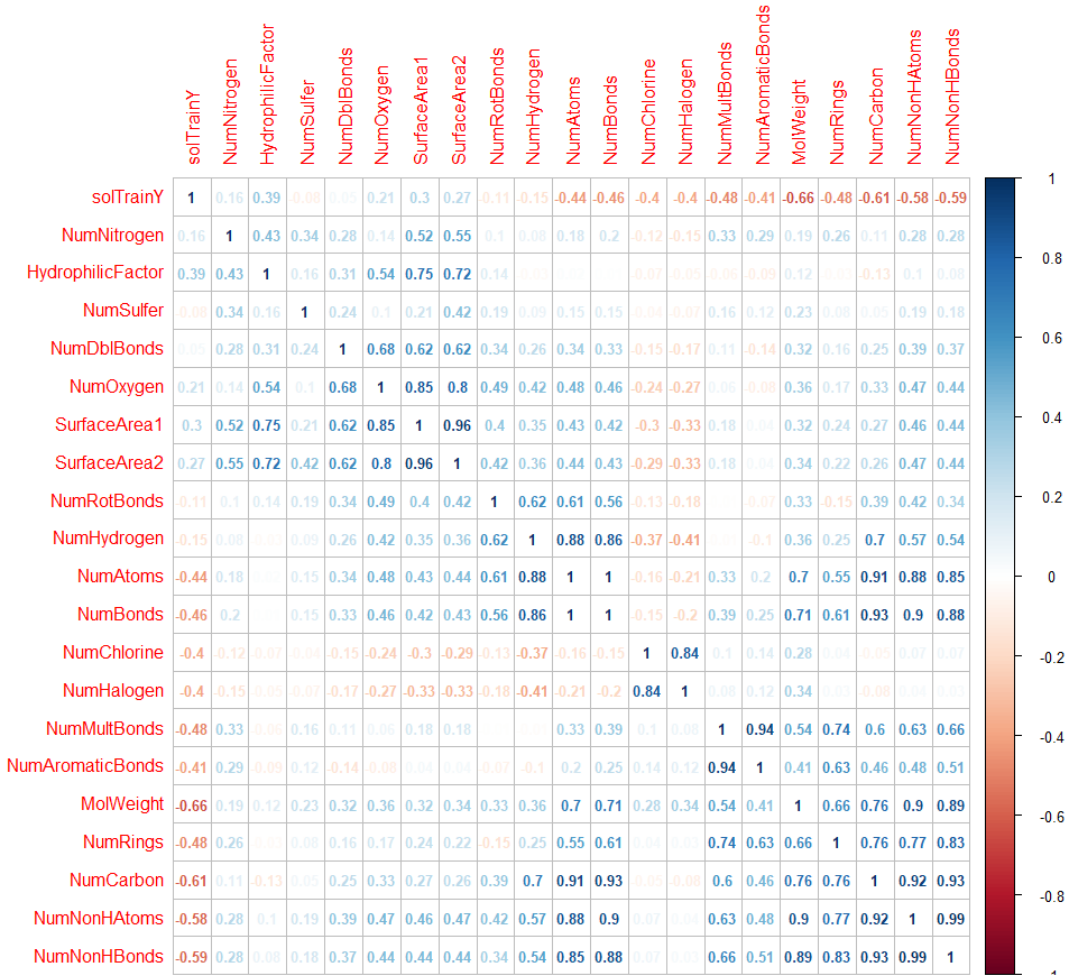

Figura 2: Matriz de Correlação entre os *predictors* e a saída representado em valores.

Parte 1

Para fazermos um modelo de regressão linear ordinário, devemos primeiramente levar em conta o nosso *dataset*. Devemos analisar se as correlações entre os *predictors* e retirar os que são muito correlacionados. Dado que utilizamos a formula $(X^TX)^{-1}X^Ty$, onde X é a matriz dos *predictors* e y é o vetor resposta, o termo $(X^TX)^{-1}$, que é proporcional a matriz de covariância dos *predictors*, só vai possuir inversa única quando os termos não forem colineares e numero de amostras for maior que os de *predictors*. Sendo assim, nosso primeiro passo será retirar os *predictors* que tem correlação maior que 0.9.

```
corThresh <- .9;
tooHigh <- findCorrelation(cor(solTrainXtrans),
↪  corThresh);
corrPred <- names(solTrainXtrans)[tooHigh];
length(tooHigh)
corrPred

trainXfiltered <- solTrainXtrans[, -tooHigh];
testXfiltered <- solTestXtrans[, -tooHigh];
```

Após executado o código, identificamos 38 *predictors* com correlação maior que 0.9. Esses *predictors* foram então descartados para podermos aplicarmos a regressão linear. O próximo passo é aplicar a função ***lm***. Para aplicar essa função, é preciso que o *output* esteja no mesmo *dataset* que os *predictors* que vão ser analisados. Depois disso, aplicamos a função e depois verificamos a eficiência dela com o *test set* utilizando a função ***predict***:

```
set.seed(200)
trainingData <- trainXfiltered
trainingData$solubility <- solTrainY

lmFiltered <-lm(Solubility ~ ., data = trainingData)
lmPred1 <- predict(lmFiltered, testXfiltered)

lmValues1 <- data.frame(obs = solTestY, pred = lmPred1)
defaultSummary(lmValues1)
```

Resultados da regressão linear ordinária aplicada ao *test set*

RMSE	R^2
0.7601790	0.8669049

Desse modo, conseguimos obter um modelo de regressão linear ordinário para o nosso *dataset*. O RMSE e o R^2 para esse modelo é 0.760 e 86,7%, respectivamente. Agora, verificaremos as medidas utilizando um *10-fold cross validation*. Para isso, utilizaremos outra função, a ***train***:

```
fold10 <- trainControl(method = "cv", number = 10);

lmTrain10 <- train(trainXfiltered, solTrainY, method =
↪  "lm", trControl = fold10);
lmTrain10

plotlm10ob<-xyplot(solTestY ~
↪  predict(lmTrain10,testXfiltered), type = c("p", "g"),
↪  xlab = "Predicted", ylab = "Observed", col = "blue")
plotlm10re<-xyplot(resid(lmTrain10) ~
↪  predict(lmTrain10,testXfiltered), type = c("p", "g"),
↪  xlab = "Predicted", ylab = "Residuals", col = "blue")

print(plotlm10ob, pos = c(0.0,0.0,0.5,1), more = TRUE)
print(plotlm10re, pos = c(0.5,0.0,1,1), more = FALSE)
```

Resultados da regressão linear ordinária utilizando *10-fold cross validation*

RMSE	R^2
0.691249	0.8863239

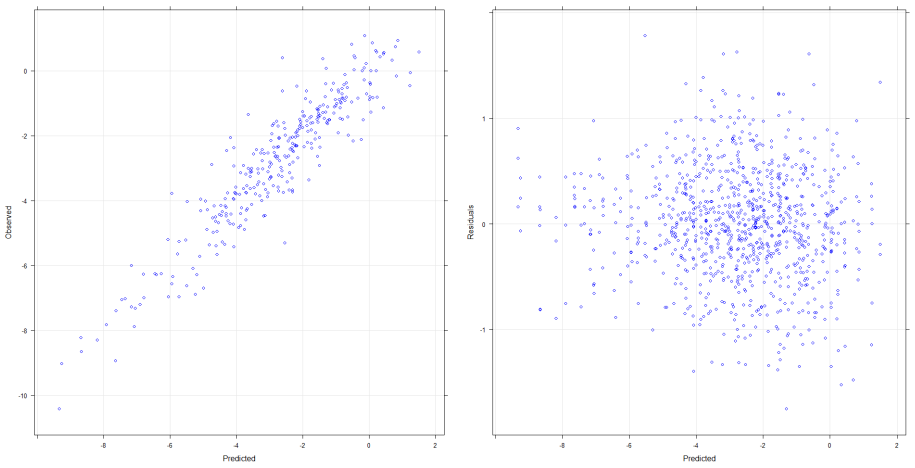

Figura 3: Esquerda: Grafico dos valores observados pelo os previstos pelo modelo de regressão linear. Direita: Gráfico dos resíduos do modelo de regressão linear pelo valor previsto.

Verificamos então o RMSE e o R^2 medido pelo *10-fold cross validation*. Obtemos 0.69 e 88,6%, respectivamente. Comparados com os valores dados

pela verificação pelo *test set*, 0.760 e 86,7%, podemos concluir que ambas verificações tem eficiência bem parecida para esse *dataset*. No nosso caso, temos muitas amostras, de modo que usar um *test set* não é problema, mas no caso de *datasets* menores, utilizar *K-fold cross validation* pode ser a única opção.
Analisando a Figura 3, podemos concluir que o modelo está coerente, pois verificamos a linearidade entre o previsto e o observado do lado esquerdo, e do lado direito vemos que os pontos de resíduo estão dispersos em torno do 0, de modo que seu somatório deve dar próximo disso.

Parte 2

Nessa parte, devemos fazer um modelo de regressão linear penalizado L_2. A regressão linear penalizada foi introduzida com o intuito de consertar algumas dificuldades geradas pela regressão linear ordinária. Essas dificuldades são, principalmente, o *overfitting*, que é uma visão otimista dos dados, gerando um modelo "viciado", e os preditores correlatos, que são tirados para que o modelo funcione. Esses casos são gerados pois as vezes o parâmetro da regressão linear ordinária pode estar *inflated*. O modelo de regressão linear penalizada de ***Ridge*** resolve isso aplicando um termo de penalização ao quadrado na função *hat*. Para aplicarmos a regressão linear penalizada, usaremos o modelo de ***Ridge***. Primeiramente, iremos aplicar a função ***train*** com *10-fold cross validation* para acharmos o melhor valor de lambda. Além disso, pre-processamos os dados para eles ficaram normalizados e centralizados. Dessa forma, os contornos da função do erro fica mais redondo e espaçado, facilitando na hora de aplicarmos *Ridge*. Se não normalizassemos os valores, teriamos fronteiras mais próximas e elípticas.

```
ridgeGrid <- data.frame(.lambda = seq(0, .1, length =
↪  15))
set.seed(200)
ridgeRegFit <- train(solTrainXtrans, solTrainY, method =
↪  "ridge", tuneGrid = ridgeGrid, trControl =
↪  fold10,preProc = c("center", "scale"))
ridgeRegFit
plot(ridgeRegFit, xlab = "Penalty", ylab = "RMSE (Cross
↪  Validation)")
```

Resultados da regressão linear Ridge L_2		
lambda	RMSE	R^2
0.000000000	0.6940069	0.8838059
0.007142857	0.6824777	0.8879880
0.014285714	0.6769873	0.8902612
0.021428571	**0.6755135**	**0.8911183**
0.028571429	0.6758089	0.8913708
0.035714286	0.6770933	0.8913038
0.042857143	0.6790107	0.8910488
0.050000000	0.6813715	0.8906763
0.057142857	0.6840645	0.8902281
0.064285714	0.6870200	0.8897302
0.071428571	0.6901922	0.8891999
0.078571429	0.6935498	0.8886490
0.085714286	0.6970709	0.8880856
0.092857143	0.7007396	0.8875153
0.100000000	0.7045442	0.8869424

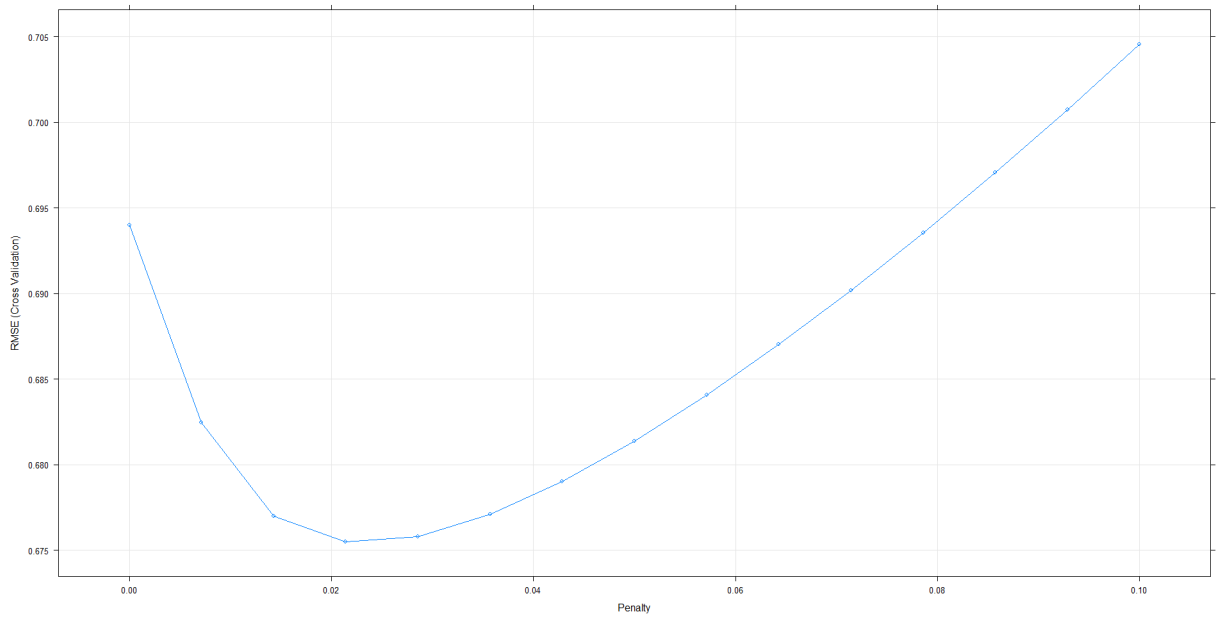

Figura 4: Gráfico da RMSE x Valores de lambda na regressão linear Ridge

Conseguimos identificar que o menor valor de RMSE dado esses 15 lambda é 0.6755135, que corresponde ao $\lambda = 0.021428571$. Podemos observar na Figura 4, a tendência que a curva RMSE x λ tem de decrescer primeiro e depois aumentar. Para melhorar a eficiência, podíamos acrescentar mais lambdas

perto do valor ótimo achado, de modo que teríamos um melhor ajuste fino. Agora, com o modelo já feito, iremos testar no *test set* e compararemos os valores de RMSE e R^2

```
lmRidgePred <- predict(ridgeRegFit, solTestXtrans)
ridge.df <- data.frame(pred = lmRidgePred, obs =
↪  solTestY)
defaultSummary(ridge.df)
```

Resultados da regressão linear Ridge L_2 aplicada ao *test set*

lambda	RMSE	R^2
0.021428571	0.7218669	0.8800143

Comparando os resultados do RMSE e R^2 do *test set* com os feitos por *cross validation*, podemos concluir que o modelo com *cross validation* estava sendo otimista, já que tem uma diferença considerável do RMSE dele (0.6755135) para o feito com o *test set* (0.7218669). Esse resultado é coerente, pois a verificação por *cross validation* usa apenas o conjunto de treino, de modo que ela é um pouco menos eficaz que uma verificação via *test set*.

Parte 3

Nesse passo, iremos aplicar um modelo de regressão linear PCR ou PLS. A PCR (*Principal Componente Regression*) é uma regressão que resolve os problemas de correlação entre os preditores utilizando PCA(*Principal Component Analysis*). Pela natureza do PCA, de garantir que nenhum preditor tem relação entre si, ele é um ótimo modelo. Contudo, ele não leva em consideração nenhum aspecto da saída, de modo que se os preditores não tiverem uma variância relacionada a saída, ele não vai ser eficiente. Tendo em vista esse problema, e como nós foi sugerido no livro, utilizaremos o PLS. O PLS (*Partial Least Squares*) surgiu com o *nonlinear iterative partial least squares algorithm* (NIPALS), que linearizava modelos que eram não-lineares em seus parâmetros. Ele funciona de modo que consegue levar em consideração as variâncias dos preditores enquanto requisita que essas componentes tenham máxima correlação com as saídas, enquanto reduz sua dimensão para fazer a regressão. Desse modo, esse procedimento é chamado de ***supervisionado***(*supervised dimension reduction procedure*). Utilizando a função ***Train***, iremos desenvolver a PLS:

```
set.seed(200)
plsTune <- train(solTrainXtrans, solTrainY, method =
↪  "pls", tuneLength = 25, trControl = fold10, preProc =
↪  c("center", "scale"))
plsTune
plot(plsTune, xlab = "Number of components", ylab = "RMSE
↪  (Cross Validation)")
```

Resultados da PLS com *10-fold Cross Validation*

Nº of components	RMSE	R^2
1	1.2849536	0.6090610
2	1.0549228	0.7335585
3	0.9369236	0.7901420
4	0.8623083	0.8217281
5	0.8191507	0.8387654
6	0.7844736	0.8528814
7	0.7573361	0.8630147
8	0.7415002	0.8685131
9	0.7327694	0.8725782
10	0.7195060	0.8774698
11	0.7127775	0.8790996
12	0.7076509	0.8809095
13	0.7032763	0.8822672
14	0.7013346	0.8830113
15	0.6966530	0.8844692
16	0.6950322	0.8850275
17	0.6929195	0.8854159
18	**0.6896578**	**0.8866734**
19	0.6901727	0.8863902
20	0.6909659	0.8860459
21	0.6943647	0.8849180
22	0.6965550	0.8838458
24	0.6971807	0.8834652
25	0.6999703	0.8825882

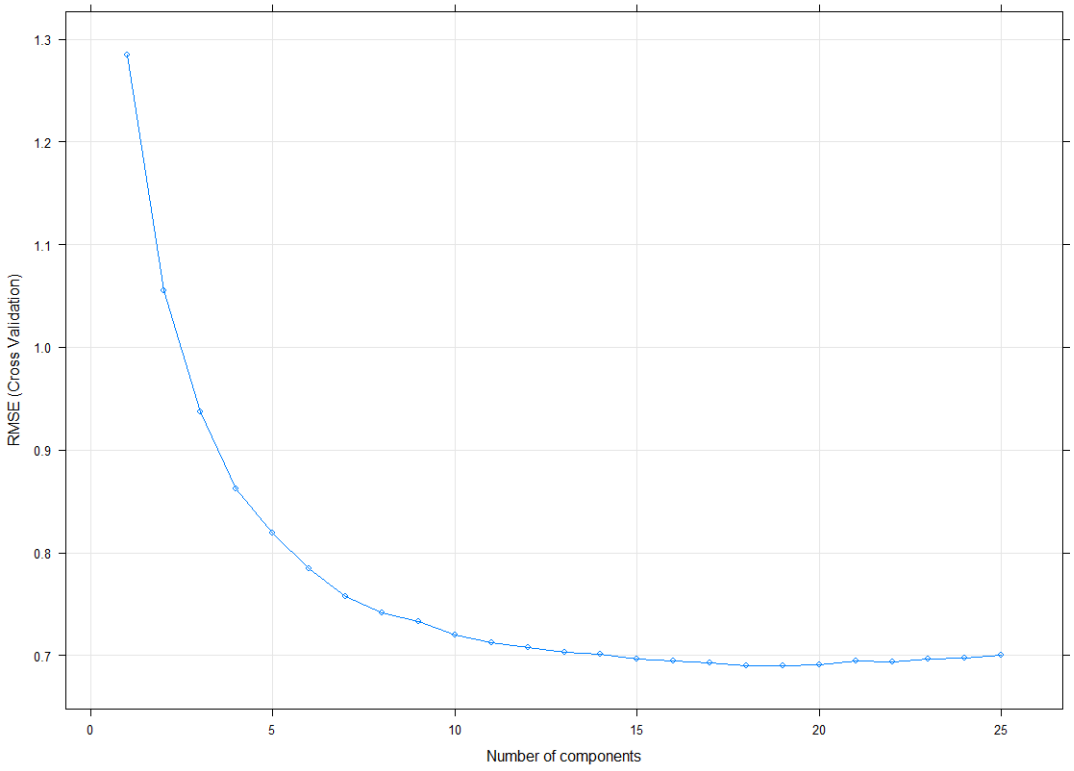

Figura 5: Gráfico da RMSE x Numeros de componentes na PLS

Na Figura 5 temos o gráfico dos erros pelo número de componentes. Percebemos que ela atinge o minimo aproximadamente com 18 componentes, o que condiz com o resultado achado na função *train*. Agora, iremos aplicar essa regressão no *test set* e medir sua eficiência.

```
plsPred <- predict(plsTune, solTestXtrans)
pls.df <- data.frame(pred = plsPred, obs = solTestY)
defaultSummary(pls.df)
```

Resultados da PLS aplicada ao *test set*

Nº Components	RMSE	R^2
18	0.7296398	0.8772282

Com isso, temos um valor razoavelmente maior de RMSE comparado ao *cross validation*, como temos tido durante todo esse trabalho (que é o usual).

Análise final

Para fechar, iremos fazer uma analise comparativa dos 3 modelos de regressão linear abaixo, e tentar verificar se os resultados estão coerentes.

Resultados das regressões lineares aplicadas no trabalho

	RLO	Ridge	PLS
RMSE	0.7601790	0.7218669	0.7296398
R^2	0.8669049	0.8800143	0.8772282

Percebemos que os valores de R^2 são semelhantes e relativamente altos, mas nem tanto. Desse modo, podemos concluir que os modelos correspondem bem e não possuem *overfitting*, que é o principal indicativo de altas porcentagens de R^2. Quanto ao RMSE, o fato mais curioso seria a melhor eficiência da regressão *Ridge* em relação a PLS, mesmo que por pouco. Como a PLS é um modelo mais robusto e supervisionado, a tendência seria de achar que sua eficiência seria melhor que a *Ridge*. Contudo, como constatado no livro-texto da disciplina, o PLS perde rendimento a medida que a quantidade de preditores aumenta. Desse modo, como possuimos muitos preditores, a PLS acaba perdendo sua eficiência a ponto da Regressão *Ridge* ser melhor que ela.