Chapter 6 homework linear regression

Chi Hang(Philip) Cheung

2025-03-29

To load data:

```
library(AppliedPredictiveModeling)
library(tidyverse)
library(caret)
library(glmnet)
library(lars)
library(elasticnet)
library(pls)
library(corrplot)
```

6.2 a) glance at the data:

```
data(permeability)
glimpse(permeability)
```

```
##  num [1:165, 1] 12.52 1.12 19.41 1.73 1.68 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:165] "1" "2" "3" "4" ...
##   ..$ : chr "permeability"
```

```
glimpse(fingerprints)
```

```
##  num [1:165, 1:1107] 0 0 0 0 0 0 0 0 0 0 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:165] "1" "2" "3" "4" ...
##   ..$ : chr [1:1107] "X1" "X2" "X3" "X4" ...
```

b) The fingerprint predictors indicate the presence or absence of substruc tures of a molecule and are often sparse meaning that relatively few of the molecules contain each substructure. Filter out the predictors that have low frequencies using the nearZeroVar function from the caret package. How many predictors are left for modeling?

Ans: only 388 predictors are remaining.

```
#original predictor count:
ncol(fingerprints) #1107 columns
```

```
## [1] 1107
```

```
#use nearZeroVar function to get the indices of the insignificant columns:
nzv_idx<- fingerprints %>% nearZeroVar()

#remove nzv columns:
filtered_fingerprints<- fingerprints[, -nzv_idx]

#calculate the remaining number of columns after nzv:
ncol(filtered_fingerprints) #Only 388 predictors are remaining
```

```
## [1] 388
```

c) Split the data into a training and a test set, pre-process the data, and tune a PLS model. How many latent variables are optimal and what is the corresponding resampled estimate of R^2?

ANS: N component = 2 and the R^2 = 0.49. PreProcessing is not needed in this dataset since the predictors are all binary data. The scale and centering are already standardized.

```
#set seed:
set.seed(123)

#Split training set:
train_idx<- createDataPartition(permeability, p = 0.7, list=FALSE) #Returns row indices

train_x<- filtered_fingerprints[train_idx,] #Make sure this is filtering rows with the ','
train_y<- permeability[train_idx]

test_x<-filtered_fingerprints[-train_idx,] #Make sure this is filtering rows with the ','
test_y<-permeability[-train_idx]
```

To tune, fit and predict the model:

```
set.seed(123)
#Combine the predictor and the response values into one training data:
train_data<- data.frame(train_x, permeability = train_y)
#Model fitting for PLS:
pls_fingerp<- plsr(permeability~., data=train_data, validation='CV')

#Auto-select the best component number based on RMSE within one standard deivation:
fingerp_ncomp<- selectNcomp(pls_fingerp, method = "onesigma") #ncomp = 2(2 latent variables)

#Alternatively, we use train() to get the full grid search of ncomp and the corresponding RMSE and R^2
pls_tune<- train(train_x, train_y,
                 method = 'pls',
                 tuneLength = 20,
                 trControl = trainControl(method='cv', number = 10))

print(pls_tune)
```

```
## Partial Least Squares
##
## 117 samples
## 388 predictors
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 105, 105, 105, 105, 106, 105, ...
## Resampling results across tuning parameters:
##
##   ncomp  RMSE      Rsquared   MAE
##    1     14.03702  0.2970378  10.957091
##    2     12.02548  0.4898218   8.759019
##    3     12.18351  0.4690614   9.110102
##    4     13.02379  0.4315368  10.189054
##    5     12.64148  0.4533627   9.366444
##    6     12.53764  0.4486215   9.257191
##    7     12.27157  0.4509604   8.990837
##    8     12.46335  0.4525561   9.488691
##    9     12.83980  0.4393277   9.936380
##   10     13.16934  0.4244272  10.009561
##   11     13.46293  0.4102709  10.373372
##   12     13.62527  0.3984178  10.418484
##   13     14.06114  0.3868089  10.744108
##   14     14.54725  0.3785087  11.067096
##   15     14.83297  0.3636468  11.197036
##   16     15.54279  0.3420223  11.664764
##   17     16.09415  0.3258030  12.035710
##   18     16.56339  0.3142702  12.212903
##   19     16.92798  0.3097090  12.502971
##   20     17.05832  0.3108168  12.519177
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was ncomp = 2.
```

```
#result is the same, which ncomp = 2 is the lowest RMSE
#R^2 from the training set is 0.446
```

(d) Predict the response for the test set. What is the test set estimate of R2?

ANS: Test set R^2 = 0.39. It is expected to be little lower than the training set.

```
#To predict:
fingerp_predict<- predict(pls_tune, newdata=test_x, ncomp=fingerp_ncomp)

#R^2 from the test set
postResample(fingerp_predict, test_y) #R^2= 0.39
```

```
##      RMSE  Rsquared       MAE
## 11.198599  0.388909  7.463975
```

(e) Try building other models discussed in this chapter. Do any have better predictive performance?

ANS: alpha = 0.8888889 and lambda = 1 and R^2 is 0.56, which is higher than the PLS method but the RMSE is still higher than the PLS method. The PLS method is still better in performance

```
#define enetGrid range:
enet_Grid<- expand.grid(alpha=seq(0,1,length=10),
                        lambda = seq(0.001, 1, length = 10))
#to find the most optimal lambda for enet():
set.seed(1)
enet_tune<- train(x=train_x, y=train_y,
                  method='glmnet',
                  tuneGrid = enet_Grid,
                  trControl = trainControl(method='cv', number = 10))

print(enet_tune)
```

```
## glmnet
##
## 117 samples
## 388 predictors
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 106, 105, 106, 106, 105, 105, ...
## Resampling results across tuning parameters:
##
##   alpha      lambda  RMSE      Rsquared   MAE
##   0.0000000  0.001   12.31233  0.4431602   9.147622
##   0.0000000  0.112   12.31233  0.4431602   9.147622
##   0.0000000  0.223   12.31233  0.4431602   9.147622
##   0.0000000  0.334   12.31233  0.4431602   9.147622
##   0.0000000  0.445   12.31233  0.4431602   9.147622
##   0.0000000  0.556   12.31233  0.4431602   9.147622
##   0.0000000  0.667   12.31233  0.4431602   9.147622
##   0.0000000  0.778   12.31233  0.4431602   9.147622
##   0.0000000  0.889   12.31233  0.4431602   9.147622
##   0.0000000  1.000   12.31233  0.4431602   9.147622
##   0.1111111  0.001   14.29808  0.3814210  10.535194
##   0.1111111  0.112   14.29808  0.3814210  10.535194
##   0.1111111  0.223   14.29808  0.3814210  10.535194
##   0.1111111  0.334   14.29808  0.3814210  10.535194
##   0.1111111  0.445   14.29808  0.3814210  10.535194
##   0.1111111  0.556   14.29808  0.3814210  10.535194
##   0.1111111  0.667   14.29808  0.3814210  10.535194
##   0.1111111  0.778   14.29808  0.3814210  10.535194
##   0.1111111  0.889   14.28568  0.3819127  10.524225
##   0.1111111  1.000   14.14101  0.3859200  10.404528
##   0.2222222  0.001   14.73980  0.3709069  10.822783
##   0.2222222  0.112   14.73980  0.3709069  10.822783
##   0.2222222  0.223   14.73980  0.3709069  10.822783
##   0.2222222  0.334   14.73980  0.3709069  10.822783
##   0.2222222  0.445   14.72644  0.3713547  10.811106
##   0.2222222  0.556   14.38592  0.3784642  10.577146
```

```
##   0.2222222  0.667   14.08255  0.3860591  10.371998
##   0.2222222  0.778   13.80555  0.3939176  10.181108
##   0.2222222  0.889   13.57216  0.4012327  10.003985
##   0.2222222  1.000   13.36811  0.4089230   9.844763
##   0.3333333  0.001   14.97803  0.3666035  10.971702
##   0.3333333  0.112   14.97803  0.3666035  10.971702
##   0.3333333  0.223   14.97803  0.3666035  10.971702
##   0.3333333  0.334   14.82220  0.3685609  10.854659
##   0.3333333  0.445   14.28165  0.3795540  10.524794
##   0.3333333  0.556   13.85643  0.3919193  10.232558
##   0.3333333  0.667   13.54672  0.4022193   9.995031
##   0.3333333  0.778   13.30729  0.4115545   9.805560
##   0.3333333  0.889   13.07864  0.4230842   9.607716
##   0.3333333  1.000   12.86881  0.4344346   9.439407
##   0.4444444  0.001   15.18098  0.3627306  11.126218
##   0.4444444  0.112   15.18098  0.3627306  11.126218
##   0.4444444  0.223   15.16446  0.3632759  11.113714
##   0.4444444  0.334   14.43202  0.3758681  10.645697
##   0.4444444  0.445   13.85509  0.3931380  10.250502
##   0.4444444  0.556   13.49702  0.4048074   9.949912
##   0.4444444  0.667   13.18270  0.4192407   9.692523
##   0.4444444  0.778   12.92569  0.4322590   9.492677
##   0.4444444  0.889   12.69248  0.4452911   9.326797
##   0.4444444  1.000   12.49557  0.4565134   9.192260
##   0.5555556  0.001   15.38027  0.3571438  11.253830
##   0.5555556  0.112   15.38027  0.3571438  11.253830
##   0.5555556  0.223   14.92838  0.3665371  10.950649
##   0.5555556  0.334   14.09456  0.3866923  10.434859
##   0.5555556  0.445   13.55097  0.4045531  10.010319
##   0.5555556  0.556   13.18049  0.4204863   9.688119
##   0.5555556  0.667   12.88431  0.4353003   9.472541
##   0.5555556  0.778   12.61785  0.4503615   9.280810
##   0.5555556  0.889   12.36166  0.4658149   9.112841
##   0.5555556  1.000   12.13253  0.4814532   8.946283
##   0.6666667  0.001   15.51385  0.3539666  11.325831
##   0.6666667  0.112   15.51385  0.3539666  11.325831
##   0.6666667  0.223   14.65057  0.3714342  10.818456
##   0.6666667  0.334   13.78837  0.3976754  10.213996
##   0.6666667  0.445   13.28392  0.4171678   9.775089
##   0.6666667  0.556   12.92749  0.4338940   9.505579
##   0.6666667  0.667   12.60435  0.4518614   9.272020
##   0.6666667  0.778   12.30348  0.4706780   9.082397
##   0.6666667  0.889   12.04223  0.4885225   8.884303
##   0.6666667  1.000   11.87822  0.4998948   8.766096
##   0.7777778  0.001   15.60357  0.3522440  11.392215
##   0.7777778  0.112   15.60357  0.3522440  11.392215
##   0.7777778  0.223   14.39365  0.3794092  10.673320
##   0.7777778  0.334   13.55867  0.4069215  10.039148
##   0.7777778  0.445   13.05245  0.4284623   9.600043
##   0.7777778  0.556   12.68200  0.4481354   9.320817
##   0.7777778  0.667   12.33127  0.4695889   9.105113
##   0.7777778  0.778   12.02300  0.4905745   8.876726
##   0.7777778  0.889   11.85327  0.5027859   8.753623
##   0.7777778  1.000   11.78604  0.5092815   8.696414
```

```
##   0.8888889  0.001   15.72894  0.3481929  11.493979
##   0.8888889  0.112   15.71621  0.3486412  11.485005
##   0.8888889  0.223   14.18135  0.3866170  10.549292
##   0.8888889  0.334   13.34921  0.4164280   9.856949
##   0.8888889  0.445   12.84442  0.4402400   9.452996
##   0.8888889  0.556   12.45010  0.4623135   9.175780
##   0.8888889  0.667   12.07993  0.4875576   8.919758
##   0.8888889  0.778   11.86489  0.5026187   8.769087
##   0.8888889  0.889   11.80442  0.5089728   8.713218
##   0.8888889  1.000   11.78327  0.5118435   8.669018
##   1.0000000  0.001   16.00438  0.3412807  11.704506
##   1.0000000  0.112   15.82510  0.3441769  11.608467
##   1.0000000  0.223   14.03680  0.3900629  10.494429
##   1.0000000  0.334   13.14504  0.4260036   9.703104
##   1.0000000  0.445   12.64544  0.4509058   9.339230
##   1.0000000  0.556   12.26015  0.4757908   9.117754
##   1.0000000  0.667   11.99236  0.4950755   8.921916
##   1.0000000  0.778   11.92765  0.5025766   8.846203
##   1.0000000  0.889   11.90345  0.5059174   8.794142
##   1.0000000  1.000   11.90330  0.5078890   8.784984
##
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were alpha = 0.8888889 and lambda = 1.
```

To predict:

```
#To fit the model:
enet_model<- enet(x=train_x, y=train_y,
                  lambda = enet_tune$bestTune$lambda, normalize=TRUE)

#prediction:
enet_pred<- predict(enet_model, newx=test_x,
                    s = enet_tune$bestTune$lambda, mode = 'fraction',
                    type = 'fit') #s should be the same value as lambda

#R^2 analysis:
postResample(enet_pred$fit, test_y)
```

```
##       RMSE   Rsquared        MAE
## 15.8460228  0.5614875 11.7196570
```

This simple linear model is used as a benchmark to compare the performance for the PLS and ENET models. Both PLS and ENET out-performed the benchmark.

```
lm_model <- lm(permeability ~., data= train_data)
lm_pred <- predict(lm_model, newdata=as.data.frame(test_x))
```

```
## Warning in predict.lm(lm_model, newdata = as.data.frame(test_x)): prediction
## from rank-deficient fit; attr(*, "non-estim") has doubtful cases
```

```
postResample(lm_pred, test_y)
```

```
##       RMSE   Rsquared        MAE
## 30.3087993  0.2130654 20.4964880
```

(f) Would you recommend any of your models to replace the permeability laboratory experiment?

ANS: I would not recommend any of the models to replace the permeability laboratory experiment. This is because the R^2 and RMSE are showing moderate performance R^2« 0.8 and the RMSE is relatively large ~11. However, these models can serve as a guideline to narrow down the experiment parameters where the desired permeability is found in these models. For example, if permeability of substance X is predicted to be 10x by A and B predictors. Researchers can design experiments with more emphasis on A and B predictors to verify the results. This is a more guided approach than just random guessing which predictor will work influence the response variable.

6.3 a) Start R and use these commands to load the data

```
library(AppliedPredictiveModeling)
data(ChemicalManufacturingProcess)
```

(b) A small percentage of cells in the predictor set contain missing values. Use an imputation function to fill in these missing values (e.g., see Sect.3.8).

```
set.seed(555)
#define a shorter name:
chem<- ChemicalManufacturingProcess

#See which column has NA values:
NA_values<- chem %>% is.na() %>% sum() #106

#To impute with the bagged tree method:
preprocess_chem<- preProcess(chem, method=c('scale','center','bagImpute'))

#applies the imputed values to the preprocessed dataset
imputed_chem<- predict(preprocess_chem, chem)

#check for NA values:
more_NA_values<- imputed_chem %>% is.na() %>% sum() #0

print(paste('Number of NA in the df:', NA_values))
```

```
## [1] "Number of NA in the df: 106"
```

```
print(paste('Number of NA in the df after imputation:', more_NA_values))
```

```
## [1] "Number of NA in the df after imputation: 0"
```

(c) Split the data into a training and a test set, pre-process the data, and tune a model of your choice from this chapter. What is the optimal value of the performance metric?

ANS: the Ncomponent for the PLS model is 3. RMSE = 0.56, and R^2 = 0.65

```
set.seed(41)
#define predictor and response
chem_predictors<- imputed_chem %>% select(-Yield)
chem_response<- imputed_chem$Yield

#Since the data is already pre-processed during imputation, we can go straight to splitting:

#splitting training data:
chem_train_idx<- createDataPartition(chem_response, p = 0.7, list=FALSE)

#the response argument must be a vector, not a dataframe
chem_train_x<- chem_predictors[chem_train_idx,]
chem_train_y<- chem_response[chem_train_idx]

chem_test_x<- chem_predictors[-chem_train_idx,]
chem_test_y<- chem_response[-chem_train_idx]
```

Tuning for PLS: ncomp = 3

```
set.seed(10)
chem_pls_tune<- train(x=chem_train_x, y=chem_train_y,
                      method = 'pls',
                      tuneLength = 5, #the tunelength was determined after visualizing plot(chem_pls_tu
                      trControl = trainControl(method = 'cv', number=10)
                      )
print(chem_pls_tune)
```

```
## Partial Least Squares
##
## 124 samples
##  57 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 112, 112, 110, 112, 112, 112, ...
## Resampling results across tuning parameters:
##
##   ncomp  RMSE       Rsquared   MAE
##   1      0.7085624  0.4761698  0.5831283
##   2      0.6144362  0.5958794  0.5045485
##   3      0.5675716  0.6518523  0.4693912
##   4      0.5699195  0.6524466  0.4718168
##   5      0.5688760  0.6620834  0.4644194
##
## RMSE was used to select the optimal model using the smallest value.
## The final value used for the model was ncomp = 3.
```

(d) Predict the response for the test set. What is the value of the performance metric and how does this compare with the resampled performance metric on the training set?

ANS: Comparing to the training set values: RMSE = 0.56, and R^2 = 0.65, the predicted values are RMSE = 0.73, R^2 = 0.57. These values are expected to be lower than the training but they do not fall short by too much. This model is still fairly capable.

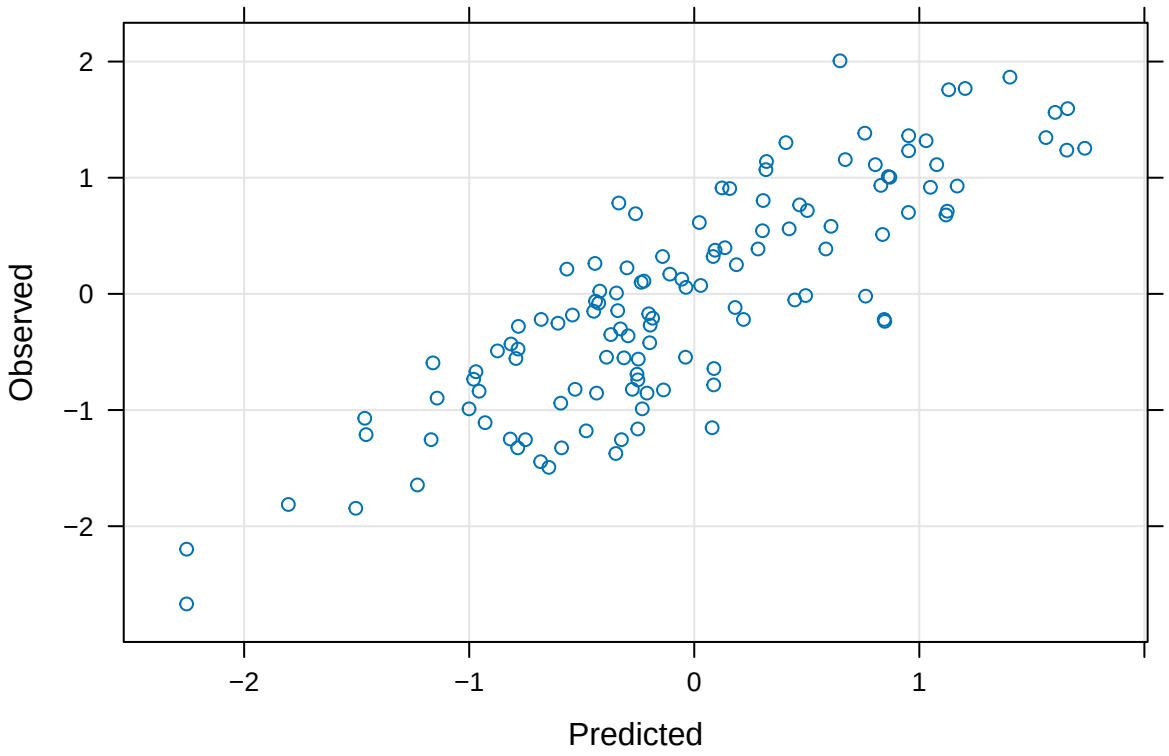
To fit and predict:

```
chem_pred<- predict(chem_pls_tune, chem_test_x)
postResample(chem_pred, chem_test_y)
```

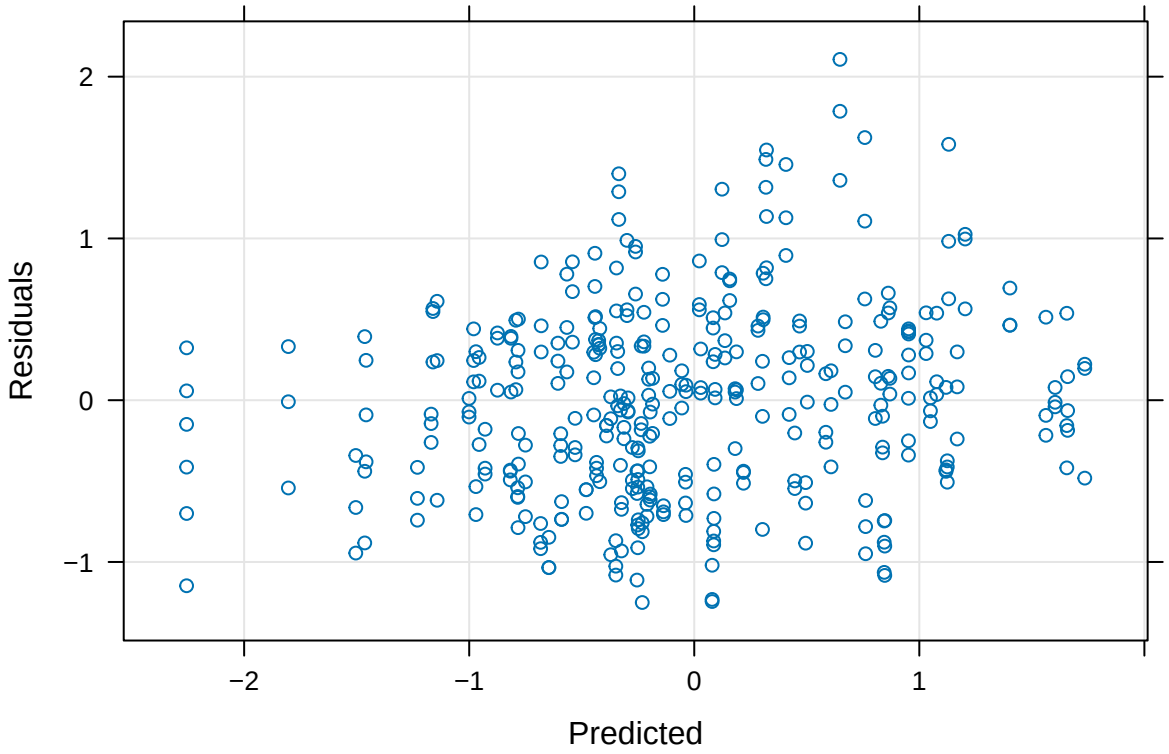
```
##      RMSE  Rsquared       MAE
## 0.7328591 0.5657281 0.5909947
```

Diagnostic plots:

```
xyplot(chem_train_y ~ predict(chem_pls_tune),
       type = c('p', 'g'),
       xlab='Predicted', ylab='Observed')
```



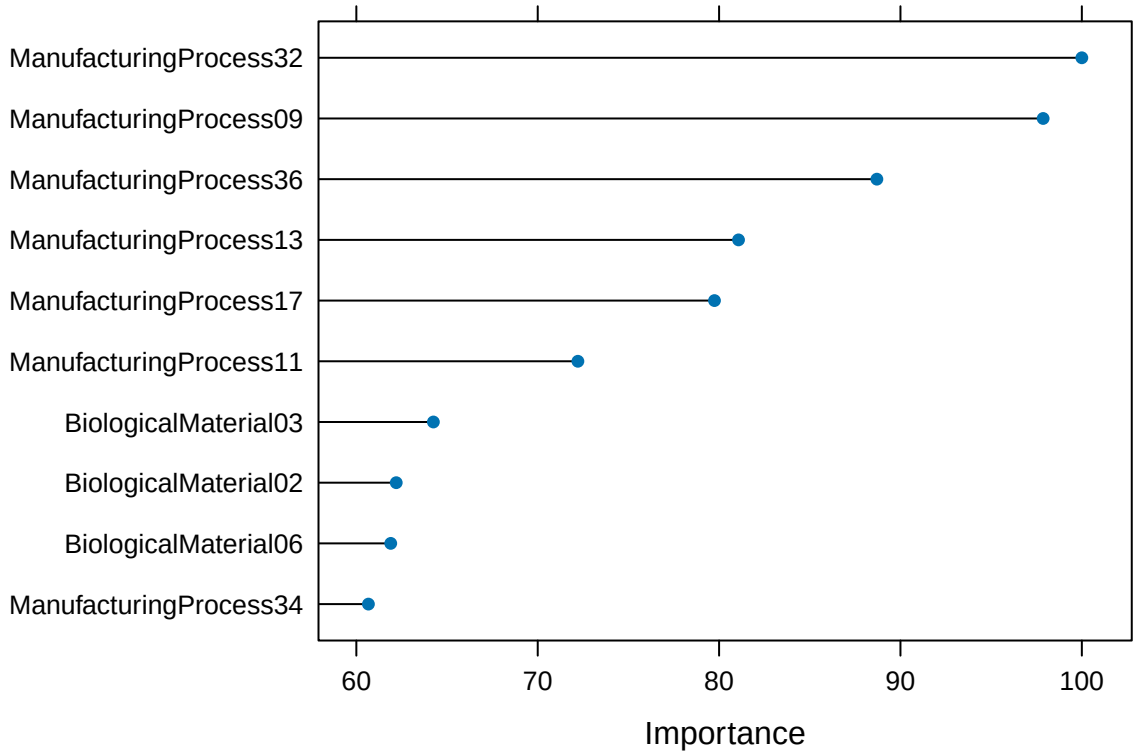

```
xyplot(resid(chem_pls_tune) ~ predict(chem_pls_tune),
       type = c('p', 'g'),
       xlab='Predicted', ylab='Residuals')
```



(e) Which predictors are most important in the model you have trained? Do either the biological or process predictors dominate the list?

ANS: Manufacturing Process seems to be the bigger contributors. 7/10 of the largest contributors are of manufacturing.

```
#to visualize it:
ranking<- varImp(chem_pls_tune) #use on the tune(AKA.fitted model) parameters
plot(ranking, top = 10)
```



(f) Explore the relationships between each of the top predictors and the response. How could this information be helpful in improving yield in future runs of the manufacturing process?

ANS: Since the coefficients of the top predictors play a direct contribution to the Yield in a per-unit-change basis, the predictors with high coefficients such as ManufacturingProcess17 should be heavily increased in the next production and the predictors with the negative coefficients should be reduced to help boosting the production.

```
#Extract the names of the top predictors:
top_predictor_names<- ranking$importance %>%
  as.data.frame() %>%
  rownames_to_column(var='predictor') %>%
  arrange(desc(Overall)) %>%
  slice_max(Overall, n = 10) %>%
  pull(predictor)

#select the best Ncomp number:
coef(chem_pls_tune$finalModel) %>%
  as.data.frame() %>%
  rownames_to_column('predictor') %>%
  rename('coefficients'='.outcome.3 comps') %>%
  filter(predictor %in% top_predictor_names) %>%
  arrange(desc(coefficients))
```

```
##                 predictor coefficients
## 1  ManufacturingProcess32   0.21023200
## 2  ManufacturingProcess34   0.16045996
## 3  ManufacturingProcess09   0.13885179
## 4    BiologicalMaterial03   0.09033525
## 5  ManufacturingProcess11   0.08768017
## 6    BiologicalMaterial06   0.06893221
## 7    BiologicalMaterial02   0.04963839
## 8  ManufacturingProcess13  -0.08932003
## 9  ManufacturingProcess17  -0.11749600
## 10 ManufacturingProcess36  -0.16386947
```

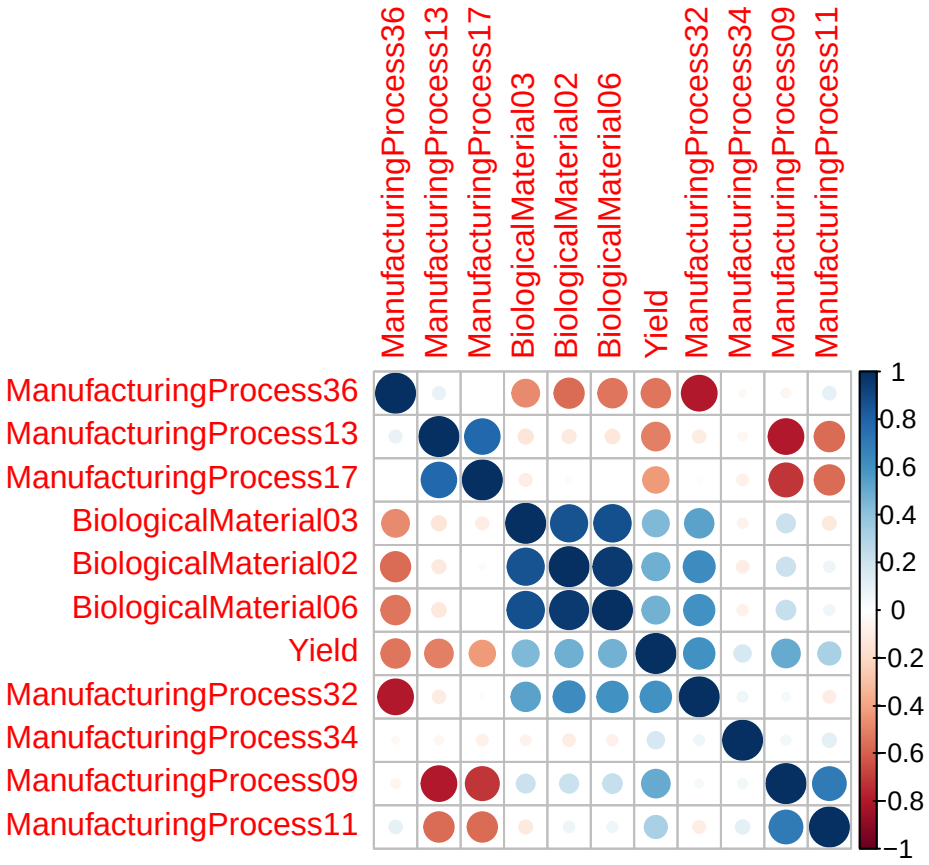
We can also check the correlations between all the predictors and the response variable visually

```
#Select only the columns with the top predictors after the imputation:
top_predictors<- imputed_chem %>%
  select(c(Yield, top_predictor_names))
```

```
## Warning: Using an external vector in selections was deprecated in tidyselect 1.1.0.
## i Please use ‘all_of()‘ or ‘any_of()‘ instead.
##   # Was:
##   data %>% select(top_predictor_names)
##
##   # Now:
##   data %>% select(all_of(top_predictor_names))
##
## See <https://tidyselect.r-lib.org/reference/faq-external-vector.html>.
## This warning is displayed once every 8 hours.
## Call ‘lifecycle::last_lifecycle_warnings()‘ to see where this warning was
## generated.
```

```
#Compute the correlations between the top ten predictors:
correlations<- cor(top_predictors)

#To visualize the correlations between the predictors:
corrplot(correlations, order = 'hclust')
```




```
pca_model<- train(
  x= chem_train_x,
  y=chem_train_y,
  method='glm',
  trControl = trainControl(method='cv', number=10),
  preProcess = c('center', 'scale', 'pca')
)
chem_pp<- predict(pca_model, chem_test_x)
postResample(chem_pp, chem_test_y)
```

```
##      RMSE  Rsquared       MAE
## 0.8606420 0.4075477 0.6560157
```