ASSIGNMENT 1

MIRABELLA SIMONE

25/09/2021

PROBLEM 1

I will start working on some “vanilla data” to construct the functions I will need in the last exercise, to test them and to prove important results that will be useful after.

First, I need to create my dataset. I obtain the two predictors x1 and x2 from a uniform distribution with values between 0 and 1. With them, I can now obtain the correspondent values on the Y, which is the response variable. The relation between Y and the predictors is set to be the Franke function, obtained as follows. A stochastic noise with a normal distribution is also implemented to obtain Y in order to have more realistic data.

```
set.seed(29012001)

x1 <- cbind(runif(100, 0, 1))
x2 <- cbind(runif(100, 0, 1))
X <- as.data.frame(cbind(x1,x2))

term1 <- 0.75 * exp(-(9*x1-2)^2/4 - (9*x2-2)^2/4)
term2 <- 0.75 * exp(-(9*x1+1)^2/49 - (9*x2+1)/10)
term3 <- 0.5 * exp(-(9*x1-7)^2/4 - (9*x2-3)^2/4)
term4 <- -0.2 * exp(-(9*x1-4)^2 - (9*x2-7)^2)

y <- cbind(term1 + term2 + term3 + term4 + 0.001*rnorm(100))

dataset <- cbind(X,y)
head(dataset)
```

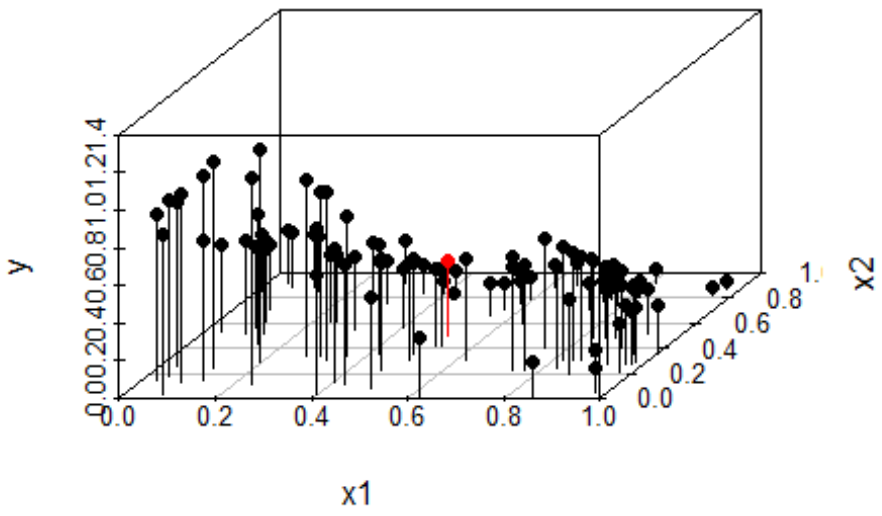
```
##          V1        V2         y
## 1 0.1164997 0.2348432 1.0991257
## 2 0.9950253 0.3747397 0.2479976
## 3 0.7608584 0.7865335 0.1013970
## 4 0.2812785 0.3299443 0.9452897
## 5 0.6513324 0.9081851 0.1146994
## 6 0.1277534 0.1405091 1.0886299
```

x1 and x2 come from the same distribution and have the same order of magnitude, and they are already scaled between 0 and 1 by their nature, so there is no point in scaling them. (Anyway, a scaling function will be implemented later when I’ll work with the Ridge regression).

The same applies for Y, which directly comes from x1 and x2. If they were not scaled though, I would have needed to standardize them. This is true because, mostly when using polynomial regression with interaction terms, the model might present multicollinearity, which would lead to incorrect estimates of the coefficients and general misleading results. It could be possible to compare two models - one obtained from unstandardized variables and the other with standardized ones - using the VIF (variance inflation factor), which measures how much the behavior (variance) of predictor is influenced, or inflated, by its correlation with the other predictors.

We can plot the data contained in the dataset to see how they are disposed in the 3d plane.

```
require(scatterplot3d);
graph<-scatterplot3d(x1, x2, y, xlab="x1", ylab="x2",
                          zlab="y", type="h", highlight.3d=FALSE,
                          col.axis="black",col.grid="gray",
                          angle=60, pch=16)
graph$points3d(mean(x1), mean(x2), mean(y),
                    col="red", type="h", pch=16)
```



I create here a matrix to collect the values of the MSE and R2 that I will calculate for each order of the polynomials:

```
results <- matrix(data=NA, nrow=2,ncol=5)
rownames(results)<-cbind("MSE","R2")
names<-c()
for (i in 1:5){
  names[i] <- paste("order", as.character(i))
}
colnames(results) <- names
```

To perform a standard least square regression analysis using polynomials in x1 and x2 up to 5th order, I now create all the functions that I will need:

```
model_matrix <- function(x,y,n){
  l <- length(x)
  if (n==1){
    X <- cbind(rep(1,l),x,y)
  }
  else if (n==2){
    X <- cbind(rep(1,l),x,y,x**2,y**2,x*y)
  }
  else if (n==3){
    X <- cbind(rep(1,l),x , y , x**2 , y**2 , x*y , x**3 , y**3, (x**2
)*y, x*(y**2))
  }
  else if (n==4){
    X <- cbind(rep(1,l),x , y , x**2 , y**2 , x*y , x**3 , y**3, (x**2
)*y, x*(y**2), (x**2)*(y**2),x**4, y**4,(x**3)*y,x*(y**3))
  }
  else if (n==5){
    X <- cbind(rep(1,l),x , y , x**2 , y**2 , x*y , x**3 , y**3, (x**2
)*y, x*(y**2), (x**2)*(y**2),x**4, y**4,(x**3)*y,x*(y**3),x**5, y**5,(
x**3)*(y**2),(x**2)*(y**3))
  }
return(X)
}

betas <- function(matX,vec_y){
  beta <- solve(t(matX) %*% matX) %*% t(matX) %*% vec_y
  return(beta)
}

predictions <- function(x,y,n,beta){
  if (n==1){
    prediction <- beta[1] + beta[2]*x + beta[3]*y
  }
```

```
  else if (n==2){
    prediction <- beta[1] + beta[2]* x + beta[3]* y + beta[4]* x**2 +
beta[5]* y**2 + beta[6]* x*y
  }
  else if (n==3){
    prediction <- beta[1] + beta[2]* x + beta[3]* y + beta[4]* x**2 +
beta[5]* y**2 + beta[6]* x*y + beta[7]* x**3 + beta[8]* y**3 + beta[9]
* (x**2)*y + beta[10]* x*(y**2)
  }
  else if (n==4){
    prediction <- beta[1] + beta[2]* x + beta[3]* y + beta[4]* x**2 +
beta[5]* y**2 + beta[6]* x*y + beta[7]* x**3 + beta[8]* y**3 + beta[9]
* (x**2)*y + beta[10]* x*(y**2) + beta[11]* (x**2)*(y**2) + beta[12]*
x**4 + beta[13]* y**4 + beta[14]* (x**3)*y+ beta[15]* x*(y**3)
  }
  else if (n==5){
    prediction <- beta[1] + beta[2]* x + beta[3]* y + beta[4]* x**2 +
beta[5]* y**2 + beta[6]* x*y + beta[7]* x**3 + beta[8]* y**3 + beta[9]
* (x**2)*y + beta[10]* x*(y**2) + beta[11]* (x**2)*(y**2) + beta[12]*
x**4 + beta[13]* y**4 + beta[14]* (x**3)*y+ beta[15]* x*(y**3) + beta[
16]* x**5 + beta[17]* y**5 + beta[18]* (x**3)*(y**2) + beta[19]* (x**2
)*(y**3)
  }

  return(prediction)

}
```

For the train-test split function I decided to split the data using 80% of the observation to make the training set, and the remaining 20% for the test set:

```
train_test <- function(X,y){
  set.seed(29012001)
  trRowIndex <- sample(1:nrow(X), 0.8*nrow(X))

  trdataX <- X[trRowIndex, ]
  trdataY <- y[trRowIndex, ]

  tedataX <- X[-trRowIndex, ]
  tedataY <- y[-trRowIndex, ]

  Train_Test <- list(trdataX, trdataY, tedataX, tedataY)

  return(Train_Test)
}
```

```r
# this function will be useful to extract the indexes of a matrix

translate <- function(vector){
  v <- c()
  for (i in 1:length(vector)){
    v[i] <- toString(vector[i])
  }
  return(v)
}
```

Let's start:

```r
splitted <- train_test(X,y)

trainingX <- splitted[[1]]
trainingY <- splitted[[2]]
testX <- splitted[[3]]
testY <- splitted[[4]]

matrix_deg1 <- model_matrix(trainingX[,1],trainingX[,2],1)
rownames(matrix_deg1) <- translate(rownames(trainingX))
testX_1 <- as.data.frame(model_matrix(testX$V1,testX$V2,1))
beta_deg1 <- betas(matrix_deg1, trainingY)
pred_deg1 <- predictions(testX[,1],testX[,2],1,beta_deg1)
mse_deg1 <- mean((testY - pred_deg1)^2)
r2_deg1 <- 1 - sum((testY - pred_deg1)^2)/sum((testY - mean(testY))^2)


matrix_deg2 <- model_matrix(trainingX[,1],trainingX[,2],2)
rownames(matrix_deg2) <- translate(rownames(trainingX))
testX_2 <- as.data.frame(model_matrix(testX$V1,testX$V2,2))
beta_deg2 <- betas(matrix_deg2, trainingY)
pred_deg2 <- predictions(testX[,1],testX[,2],2,beta_deg2)
mse_deg2 <- mean((testY - pred_deg2)^2)
r2_deg2 <- 1 - sum((testY - pred_deg2)^2)/sum((testY - mean(testY))^2)


matrix_deg3 <- model_matrix(trainingX[,1],trainingX[,2],3)
rownames(matrix_deg3) <- translate(rownames(trainingX))
testX_3 <- as.data.frame(model_matrix(testX$V1,testX$V2,3))
beta_deg3 <- betas(matrix_deg3, trainingY)
pred_deg3 <- predictions(testX[,1],testX[,2],3,beta_deg3)
mse_deg3 <- mean((testY - pred_deg3)^2)
r2_deg3 <- 1 - sum((testY - pred_deg3)^2)/sum((testY - mean(testY)^2))
```

```
matrix_deg4 <- model_matrix(trainingX[,1],trainingX[,2],4)
rownames(matrix_deg4) <- translate(rownames(trainingX))
testX_4 <- as.data.frame(model_matrix(testX$V1,testX$V2,4))
beta_deg4 <- betas(matrix_deg4, trainingY)
pred_deg4 <- predictions(testX[,1],testX[,2],4,beta_deg4)
mse_deg4 <- mean((testY - pred_deg4)^2)
r2_deg4 <- 1 - sum((testY - pred_deg4)^2)/sum((testY - mean(testY))^2)


matrix_deg5 <- model_matrix(trainingX[,1],trainingX[,2],5)
rownames(matrix_deg5) <- translate(rownames(trainingX))
testX_5 <- as.data.frame(model_matrix(testX$V1,testX$V2,5))
beta_deg5 <- betas(matrix_deg5, trainingY)
pred_deg5 <- predictions(testX[,1],testX[,2],5,beta_deg5)
mse_deg5 <- mean((testY - pred_deg5)^2)
r2_deg5 <- 1 - sum((testY - pred_deg5)^2)/sum((testY - mean(testY))^2)

results[1,] <- rbind(mse_deg1, mse_deg2, mse_deg3, mse_deg4, mse_deg5)
results[2,] <- rbind(r2_deg1, r2_deg2, r2_deg3, r2_deg4, r2_deg5)

results

##         order 1    order 2     order 3    order 4    order 5
## MSE 0.01321887 0.01737684 0.004679752 0.03612006 0.03218911
## R2  0.80055862 0.73782457 0.978825514 0.45503388 0.51434256
```

It is possible to see here that the best model among those five appears to be the 3rd one, since it is the model which gives values of the predictions for the test data that are the closest to the observed values. It translates to the lowest MSE, which means, the lowest error during predictions. In the same way, the R^2 for the 3rd model is the highest, and this means that the 3rd model is the one that fits the data the best, almost 97% of them. It is important to check how much the noise affects the results of the fit.

In the model above, I set a coefficient for the noise of 0.001, which is a value that is not high. If I had put a value of 0.5 for example, the results of the MSEs and R2s would have given a different situation, as follows:

```
y2 <- cbind(term1 + term2 + term3 + term4 + 0.5*rnorm(100))

results_2 <- matrix(data=NA, nrow=2,ncol=5)
rownames(results_2)<-cbind("MSE","R2")
names<-c()
for (i in 1:5){
  names[i] <- paste("order", as.character(i))
}
```

```
colnames(results_2) <- names

splitted_2 <- train_test(X,y2)

matrix_deg1_2 <- model_matrix(splitted_2[[1]][,1],splitted_2[[1]][,2],
1);
beta_deg1_2 <- betas(matrix_deg1_2, splitted_2[[2]])
pred_deg1_2 <- predictions(splitted_2[[3]][,1],splitted_2[[3]][,2],1,b
eta_deg1_2)
mse_deg1_2 <- mean((splitted_2[[4]] - pred_deg1_2)^2)
r2_deg1_2 <- 1 - sum((splitted_2[[4]] - pred_deg1_2)^2)/sum((splitted_
2[[4]] - mean(splitted_2[[4]]))^2)


matrix_deg2_2 <- model_matrix(splitted_2[[1]][,1],splitted_2[[1]][,2],
2);
beta_deg2_2 <- betas(matrix_deg2_2, splitted_2[[2]])
pred_deg2_2 <- predictions(splitted_2[[3]][,1],splitted_2[[3]][,2],2,b
eta_deg2_2)
mse_deg2_2 <- mean((splitted_2[[4]] - pred_deg2_2)^2)
r2_deg2_2 <- 1 - sum((splitted_2[[4]] - pred_deg2_2)^2)/sum((splitted_
2[[4]] - mean(splitted_2[[4]]))^2)


matrix_deg3_2 <- model_matrix(splitted_2[[1]][,1],splitted_2[[1]][,2],
3);
beta_deg3_2 <- betas(matrix_deg3_2, splitted_2[[2]])
pred_deg3_2 <- predictions(splitted_2[[3]][,1],splitted_2[[3]][,2],3,b
eta_deg3_2)
mse_deg3_2 <- mean((splitted_2[[4]] - pred_deg3_2)^2)
r2_deg3_2 <- 1 - sum((splitted_2[[4]] - pred_deg3_2)^2)/sum((splitted_
2[[4]] - mean(splitted_2[[4]]))^2)


matrix_deg4_2 <- model_matrix(splitted_2[[1]][,1],splitted_2[[1]][,2],
4);
beta_deg4_2 <- betas(matrix_deg4_2, splitted_2[[2]])
pred_deg4_2 <- predictions(splitted_2[[3]][,1],splitted_2[[3]][,2],4,b
eta_deg4_2)
mse_deg4_2 <- mean((splitted_2[[4]] - pred_deg4_2)^2)
r2_deg4_2 <- 1 - sum((splitted_2[[4]] - pred_deg4_2)^2)/sum((splitted_
2[[4]] - mean(splitted_2[[4]]))^2)


matrix_deg5_2 <- model_matrix(splitted_2[[1]][,1],splitted_2[[1]][,2],
5);
```

```
beta_deg5_2 <- betas(matrix_deg5_2, splitted_2[[2]])
pred_deg5_2 <- predictions(splitted_2[[3]][,1],splitted_2[[3]][,2],5,b
eta_deg5_2)
mse_deg5_2 <- mean((splitted_2[[4]] - pred_deg5_2)^2)
r2_deg5_2 <- 1 - sum((splitted_2[[4]] - pred_deg5_2)^2)/sum((splitted_
2[[4]] - mean(splitted_2[[4]]))^2)

results_2[1,] <- rbind(mse_deg1_2, mse_deg2_2, mse_deg3_2, mse_deg4_2,
mse_deg5_2)
results_2[2,] <- rbind(r2_deg1_2, r2_deg2_2, r2_deg3_2, r2_deg4_2, r2_
deg5_2)

results_2
```

```
##       order 1   order 2    order 3    order 4    order 5
## MSE 0.1871913 0.2149270 0.24733928  0.3601995  0.5668137
## R2  0.2553131 0.1449748 0.01603184 -0.4329499 -1.2549053
```

As we can see here, with a higher noise, the model with the lowest MSE is now the first model, which is the one with the highest R2 as well. In this case we can also point out that models 4 and 5 fit the model even worse than the null model (straight horizontal line), which is why they give a negative R2.

That said, the noise really affects the accuracy of the model, so we should be careful in choosing a right coefficient for it that might be plausible. We hence must consider the magnitude of the data, which here ranges between 0 and 1: it makes no sense to multiply the noise by 0.5 because it affects the data too much

Let's now get back to the case 1 with the noise being multiplied by 0.001.

I can now obtain the confidence intervals for the coefficients' estimators along with their standard errors. The intervals are a useful tool to evaluate test hypothesis. Here, the null hypothesis is stating that the coefficients should be, one at the time, equal to zero. If the intervals contain the value 0, we cannot reject the null hypothesis. On the other hand, we will state that, with a confidence of 95% (standard case, from which z = 1.96) we can say that the coefficient is not zero and, therefore, it has to be kept in the model (and its correspondent variable as well).

```
standard_errors <- function(trainY, matX, beta){
  dSigmaSq <- sum((trainY - matX%*%beta)^2)/(nrow(matX)-ncol(matX))
mVarCovar <- dSigmaSq*chol2inv(chol(t(matX)%*%matX))
vStdErr <- sqrt(diag(mVarCovar))
print(cbind(beta, vStdErr))
return(vStdErr)}

confidence_int <- function(beta, se){
  c <- matrix(data = NA, nrow = length(beta), ncol = 2)
```

```
  colnames(c) <- c("Lower","Upper")
  for (i in 1:length(beta)){
    interval = cbind(beta[i]-1.96*se[i], beta[i]+1.96*se[i])
    c[i,] <- interval
  }
return(c)}

se_1 <- standard_errors(trainingY, matrix_deg1, beta_deg1)

##                 vStdErr
##    1.0264560 0.04728087
## x -0.5015665 0.05350627
## y -0.7241899 0.06481411

confidence_int(beta_deg1, se_1)

##           Lower      Upper
## [1,]  0.9337855  1.1191265
## [2,] -0.6064388 -0.3966942
## [3,] -0.8512255 -0.5971542

se_2 <- standard_errors(trainingY, matrix_deg2, beta_deg2)

##                  vStdErr
##    1.11805536 0.07912657
## x -0.96368615 0.23040515
## y -0.46367261 0.25243649
##    0.07430003 0.19119800
##   -0.71137078 0.22011793
##    0.87406605 0.18492392

confidence_int(beta_deg2, se_2)

##           Lower       Upper
## [1,]  0.9629673  1.27314343
## [2,] -1.4152802 -0.51209206
## [3,] -0.9584481  0.03110291
## [4,] -0.3004481  0.44904811
## [5,] -1.1428019 -0.27993964
## [6,]  0.5116152  1.23651693

se_3 <- standard_errors(trainingY, matrix_deg3, beta_deg3)

##                vStdErr
##    0.9710849 0.1073372
## x -0.6260508 0.5160952
## y  1.3151183 0.6042759
##   -0.6362405 1.0000091
##   -6.4938009 1.1414333
```

```
##    1.3815866 0.9482740
##    0.1716163 0.5990258
##    4.7153959 0.7116044
##    0.8394775 0.6305289
##   -1.3725439 0.5955350

confidence_int(beta_deg3, se_3)

##            Lower      Upper
##  [1,]  0.7607040  1.1814658
##  [2,] -1.6375975  0.3854958
##  [3,]  0.1307375  2.4994990
##  [4,] -2.5962583  1.3237772
##  [5,] -8.7310101 -4.2565917
##  [6,] -0.4770305  3.2402037
##  [7,] -1.0024743  1.3457068
##  [8,]  3.3206513  6.1101405
##  [9,] -0.3963592  2.0753142
## [10,] -2.5397925 -0.2052953

se_4 <- standard_errors(trainingY, matrix_deg4, beta_deg4)

##                vStdErr
##     0.5320875 0.1378210
## x   6.0457690 0.8258927
## y   2.6717602 1.1552914
##   -26.2107713 2.5585446
##    -8.6116775 3.2863686
##    -6.1554710 2.6703478
##    34.9258650 3.2405368
##     4.8675209 4.0464100
##    15.1765906 3.4293046
##     5.3091589 3.6540732
##    -2.2921563 1.6515172
##   -15.3579959 1.4314885
##     1.2021573 1.8929099
##    -9.2679626 1.8296998
##    -3.7434477 1.7805362

confidence_int(beta_deg4, se_4)

##             Lower       Upper
##  [1,]   0.2619583   0.8022167
##  [2,]   4.4270194   7.6645186
##  [3,]   0.4073890   4.9361314
##  [4,] -31.2255187 -21.1960239
##  [5,] -15.0529599  -2.1703951
```

```
##  [6,] -11.3893528  -0.9215892
##  [7,]  28.5744128  41.2773172
##  [8,]  -3.0634426  12.7984844
##  [9,]   8.4551535  21.8980276
## [10,]  -1.8528246  12.4711425
## [11,]  -5.5291300   0.9448174
## [12,] -18.1637133 -12.5522784
## [13,]  -2.5079460   4.9122606
## [14,] -12.8541741  -5.6817510
## [15,]  -7.2332986  -0.2535967
```

```
se_5 <- standard_errors(trainingY, matrix_deg5, beta_deg5)
```

```
##                   vStdErr
##     0.9027233  0.1715577
## x   5.2750530  1.1245226
## y  -2.7865074  1.7764781
##   -24.8013858  4.7799793
##    20.2730675  7.5299185
##     0.2745816  5.9612555
##    29.7012873 11.4029354
##   -65.0679800 17.0772697
##    13.0983177  9.8088219
##   -11.3311045 12.0097441
##     8.2891421 14.3517462
##    -7.5998424 12.6428711
##    79.8073601 18.9243753
##   -11.7396816  5.4766093
##     7.8365066  7.6145278
##    -3.4301565  5.0593002
##   -33.0245656  7.9063616
##     1.7739017  5.7807661
##    -8.7479676  6.9485216
```

```
confidence_int(beta_deg5, se_5)
```

```
##              Lower       Upper
##  [1,]   0.5664703   1.2389763
##  [2,]   3.0709886   7.4791174
##  [3,]  -6.2684044   0.6953896
##  [4,] -34.1701452 -15.4326263
##  [5,]   5.5144273  35.0317077
##  [6,] -11.4094791  11.9586423
##  [7,]   7.3515339  52.0510406
##  [8,] -98.5394287 -31.5965313
##  [9,]  -6.1269733  32.3236086
## [10,] -34.8702030  12.2079939
```

```
## [11,] -19.8402805  36.4185648
## [12,] -32.3798698  17.1801850
## [13,]  42.7155845 116.8991358
## [14,] -22.4738360  -1.0055273
## [15,]  -7.0879679  22.7609811
## [16,] -13.3463849   6.4860719
## [17,] -48.5210343 -17.5280969
## [18,]  -9.5563998  13.1042032
## [19,] -22.3670700   4.8711347
```

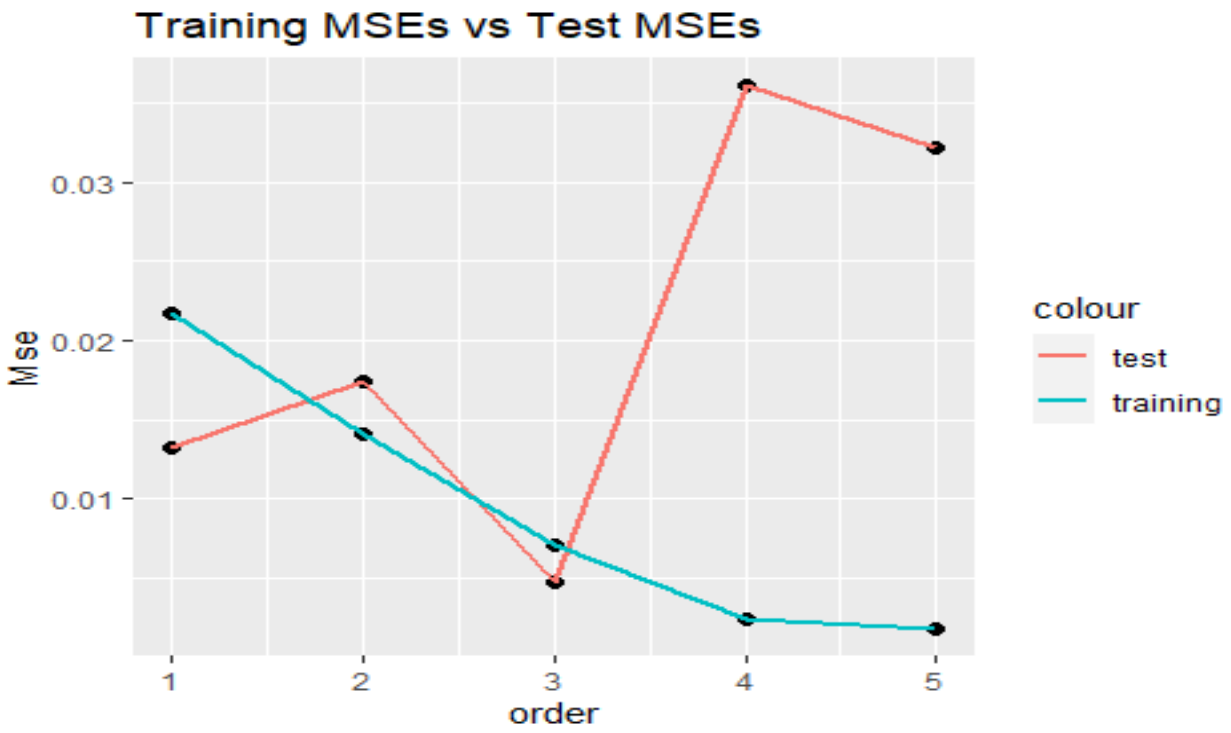
We can see up above the estimate of the coefficients, their standard errors and corresponding confidence intervals. With a closer look we could for example notice that the interval for the intercept always has positive lower and upper bounds, which means that we can say with a confidence of 95% that the intercept is never equal to zero. As we can further notice, as the order of the polynomial increase, the ranges of the intervals get bigger, and sometimes they have a negative lower bound and a positive upper bound. That is, the interval ends up containing the value 0, which is equally likely as the other values in the interval (by definition of confidence interval). When this happens, we don't have enough empirical evidence to say, with a confidence of 95%, that that coefficient is different from 0. With a closer look to the polynomial of degree 3 (which was the optimal one in terms of MSE and R2) this happens for example for the coefficients corresponding to x_1, x_1^2, x_1x_2, x_1^3 and $x_1^2x_2$.

PROBLEM 2

I am asked to make a figure which displays the test and training MSEs and to study their behavior as the complexity of the model (its order) increases. First, I need to calculate the MSEs for the training data, since I already have calculated the MSEs test.

```
pred_deg_train1 <- predictions(trainingX[,1],trainingX[,2],1,beta_deg1
)
mse_deg_train1 <- mean((trainingY - pred_deg_train1)^2)

pred_deg_train2 <- predictions(trainingX[,1],trainingX[,2],2,beta_deg2
)
mse_deg_train2 <- mean((trainingY - pred_deg_train2)^2)

pred_deg_train3 <- predictions(trainingX[,1],trainingX[,2],3,beta_deg3
)
mse_deg_train3 <- mean((trainingY - pred_deg_train3)^2)

pred_deg_train4 <- predictions(trainingX[,1],trainingX[,2],4,beta_deg4
)
mse_deg_train4 <- mean((trainingY - pred_deg_train4)^2)

pred_deg_train5 <- predictions(trainingX[,1],trainingX[,2],5,beta_deg5
```

```
)
mse_deg_train5 <- mean((trainingY - pred_deg_train5)^2)

msematrix <- data.frame(1:5, rbind(mse_deg1,mse_deg2,mse_deg3,mse_deg4
,mse_deg5),rbind(mse_deg_train1,mse_deg_train2,mse_deg_train3,mse_deg_
train4,mse_deg_train5))
colnames(msematrix) <- c("order", "test", "train")
rownames(msematrix) <- c(1:5)
msematrix

##   order        test       train
## 1     1 0.013218866 0.021719496
## 2     2 0.017376845 0.014055474
## 3     3 0.004679752 0.007077122
## 4     4 0.036120057 0.002344022
## 5     5 0.032189111 0.001776079

require(ggplot2)

ggplot()+
  geom_point(data=msematrix, aes(x=order, y=test), size=2)+
  geom_line(data=msematrix, aes(x=order, y=test, colour="test"), size=
1)+
  geom_point(data=msematrix, aes(x=order, y=train), size=2)+
  geom_line(data=msematrix, aes(x=order, y=train, colour="training"),
size=1)+
  ylab("Mse")+ggtitle("Training MSEs vs Test MSEs")
```



From the plot above, we can see that as the order of the polynomial increases, the MSE in the training set monotonically decreases. In fact, the model tends to overfit the data, giving predictions very close to the real observed values, which leads to smaller and smaller MSEs (0.00177 for order 5). On the other side, for the MSEs on the test data, we can see that it reaches its lowest value on the 3rd order, which is, as shown before, the one that gives the best results in terms of both MSE and R2. After that, the MSE increases since the model starts to overfit the data and therefore the predictions $\hat{Y}_i$ on the test set result to be far from the observed values Y_i. (A deeper explanation comes after the decomposition of the MSE).

In general, from the model $y = f(x) + \epsilon$ with $\epsilon \sim N(0, \sigma^2)$ we can estimate f in terms of β and the design matrix X, that is, $\tilde{y} = X\hat{\beta}$, where $\hat{\beta}$minimizes the Loss / Cost function

$$C(X, \beta) = \frac{1}{n}\Sigma_{i=0}^{n-1}(y_i - \tilde{y}_i)^2 = E[(y - \tilde{y})^2]$$

(Where $E[(y - \tilde{y})^2]$ is defined as the Mean Squared Error, or $MSE(y, \tilde{y})$).

We can see that: $E[(y - \tilde{y})^2] = E\left[\left(f(x) + \epsilon - \hat{f}(x)\right)^2\right] = E\left[(f(x)^2 + \epsilon^2 + \hat{f}(x)^2 + 2\epsilon f(x) - 2f(x)\hat{f}(x) - 2\epsilon\hat{f}(x)\right]$

before moving on, we know that $E[\epsilon] = 0$ and that $V(\epsilon) = E[\epsilon^2] - E[\epsilon]^2$ thus it follows that $V(\epsilon) = E[\epsilon^2] = \sigma^2$.

Coming back to the equation we have then:

$$E\left[(f(x)^2 + \epsilon^2 + \hat{f}(x)^2 + 2\epsilon f(x) - 2f(x)\hat{f}(x) - 2\epsilon\hat{f}(x)\right] =$$

$$\sigma^2 + E\left[(f(x)^2 + \hat{f}(x)^2 + 2\epsilon f(x) - 2f(x)\hat{f}(x) - 2\epsilon\hat{f}(x)\right] =$$

$$\sigma^2 + E\left[2\epsilon f(x) - 2\epsilon\hat{f}(x)\right] + E\left[-2f(x)\hat{f}(x) + f(x)^2 + \hat{f}(x)^2\right].$$

The term $E\left[2\epsilon f(x) - 2\epsilon\hat{f}(x)\right]$ is zero because of $E[\epsilon] = 0$, so we are just left with $E[(y - \tilde{y})^2] = \sigma^2 + E\left[-2f(x)\hat{f}(x) + f(x)^2 + \hat{f}(x)^2\right] = \sigma^2 + E\left[\left(\hat{f}(x) - f(x)\right)^2\right]$

we can move forward observing that:

- $E\left[\left(\hat{f}(x) - f(x)\right)^2\right] = E\left[\hat{f}(x)^2\right] + E[f(x)^2] - 2E\left[f(x)\hat{f}(x)\right]$;

- $E[f(x)] = f(x)$, being $f(x) = X\beta$ not random, and

- $E\left[\hat{f}(x)^2\right] = V\left[\hat{f}(x)\right] + E\left[\hat{f}(x)\right]^2$

thus:

$$E\left[\left(\hat{f}(x) - f(x)\right)^2\right] = V\left[\hat{f}(x)\right] + E\left[\hat{f}(x)\right]^2 + f(x)^2 - 2f(x)E\left[\hat{f}(x)\right]$$
$$= V\left[\hat{f}(x)\right] + \left\{E\left[\hat{f}(x)\right] - f(x)\right\}^2$$

Summing up we are left with $E[(y - \tilde{y})^2] = \sigma^2 + V\left[\hat{f}(x)\right] + \left\{E\left[\hat{f}(x)\right] - f(x)\right\}^2$ which is

$$E[(y - \tilde{y})^2] = \sigma^2 + V\left[\hat{f}(x)\right] + Bias\left[\hat{f}(x)\right]^2$$

as requested.

The term σ^2 is, as shown before, the variance of the noise, which means that this is an irreducible term due to everything that has not been included in the model (for example interactions with other variables not included in the model or simply bad data collection).

The term $V\left[\hat{f}(x)\right]$ expresses the variance of the predictions, that is, it tells us how far the predictions on the test data are from their mean value. (It is more related to the accuracy aspect of the model). Having different models, the variance explains how much, on average, the predictions on the same test data differ from model to model

The term $\left\{E\left[\hat{f}(x)\right] - f(x)\right\}^2 = Bias\left[\hat{f}(x)\right]^2$ tells us how the model captures the true pattern in the dataset on which it has been trained. (It is more related to the explanatory aspect of the model). Having different models, the bias explains how much, on average, the models "stick" to the real relation that exists between the data.

When the model "cares too little about the training set", we might talk about underfitting. It usually comes from a simple model, which is unlikely to fit real life situations, and leads to predictions on the training data that are not really close to the observed data. That said, with

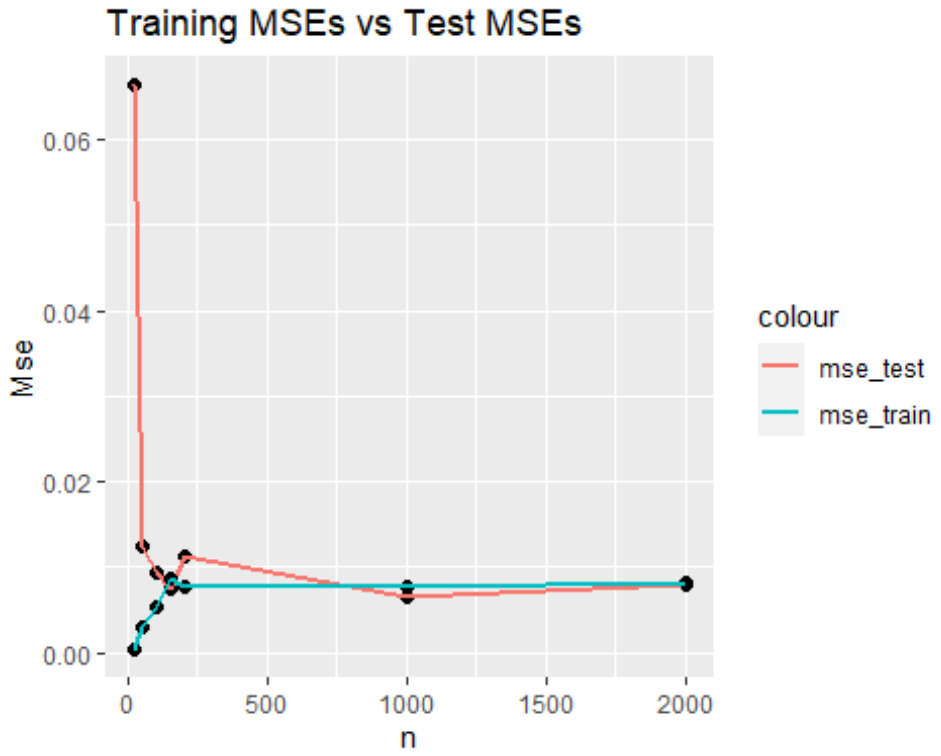
underfitting comes a high bias in the model. When underfitting occurs, a change in the training data doesn't affect too much the predictions on the test data, because the model will not change much. Therefore, with underfitting we *usually* have a low variance as well, even though this doesn't come necessarily.

On the other side, when the model depends too much on the training data, it will give almost perfect predictions for this set, resulting in a low bias. When this happens, the model will depend too much on the noise contained in the training error and will not be able to give accurate predictions for the test data (which are independent from the training data). Additionally, since the model will excessively depend on the training data, it will also have a high variance: changing the training data will give very different results.

I have already explained how the training and test MSEs behave in the case of the Franke function (Graph above) as function of the model complexity (order of the polynomial), but it might be interesting to check how they change as the number of data points increases or decreases. To do so, I can consider the 3rd order polynomial model, which is the best among those five, and check how the train and test MSEs change as the number of observations change.

```
datapoints <- c(20, 50, 100, 150, 200, 1000, 2000)
comparison_matrix <- as.data.frame(matrix(NA, 7, 3))
comparison_matrix[,1] <- datapoints
colnames(comparison_matrix) <- c("n", "mse_test", "mse_train")

for (i in 1:7){
  n <- datapoints[i]
  x1_bis <- cbind(runif(n, 0, 1))
  x2_bis <- cbind(runif(n, 0, 1))
  X_bis <- as.data.frame(cbind(x1_bis,x2_bis))

  term1_bis <- 0.75 * exp(-(9*x1_bis-2)^2/4 - (9*x2_bis-2)^2/4)
  term2_bis <- 0.75 * exp(-(9*x1_bis+1)^2/49 - (9*x2_bis+1)/10)
  term3_bis <- 0.5 * exp(-(9*x1_bis-7)^2/4 - (9*x2_bis-3)^2/4)
  term4_bis <- -0.2 * exp(-(9*x1_bis-4)^2 - (9*x2_bis-7)^2)

  y_bis <- cbind(term1_bis + term2_bis + term3_bis + term4_bis + 0.001
*rnorm(n))

  splitted_bis <- train_test(X_bis,y_bis)

  trainingX_bis <- splitted_bis[[1]]
  trainingY_bis <- splitted_bis[[2]]
  testX_bis <- splitted_bis[[3]]
  testY_bis <- splitted_bis[[4]]

  matrix_deg3_bis <- model_matrix(trainingX_bis[,1],trainingX_bis[,2],
3)
```

```
  beta_deg3_bis <- betas(matrix_deg3_bis, trainingY_bis)
  pred_deg3_bis <- predictions(testX_bis[,1],testX_bis[,2],3,beta_deg3
_bis)
  mse_deg3_bis <- mean((testY_bis - pred_deg3_bis)^2)

  pred_deg_train3_bis <- predictions(trainingX_bis[,1],trainingX_bis[,
2],3,beta_deg3_bis)
  mse_deg_train3_bis <- mean((trainingY_bis - pred_deg_train3_bis)^2)

  comparison_matrix[i,] <- cbind(n, mse_deg3_bis, mse_deg_train3_bis)
}

ggplot()+
  geom_point(data=comparison_matrix, aes(x=n, y=mse_test), size=2)+
  geom_line(data=comparison_matrix, aes(x=n, y=mse_test, colour="mse_t
est"), size=1)+
  geom_point(data=comparison_matrix, aes(x=n, y=mse_train), size=2)+
  geom_line(data=comparison_matrix, aes(x=n, y=mse_train, colour="mse_
train"), size=1)+
  ylab("Mse")+ggtitle("Training MSEs vs Test MSEs")
```



We can see that as the sample size increases, the MSE train increases as well, since the model doesn't overfit and, on the other hand, it gives "more importance" to the test data, compared to what it does with small sample sizes. From this point of view, we can also notice that the test MSE tends to decrease as n increases, for the same reason as before (less overfitting).

I added the cases n=1000 and n=2000 to show that since these MSEs are estimators of the real MSE (with bias of order n^{-1}), as *n* increases (and hypothetically goes to ∞), their value get closer to the real MSE. As n goes to infinite, the train and test MSE converge to the same value.

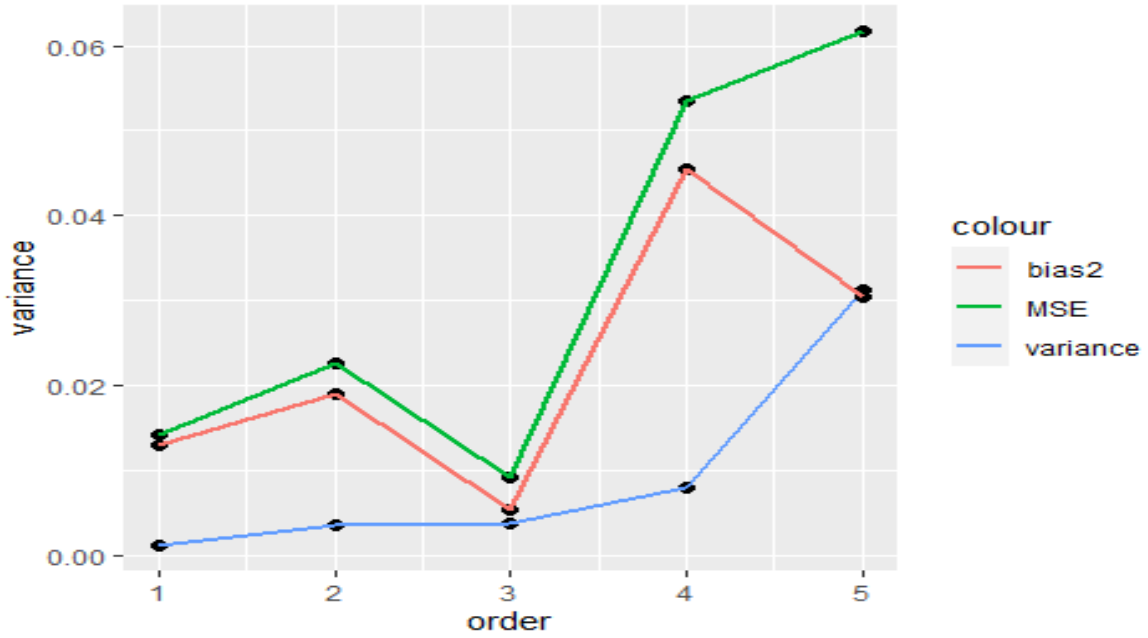
Now I use the bootstrap resampling method to evaluate not only how the MSEs of the predictor $\hat{f}(x)$ change as the order of the polynomial increases, but also to understand the impact of its two components, the variance, and the bias. This method works by creating k samples with replacement from the one we have, where these new samples have the same length of the one they are drawn from. Here I create 500 bootstrap samples from the training set I used before, and for each order of the polynomial I thus obtain 500 predictions for the 20-test data. From these predictions, I estimate the Bias and the variance of the predictor in addition to the MSE. The last column of the results matrix proves the MSE decomposition, that is, $MSE = VAR + BIAS^2$ (to which we should also add the variance of the noise).

```
set.seed(29012001)

pre = function(data, index, datitest) return(predict(lm(trainingY ~.-1
,data = as.data.frame(data),

subset = index), newdata=datitest))

numofboot <- 500
results_matrix <- matrix(NA, 5,5)
colnames(results_matrix) <- c("order","MSE","bias2","variance","proof"
)
for (degree in 1:5){

  mat <- get(paste("matrix_deg",toString(degree),sep=""))
  test <- get(paste("testX_", toString(degree), sep=""))
  prediction_matrix <- matrix(0, nrow = nrow(testX), ncol = numofboot)
# 20x500 matrix
  prediction_means <- matrix(0,1,nrow = nrow(testX)) # 20x1 vector
  rownames(prediction_matrix) <- rownames(testX)

  for (k in 1:numofboot){
    p <- pre(mat, sample(as.numeric(rownames(trainingX)), 80, replace
= T),test)

    prediction_matrix[,k] = cbind(p) #fill the matrix with predictions
for each bootstrap
  }

  for (i in 1:nrow(prediction_matrix)){
```

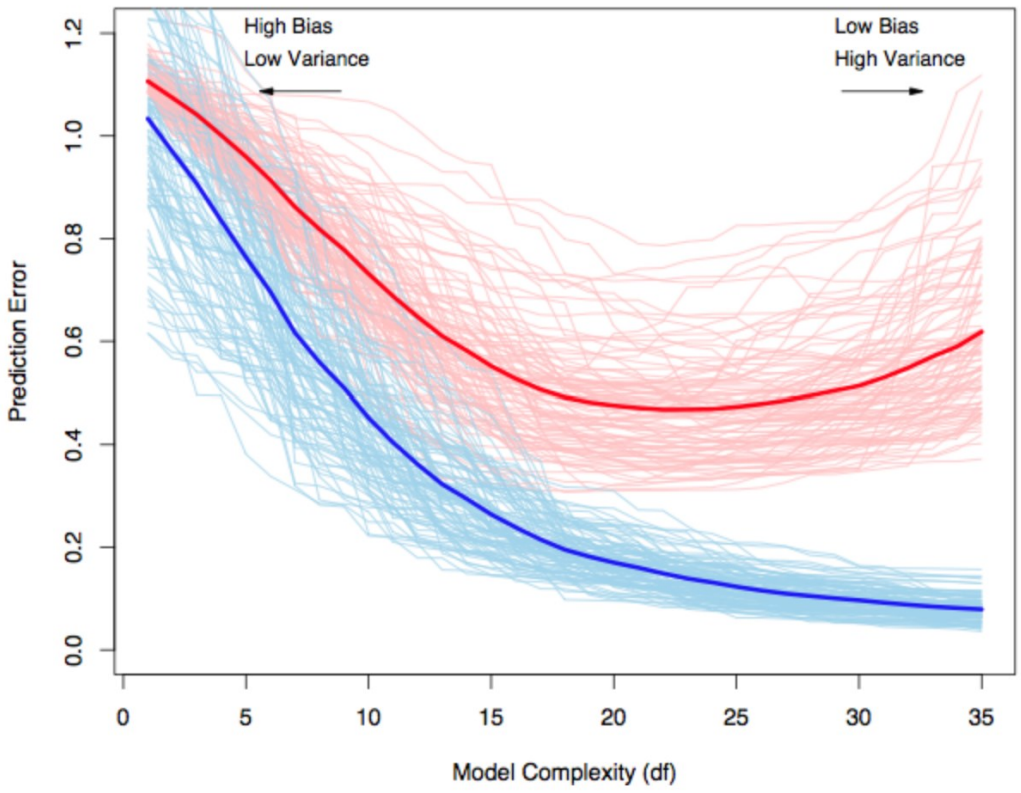
```
   prediction_means[i,] <- mean(prediction_matrix[i,]) #filling the ve
ctor with the mean prediction for each test data
  }
  bias_squared <- mean((testY - prediction_means)^2)

  q <- matrix(0,1,nrow = nrow(testX))
  for (k in 1:ncol(prediction_matrix)){
  q <- q + (prediction_matrix[,k]-prediction_means)^2}
  variance <- 1/(numofboot*20)*sum(q); variance

  m <- matrix(0,1,nrow = nrow(testX))
  for (k in 1:ncol(prediction_matrix)){
  m <- m + (testY - prediction_matrix[,k])^2}
  mean_squared_error <- 1/(numofboot*20) * sum(m); mean_squared_error

  proof <- mean_squared_error- variance -bias_squared
  results_matrix[degree,] <- c(degree,mean_squared_error, bias_squared
, variance, round(proof,8))
}
results_matrix
```

```
##      order         MSE       bias2    variance proof
## [1,]     1 0.014303851 0.013071983 0.001231868     0
## [2,]     2 0.022710291 0.019053959 0.003656332     0
## [3,]     3 0.009277764 0.005424529 0.003853235     0
## [4,]     4 0.053464024 0.045471986 0.007992038     0
## [5,]     5 0.061729210 0.030402749 0.031326462     0
```

```
require(ggplot2)
ggplot()+
  geom_point(data=as.data.frame(results_matrix), aes(x=order, y=varian
ce), size=2)+
  geom_line(data=as.data.frame(results_matrix), aes(x=order, y=varianc
e, colour="variance"), size=1)+
  geom_point(data=as.data.frame(results_matrix), aes(x=order, y=bias2)
, size=2)+
  geom_line(data=as.data.frame(results_matrix), aes(x=order, y=bias2,
colour="bias2"), size=1)+
  geom_point(data=as.data.frame(results_matrix), aes(x=order, y=MSE),
size=2)+
  geom_line(data=as.data.frame(results_matrix), aes(x=order, y=MSE, co
lour="MSE"), size=1)
```



The graph above gives an insight of the bias-variance tradeoff. It is possible to see that the lowest MSE is once again reached by the third order polynomial. Apart from the increase on the second order, the MSE is decreasing before the third order and increasing after it (Usually this is represented as a U shape, but it's behavior depends on the nature and quality of the data). For the variance, we can see that it increases monotonically: as the model gets more complex, the 500 predictions on each test data get more different and, thus, further from their mean value. On the other side, the bias has a different behavior. Usually, the bias should be monotonically decreasing as the complexity of the model increases. However, it is crucial to notice that the polynomials I am using to fit the data have predictors that are highly correlated (as shown when I analyzed the confidence intervals for the $\beta_j's$), and it has been proved that when we have highly correlated variables in a model, the bias can increase. This is what happens for the fourth order, and in fifth as well if compared to the first, second and third orders.



This image I found explains the tradeoff in a clear way: the thin lines correspond to the training errors (light blue) and test errors (light red) while the thick ones are their means. We can see that for the lowest orders (model complexities) of the polynomials, the test predictions that we obtain are “close to each other” when we change the sample (low variance) but the train predictions are different from the real values (high bias). On the other hand, higher orders present test predictions that differ a lot from each other as the sample changes but the train predictions are closer to their mean value. We can also see that the train error decrease monotonically, while it does not happen for the test error, that present the U shape I was talking about before.

PROBLEM 3

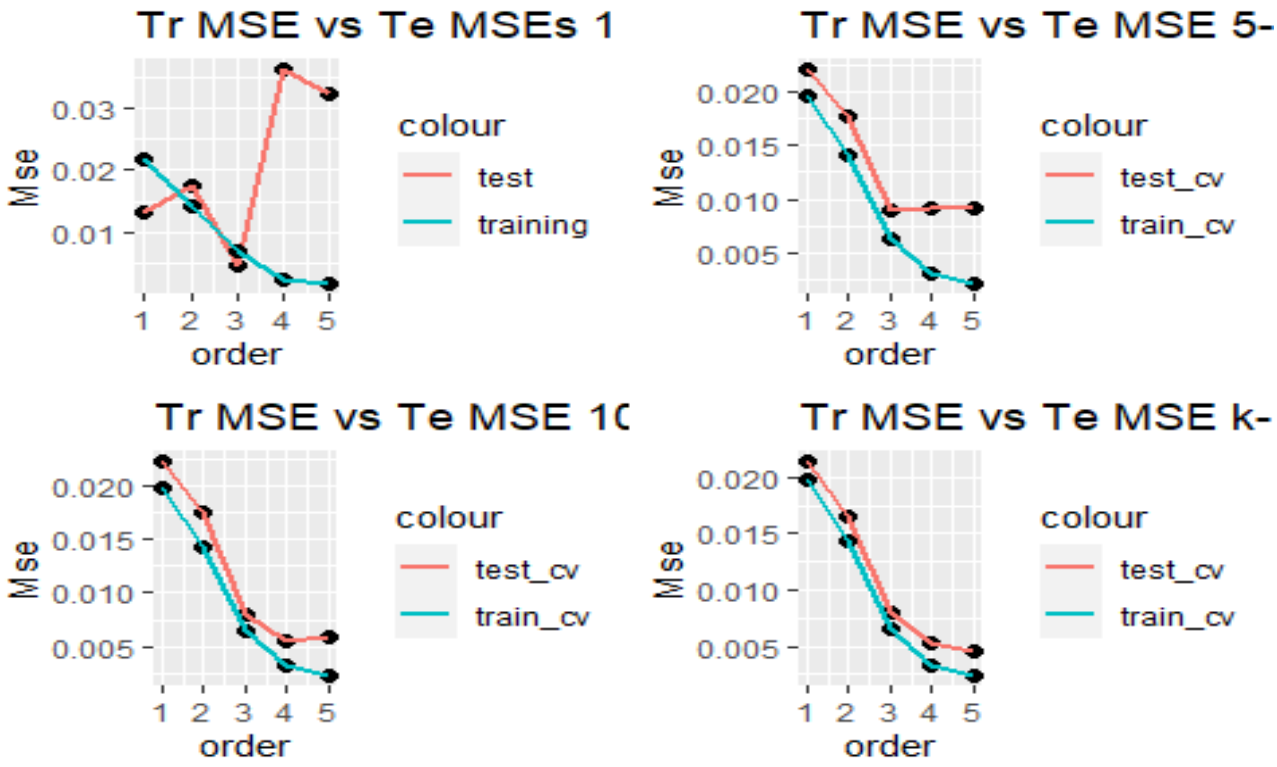
When we have a small dataset (like the one I created, which has just 100 “observations”), the values we obtain for the statistics we are interested in may be misleading, since they would be obtained using one single sample (of a small size). Therefore, we can use cross validation to “fake” more than one training and test sets from the dataset we are given. For each of these test sets we can hence obtain the value for the statistic of interest and then average those values, obtaining a more reliable quantity. Talking about the test MSE, since it depends on the data I use to test the model, it is completely normal that different sets of observations return different values for this statistic: I will need to average the results obtained on mor samples to

smooth the results. A clever way to move is to use cross validation. In addition to 5- and 10-folds cross validation, I decided to use leave-one-out-cross-validation as well (LOOCV), to have a better understanding of how the MSE changes with the number of folds.

Below there's my code for k-fold cross validation split. I will implement it as a 5-fold at first, then as a 10-fold and a n-fold as well, in order to see how much the results change in terms of MSE (train and test). I decided to plot and compare the "one-sample" case as well, to observe clearly how much cross validation helps:

```
require(gridExtra)

numfolds <- c(5,10,nrow(dataset))
matrixlist <- list()
msematrix_cv <- as.data.frame(matrix(NA, nrow = 5,3))
colnames(msematrix_cv) <- c("order", "test_cv", "train_cv")


for (k in 1:length(numfolds)){
  folds<-cut(seq(1,nrow(dataset)),breaks=numfolds[k],labels=FALSE)

  for (degree in 1:5){

    msestest <- c()
    msestrain <- c()

    for(i in 1:numfolds[k]){

      testIndexes_cv <- which(folds==i,arr.ind=TRUE)
      testData_cv <- dataset[testIndexes_cv, ]
      trainData_cv <- dataset[-testIndexes_cv, ]
      dataset_cv<-model_matrix(trainData_cv[,1],trainData_cv[,2],degre
e)
      beta_cv <- betas(dataset_cv, trainData_cv[,3])
      predmse_cv <- predictions(testData_cv[,1],testData_cv[,2],degree
,beta_cv)
      msetest_cv <- mean((testData_cv[,3] - predmse_cv)^2)
      predtrain_cv<-predictions(trainData_cv[,1],trainData_cv[,2],degr
ee,beta_cv)
      msetrain_cv <- mean((trainData_cv[,3] - predtrain_cv)^2)
      msestest[i] <- msetest_cv
      msestrain[i] <- msetrain_cv
    }

    msematrix_cv[degree,] <- c(degree, mean(msestest),mean(msestrain))
  }

  matrixlist[[k]] <- msematrix_cv
```

```
}
fivefolds<- matrixlist[[1]]
tenfolds <- matrixlist[[2]]
nfolds <- matrixlist[[3]]

p1 <- ggplot()+  geom_point(data=msematrix, aes(x=order, y=test), size
=2)+  geom_line(data=msematrix, aes(x=order, y=test, colour="test"), s
ize=1)+  geom_point(data=msematrix, aes(x=order, y=train), size=2)+  g
eom_line(data=msematrix, aes(x=order, y=train, colour="training"), siz
e=1)+  ylab("Mse")+ggtitle("Tr MSE vs Te MSEs 1 sample")

p2 <- ggplot()+  geom_point(data=fivefolds, aes(x=order, y=test_cv), s
ize=2)+  geom_line(data=fivefolds, aes(x=order, y=test_cv, colour="tes
t_cv"), size=1)+  geom_point(data=fivefolds, aes(x=order, y=train_cv),
size=2)+  geom_line(data=fivefolds, aes(x=order, y=train_cv, colour="t
rain_cv"), size=1)+  ylab("Mse")+ggtitle("Tr MSE vs Te MSE 5-FOLDS CV"
)

p3 <- ggplot()+  geom_point(data=tenfolds, aes(x=order, y=test_cv), si
ze=2)+  geom_line(data=tenfolds, aes(x=order, y=test_cv, colour="test_
cv"), size=1)+  geom_point(data=tenfolds, aes(x=order, y=train_cv), si
ze=2)+  geom_line(data=tenfolds, aes(x=order, y=train_cv, colour="trai
n_cv"), size=1)+  ylab("Mse")+ggtitle("Tr MSE vs Te MSE 10-FOLDS CV")

p4 <- ggplot()+  geom_point(data=nfolds, aes(x=order, y=test_cv), size
=2)+  geom_line(data=nfolds, aes(x=order, y=test_cv, colour="test_cv")
, size=1)+  geom_point(data=nfolds, aes(x=order, y=train_cv), size=2)+
geom_line(data=nfolds, aes(x=order, y=train_cv, colour="train_cv"), si
ze=1)+  ylab("Mse")+ggtitle("Tr MSE vs Te MSE k-FOLDS CV")

grid.arrange(p1, p2, p3, p4, ncol = 2, nrow=2)
```



As we can see, the first plot has values that fluctuate a lot, in comparison to the other graphs, and that is because those MSEs are affected a lot by the training and test sets I used.

In the other three graphs, it stands out that the results are “smoother” (because of having averaged out). We can also notice that the train MSEs are obtained on more values (80) than the test MSEs. Therefore, each fold tends, for each order of the polynomial, to give a train value which is closer to the mean MSE. Consequently, the cross-validation train MSEs are relatively the same from the 5-fold case to the 10 and LOOCV case. (With the LOOCV we also get a minimum Bias, since the training sets differ only for one observation, and a higher variance in the meanwhile, since the model “over-depends” on the train sets. That’s another proof of the trade-off).

On the other hand, what happens for the test MSE is that it is obtained on fewer values (20 instead of 80). In this case, it really helps to use many folds, because each of them might give a very different value. Therefore, the more samples (folds) I use, the more accurate the test MSE will be (we will have to check for the complexity of the model, as I will explain after).

In other words, looking at the graphs, we can see that the test MSEs for the fourth and fifth order get lower as I increase the number of folds: I avoid overfitting by varying many times the set I learn the model on, that is, the model generalizes better. As a result, while increasing the number of folds, the test MSEs does not increases as much as it does for the “1-sample” case. Nonetheless, if I had increased the complexity of model up to order 10 or more, I would have probably seen a bigger increase in the test MSE for the 10 folds cv as well and I would have needed to check more situations between the 10 and k folds.

I can now do a comparison between the values of the test MSEs that I got from the different methods:

```
comparisonMSE <- matrix(NA, 5, 4)
dimnames(comparisonMSE) <- list(c(1:5), c("5-folds", "10-folds", "n-fo
lds", "bootstrap"))
comparisonMSE[,1] <- fivefolds[,2]
comparisonMSE[,2] <- tenfolds[,2]
comparisonMSE[,3] <- nfolds[,2]
comparisonMSE[,4] <- results_matrix[,2]
comparisonMSE
```

```
##       5-folds    10-folds     n-folds   bootstrap
## 1 0.022033754 0.022177680 0.021347734 0.014303851
## 2 0.017791296 0.017557285 0.016490997 0.022710291
## 3 0.008898559 0.008082858 0.007995564 0.009277764
## 4 0.009249017 0.005514891 0.005211207 0.053464024
## 5 0.009225601 0.005826593 0.004562921 0.061729210
```

As we can see, apart from the first order polynomial, for the other four degrees the test MSEs we obtain with the bootstrap resampling method are higher than the cross-validation results that I just obtained. It has been proved that (on average) in the bootstrap method, inside each of the samples obtained from the original one there are approximately only 63.2% observations that are drawn only once. Because of this, the test MSEs obtained as before come from models that tend to a mild overfit, and it is followed by an increase on the test MSE, as reported in the matrix. We can also see that as either the used method changes or the number of folds changes, a different model is identified as the "best one" in terms of test MSE. In particular, the 5-fold and the bootstrap would lead to choose the third one, while the 10 folds would lead to the fourth order and the n folds to the fifth one.

In general, the bias decreases as the number k of fold increases in cross validation. Once again though, we want a model which has both a low variance and a low bias. The choose of k, and then of the model to keep, closely depends on the data I have, the size of the sample, etc..., so there is no real "general rule", even though the 5- and 10-folds cases are the most implemented.

PROBLEM 4

When we want to select a model to fit the data, we want one that is able both to explain them well and, hopefully at the same time, to predict values for new and unseen data.

Generally, focusing now about predictions, we can encounter three types of errors, as said before, expressed in the bias, the variance and the unexplainable variance (or irreducible error, about which, as the name says, we can't do anything about).

When we work with Ordinary Least Squares, the parameters' estimates will be unbiased and will have minimal variance among all unbiased linear estimators (BLUE estimators). Coming from this, the OLS model will give the best linear *unbiased* predictions. Even though it may look like the best model we can think of, having the minimal variance among the BLUEs doesn't really mean that this variance is going to be low. Indeed, it can be very large (we will also have to consider the complexity of the model). From these observations, it follows that we could decide to "sacrifice" the unbiasedness to save a lot (we hope) of variance.

This is, once again, the bias-variance tradeoff.

Following from these observations, we can introduce Ridge regression, which is an extension of Linear regression. Ridge works by adding a penalty factor λ when we estimate the coefficients of the model. This factor penalizes high values of these coefficients by shrinking them.

For this reason, it is obvious how important it is to scale the data beforehand: if we do not scale them, the coefficients might be high just for a matter of magnitude, and it would lead to wrong or misleading results. Follows, if the features have different scales, they contribute differently to the penalized term (which is a sum of squares of the $\beta_j's$).

```
scaling <- function(vector){
  mysd <- function(u) sqrt(sum((u - mean(u))^2)/length(u))
  vnew <- scale(vector, center = TRUE, scale = apply(vector,2,mysd))
  return(vnew)
}

scaled_trainingX <- cbind(scaling(trainingX))
rownames(scaled_trainingX) <- translate(rownames(trainingX))
scaled_trainingY <- scaling(as.matrix(trainingY))
rownames(scaled_trainingY) <- translate(rownames(trainingX))
scaled_testX <- cbind(scaling(testX))
rownames(scaled_testX) <- translate(rownames(testX))
scaled_testY<- scaling(as.matrix(testY))
rownames(scaled_testY) <- translate(rownames(testX))

matrix_deg_sc1 <- model_matrix(scaled_trainingX[,1],scaled_trainingX[,
2],1)
matrix_deg_sc2 <- model_matrix(scaled_trainingX[,1],scaled_trainingX[,
2],2)
matrix_deg_sc3 <- model_matrix(scaled_trainingX[,1],scaled_trainingX[,
2],3)
matrix_deg_sc4 <- model_matrix(scaled_trainingX[,1],scaled_trainingX[,
2],4)
matrix_deg_sc5 <- model_matrix(scaled_trainingX[,1],scaled_trainingX[,
2],5)

testX_sc_1 <- as.data.frame(model_matrix(scaled_testX[,1],scaled_testX
[,2],1))
```

```
testX_sc_2 <- as.data.frame(model_matrix(scaled_testX[,1],scaled_testX
[,2],2))
testX_sc_3 <- as.data.frame(model_matrix(scaled_testX[,1],scaled_testX
[,2],3))
testX_sc_4 <- as.data.frame(model_matrix(scaled_testX[,1],scaled_testX
[,2],4))
testX_sc_5 <- as.data.frame(model_matrix(scaled_testX[,1],scaled_testX
[,2],5))
```

What we do with Ridge regression, thus, is obtaining an estimator of β which is not unbiased, but that give a variance which is smaller than the one given by the OLS. In particular, it has been proved that there always exists a value of λ such that the ridge estimator has lower MSE than the OLS estimator (Theobald, C. M. (1974)): to choose the best value of λ among a grid of values we can use cross-validation (as shown below) and look for the one that gives the lowest test MSE. Even though Ridge regression enforces the coefficients to converge to 0, their value will never be exactly zero: coming from this, Ridge can be a useful tool to reduce the impact of the corresponding variables but will not perform a proper variable selection.

The following function gives the $\hat{\beta}^{ridge}$ estimate of the coefficients:

```
betaridge <- function(matX,vec_y,lambda){
  ridge <- solve(t(matX) %*% matX + lambda * diag(1,ncol(matX), ncol(m
atX))) %*% t(matX) %*% vec_y
  return(ridge)
}
```

I here set a sequence of 10 random lambdas, but later on, I will use built-in R functions and will let those functions decide from which lambdas they will have to find the best one. For simplicity, the sequence I am creating contains only 10 values.

The lambdas here are: 100, 35.938, 12.915, 4.642, 1.668, 0.599, 0.215, 0.077, 0.028 and 0.01

```
numlambda <- 10
mseridge <- list()
lambdas <- 10^(seq(2,-2,length=numlambda))
pred_rid = function(matrice, index, datitest, trainy, lambda){
    index <- translate(index)
    ridge_b<-betaridge(matrice[index,], trainy[index,], lambda)
    return(predictions(datitest[,1],datitest[,2],degree,ridge_b))}

ty <- cbind(trainingY)
rownames(ty) <- translate(rownames(trainingX))

for (l in 1:numlambda){
  lmb <- lambdas[l]
  numofboot_r <- 1000
```

```
  results_matrix_r <- matrix(NA, 5,6)
  colnames(results_matrix_r) <- c("order","MSE","bias2","variance","pr
oof", "used_lambda")
  for (degree in 1:5){
    mat_r <- get(paste("matrix_deg",toString(degree),sep=""))
    test_r <- get(paste("testX_", toString(degree), sep=""))
    prediction_matrix_r <- matrix(0, nrow = nrow(testX), ncol = numofb
oot_r)
    prediction_means_r <- matrix(0,1,nrow = nrow(testX))
    rownames(prediction_matrix_r) <- rownames(testX)
    for (f in 1:numofboot_r){
      p_r <- pred_rid(mat_r,sample(as.numeric(rownames(trainingX)), 80
, replace = T),test_r, ty, lmb)
      prediction_matrix_r[,f] = cbind(p_r)
    }

    for (i in 1:nrow(prediction_matrix_r)){

    prediction_means_r[i,] <- mean(prediction_matrix_r[i,])}
    bias_squared_r <- mean((testY - prediction_means_r)^2)

    q_r <- matrix(0,1,nrow = nrow(testX))
    for (z in 1:ncol(prediction_matrix_r)){
    q_r <- q_r + (prediction_matrix_r[,z]-prediction_means_r)^2}
    variance_r <- 1/(numofboot_r*20)*sum(q_r); variance_r

    m_r <- matrix(0,1,nrow = nrow(testX))
    for (k in 1:ncol(prediction_matrix_r)){
    m_r <- m_r + (testY - prediction_matrix_r[,k])^2}
    mean_squared_error_r <- 1/(numofboot_r*20) * sum(m_r); mean_square
d_error_r

    proof_r <- mean_squared_error_r- variance_r -bias_squared_r
    results_matrix_r[degree,] <- c(degree,mean_squared_error_r, bias_s
quared_r, variance_r, round(proof_r,8), lmb)
  }
  mseridge[[l]] <- results_matrix_r
}
```

```
mseridge[[10]] #to see what the results look like and prove the MSE
               decomposition
```

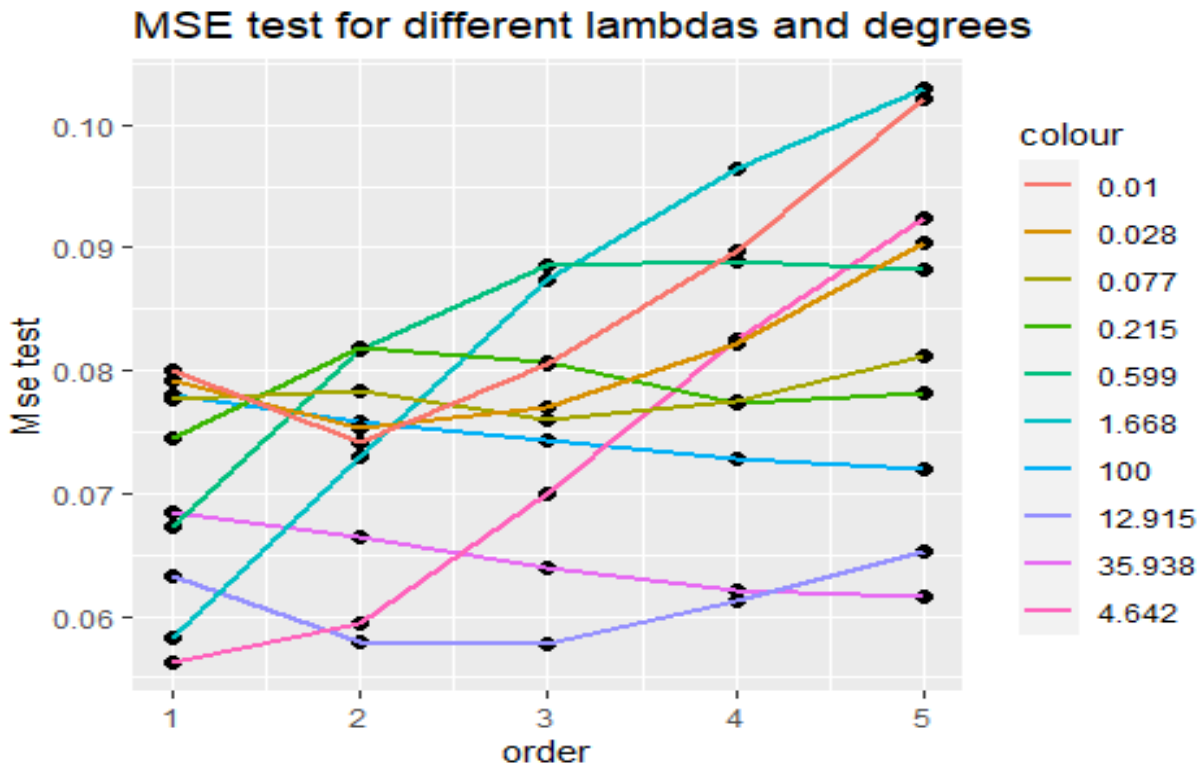
```
##      order        MSE      bias2    variance proof used_lambda
## [1,]     1 0.07997775 0.07867572 0.001302032     0        0.01
## [2,]     2 0.07417987 0.07220316 0.001976705     0        0.01
## [3,]     3 0.08049386 0.07882519 0.001668673     0        0.01
## [4,]     4 0.08977509 0.08717587 0.002599217     0        0.01
## [5,]     5 0.10219589 0.09850721 0.003688683     0        0.01
```

```
lambdamses <- as.data.frame(matrix(NA, nrow = 5, ncol = numlambda+1))
lambdamses[,1] <- c(1:5)
colnames(lambdamses) <- c( "order",round(lambdas, 3))
rownames(lambdamses) <- c(1:5)
for (M in 1:length(mseridge)){
  ma <- mseridge[[M]]
  mse_ <- ma[,2]
  lambdamses[,M+1] <- mse_
}
lambdamses
```

```
##   order        100     35.938     12.915      4.642      1.668
0.599
## 1     1 0.07803507 0.06854054 0.06329774 0.05623761 0.05826443 0.06
735715
## 2     2 0.07587550 0.06641816 0.05799679 0.05945719 0.07302217 0.08
166708
## 3     3 0.07425611 0.06402808 0.05782888 0.07001069 0.08733912 0.08
852630
## 4     4 0.07290640 0.06215384 0.06132200 0.08258601 0.09653158 0.08
891330
## 5     5 0.07198868 0.06164135 0.06524602 0.09236162 0.10297199 0.08
818878
##        0.215      0.077      0.028       0.01
## 1 0.07444452 0.07774127 0.07923572 0.07997775
## 2 0.08182024 0.07833004 0.07537752 0.07417987
## 3 0.08069557 0.07607131 0.07699911 0.08049386
## 4 0.07736315 0.07752509 0.08220987 0.08977509
## 5 0.07815026 0.08112037 0.09041661 0.10219589
```

I can now plot the different MSEs given by the 10 lambdas as the order of the polynomial increases:

```
require(ggplot2)
ggplot()+  geom_point(data=lambdamses, aes(x=order, y=lambdamses[,2]), size=2)+  geom_line(data=lambdamses, aes(x=order, y=lambdamses[,2], colour="100"), size=1)+  geom_point(data=lambdamses, aes(x=order, y=lambdamses[,3]), size=2)+  geom_line(data=lambdamses, aes(x=order, y=lambdamses[,3], colour="35.938"), size=1)+  geom_point(data=lambdamses, aes(x=order, y=lambdamses[,4]), size=2)+  geom_line(data=lambdamses, aes(x=order, y=lambdamses[,4], colour="12.915"), size=1)+  geom_point(data=lambdamses, aes(x=order, y=lambdamses[,5]), size=2)+  geom_line(data=lambdamses, aes(x=order, y=lambdamses[,5], colour="4.642"), size=1)+
geom_point(data=lambdamses, aes(x=order, y=lambdamses[,6]), size=2)+
geom_line(data=lambdamses, aes(x=order, y=lambdamses[,6], colour="1.668"), size=1)+  geom_point(data=lambdamses, aes(x=order, y=lambdamses[,7]), size=2)+  geom_line(data=lambdamses, aes(x=order, y=lambdamses[,7], colour="0.599"), size=1)+  geom_point(data=lambdamses, aes(x=order, y=lambdamses[,8]), size=2)+  geom_line(data=lambdamses, aes(x=order, y=lambdamses[,8], colour="0.215"), size=1)+  geom_point(data=lambdamses, aes(x=order, y=lambdamses[,9]), size=2)+  geom_line(data=lambdamses, aes(x=order, y=lambdamses[,9], colour="0.077"), size=1)+  geom_point(data=lambdamses, aes(x=order, y=lambdamses[,10]), size=2)+  geom_line(data=lambdamses, aes(x=order, y=lambdamses[,10], colour="0.028"), size=1)+  geom_point(data=lambdamses, aes(x=order, y=lambdamses[,11]), size=2)+  geom_line(data=lambdamses, aes(x=order, y=lambdamses[,11], colour="0.01"), size=1)+  ylab("Mse test")+ggtitle("MSE test for different lambdas and degrees")
```



Looking at this graph, we can see how different lambdas give different values of the test MSEs, as the order of the polynomial changes. We can see that for the majority, the bootstrap method seems to choose the value of 12.915, which is pretty high. When we penalize a lot, we drive to zero the impact of "non informative" variables. We can still see that the order to which corresponds the lowest test MSE in the 3rd one. In general, the bootstrap method gives higher test MSE compared to cross validation, as explained before (0.632 rule), and therefore we usually use cross validation to find the best values of lambda in term of MSE.

Since the model will depend less on the train test, the overall results will change, and so will do the best lambda: less dependence of the train data will lead to more precise predictions even with a lower penalization, and that is what happen with cross validation. Therefore, I am now implementing cross-validation to select lambda as the complexity of the model increases:

```
numlambda <- 10
lambdas <- 10^(seq(-2,2,length=numlambda))

matrixlist_rid5 <- list()

for (l in 1:numlambda){
  lmb_rid5 <- lambdas[l]
  msematrix_cv_rid5 <- as.data.frame(matrix(NA, nrow = 5,4))
  colnames(msematrix_cv_rid5) <- c("used lambda","order", "test_cv", "
train_cv")
  folds_rid5<-cut(seq(1,nrow(dataset)),breaks=5,labels=FALSE)
```

```
    for (degree in 1:5){

      msestest_rid5 <- c()
      msestrain_rid5 <- c()

      for(i in 1:5){

        testIndexes_cv_rid5 <- which(folds_rid5==i,arr.ind=TRUE)
        testData_cv_rid5 <- dataset[testIndexes_cv_rid5, ]
        trainData_cv_rid5 <- dataset[-testIndexes_cv_rid5, ]
        dataset_cv_rid5<-model_matrix(trainData_cv_rid5[,1],trainData_
cv_rid5[,2],degree)
        beta_cv_rid5 <- betaridge(dataset_cv_rid5, trainData_cv_rid5[,
3], lmb_rid5)
        predmse_cv_rid5 <- predictions(testData_cv_rid5[,1],testData_c
v_rid5[,2],degree,beta_cv_rid5)
        msetest_cv_rid5 <- mean((testData_cv_rid5[,3] - predmse_cv_rid
5)^2)
        predtrain_cv_rid5<-predictions(trainData_cv_rid5[,1],trainData
_cv_rid5[,2],degree,beta_cv_rid5)
        msetrain_cv_rid5 <- mean((trainData_cv_rid5[,3] - predtrain_cv
_rid5)^2)
        msestest_rid5[i] <- msetest_cv_rid5
        msestrain_rid5[i] <- msetrain_cv_rid5
      }

      msematrix_cv_rid5[degree,] <- c(lmb_rid5, degree, mean(msestest_
rid5), mean(msestrain_rid5))
    }

    matrixlist_rid5[[l]] <- msematrix_cv_rid5
}

matrixlist_rid5[[1]] #to see what the results look like

##   used lambda order     test_cv    train_cv
## 1        0.01     1 0.022021380 0.019641142
## 2        0.01     2 0.017694421 0.014066393
## 3        0.01     3 0.010246792 0.008431758
## 4        0.01     4 0.008578762 0.007099525
## 5        0.01     5 0.008188618 0.006693915



numlambda <- 10
lambdas <- 10^(seq(-2,2,length=numlambda))
```

```
matrixlist_rid10 <- list()

for (l in 1:numlambda){
  lmb_rid10 <- lambdas[l]
  msematrix_cv_rid10 <- as.data.frame(matrix(NA, nrow = 5,4))
  colnames(msematrix_cv_rid10) <- c("used lambda","order", "test_cv",
"train_cv")
  folds_rid10<-cut(seq(1,nrow(dataset)),breaks=10,labels=FALSE)


    for (degree in 1:5){

      msestest_rid10 <- c()
      msestrain_rid10 <- c()

      for(i in 1:10){

        testIndexes_cv_rid10 <- which(folds_rid10==i,arr.ind=TRUE)
        testData_cv_rid10 <- dataset[testIndexes_cv_rid10, ]
        trainData_cv_rid10 <- dataset[-testIndexes_cv_rid10, ]
        dataset_cv_rid10<-model_matrix(trainData_cv_rid10[,1],trainDat
a_cv_rid10[,2],degree)
        beta_cv_rid10 <- betaridge(dataset_cv_rid10, trainData_cv_rid1
0[,3], lmb_rid10)
        predmse_cv_rid10 <- predictions(testData_cv_rid10[,1],testData
_cv_rid10[,2],degree,beta_cv_rid10)
        msetest_cv_rid10 <- mean((testData_cv_rid10[,3] - predmse_cv_r
id10)^2)
        predtrain_cv_rid10<-predictions(trainData_cv_rid10[,1],trainDa
ta_cv_rid10[,2],degree,beta_cv_rid10)
        msetrain_cv_rid10 <- mean((trainData_cv_rid10[,3] - predtrain_
cv_rid10)^2)
        msestest_rid10[i] <- msetest_cv_rid10
        msestrain_rid10[i] <- msetrain_cv_rid10
      }

      msematrix_cv_rid10[degree,] <- c(lmb_rid10, degree, mean(msestes
t_rid10), mean(msestrain_rid10))
    }

    matrixlist_rid10[[l]] <- msematrix_cv_rid10
}
```

Before moving on, I want to show the results we get if we test each model on the same test set. I therefore applied 10-folds cross-validation only on the training set I already had (80 observation): this set is split in 10 and the model is trained on 70 observation and validated on

10. After that, each model is tested on the same 20 observation. We can see that the results are slightly different:

```
#CROSS VALIDATION WITH THE SAME TEST SET

numlambda <- 10
lambdas <- 10^(seq(-2,2,length=numlambda))
Zmatrixlist_rid10 <- list()
ZtestData_cv_rid10 <- testX
trainingY <- cbind(trainingY)
rownames(trainingY) <- rownames(trainingX)

for (l in 1:numlambda){
  Zlmb_rid10 <- lambdas[l]
  Zmsematrix_cv_rid10 <- as.data.frame(matrix(NA, nrow = 5,4))
  colnames(Zmsematrix_cv_rid10) <- c("used lambda","order", "test_cv",
"train_cv")
  Zfolds_rid10<-cut(seq(1,nrow(trainingX)),breaks=10,labels=FALSE)

    for (degree in 1:5){

      Zmsestest_rid10 <- c()
      Zmsestrain_rid10 <- c()

      for(i in 1:10){

        ZtestIndexes_cv_rid10 <- which(Zfolds_rid10==i,arr.ind=TRUE)
        datatrainX_ridge10 <- trainingX[-ZtestIndexes_cv_rid10, ]
        datatrainY_ridge10 <- trainingY[-ZtestIndexes_cv_rid10, ]
        Zdataset_cv_rid10<-model_matrix(datatrainX_ridge10[,1],datatra
inX_ridge10[,2],degree)
        Zbeta_cv_rid10 <- betaridge(Zdataset_cv_rid10, datatrainY_ridg
e10, l)
        Zpredmse_cv_rid10 <- predictions(testX[,1],testX[,2],degree,Zb
eta_cv_rid10)
        Zmsetest_cv_rid10 <- mean((testY - Zpredmse_cv_rid10)^2)
        Zpredtrain_cv_rid10<-predictions(datatrainX_ridge10[,1],datatr
ainX_ridge10[,2],degree,Zbeta_cv_rid10)
        Zmsetrain_cv_rid10 <- mean((datatrainY_ridge10 - Zpredtrain_cv
_rid10)^2)
        Zmsestest_rid10[i] <- Zmsetest_cv_rid10
        Zmsestrain_rid10[i] <- Zmsetrain_cv_rid10
      }

      Zmsematrix_cv_rid10[degree,] <- c(Zlmb_rid10, degree, mean(Zmses
test_rid10), mean(Zmsestrain_rid10))
```

```
    }

    Zmatrixlist_rid10[[l]] <- Zmsematrix_cv_rid10
}
Zmatrixlist_rid10[[1]]

##   used lambda order    test_cv   train_cv
## 1        0.01     1 0.01506553 0.02591217
## 2        0.01     2 0.01372076 0.02100353
## 3        0.01     3 0.01595117 0.02027824
## 4        0.01     4 0.01683355 0.02022399
## 5        0.01     5 0.01713743 0.02027656

matrixlist_rid10[[1]]

##   used lambda order     test_cv    train_cv
## 1        0.01     1 0.022168654 0.019756232
## 2        0.01     2 0.017508394 0.014238447
## 3        0.01     3 0.010481214 0.008274352
## 4        0.01     4 0.008976498 0.007042623
## 5        0.01     5 0.008624236 0.006657168
```

Apart from the first order, we can see that usually cross-validation tends to be a bit optimistic, but luckily the results are not too different. Anyway, the difference comes from the fact that as the test fold changes, the test data are used as train data as well, and when we obtain the test MSE as a mean of the MSEs obtained for each fold, this overlap shows in a lower MSE. (The sets are not independent as they should be).

Moving on, I can now use the results obtained with the cross validation to check, for each order of the polynomial, which is the best lambda to use. I will also use a bult-in R function to show that the results are the same:

```
#5 FOLDS

s5 <- as.data.frame(matrix(NA, nrow = 5, ncol = numlambda+1))
s5[,1] <- c(1:5)
colnames(s5) <- c( "order",round(lambdas, 3))
rownames(s5) <- c(1:5)
for (M in 1:length(matrixlist_rid5)){
  mt <- matrixlist_rid5[[M]]
  msete5 <- mt[,3]
  s5[,M+1] <- msete5}

bestlam_cv5 <- matrix(NA,5,1)
dimnames(bestlam_cv5) <- list(c("order 1","order 2","order 3","order
4","order 5"),"lambdas")
```

```
for (row in 1:5){
  lb_cv5 <- which.min(s5[row,])
  bestlam_cv5[row,] <- colnames(s5)[lb_cv5]
}
```

```
bestlam_cv5
```

```
##         lambdas
## order 1 "0.077"
## order 2 "0.028"
## order 3 "0.01"
## order 4 "0.01"
## order 5 "0.01"
```

```
require(glmnet)
lambdas <- 10^(seq(-2,2,length=numlambda))
p5 <- matrix(NA,5,2)
dimnames(p5) <- list(1:5, c("order", "best lambda"))
testfeatures <- list(testX_sc_1, testX_sc_2,testX_sc_3,testX_sc_4,test
X_sc_5)
for (order in 1:5){
  mt <- get(paste("matrix_deg_sc",toString(order),sep=""))
  cv_ridge <- cv.glmnet(mt, scaled_trainingY, alpha = 0, lambda = lamb
das, nfolds = 5)
  optimal_lambda <- cv_ridge$lambda.min
  p5[order,] <- c(order, optimal_lambda)
}
p5
```

```
##   order best lambda
## 1     1  0.02782559
## 2     2  0.02782559
## 3     3  0.01000000
## 4     4  0.01000000
## 5     5  0.01000000
```

```
#10 FOLDS
```

```
s10 <- as.data.frame(matrix(NA, nrow = 5, ncol = numlambda+1))
s10[,1] <- c(1:5)
colnames(s10) <- c( "order",round(lambdas, 3))
rownames(s10) <- c(1:5)
for (M in 1:length(matrixlist_rid10)){
  mt <- matrixlist_rid10[[M]]
  msete10 <- mt[,3]
  s10[,M+1] <- msete10}

bestlam_cv10 <- matrix(NA,5,1)
```

```
dimnames(bestlam_cv10) <- list(c("order 1","order 2","order 3","order
4","order 5"),"lambdas")
for (row in 1:5){
  lb_cv10 <- which.min(s10[row,])
  bestlam_cv10[row,] <- colnames(s10)[lb_cv10]
}
bestlam_cv10
```

```
##         lambdas
## order 1 "0.077"
## order 2 "0.028"
## order 3 "0.01"
## order 4 "0.01"
## order 5 "0.01"
```

```
require(glmnet)
lambdas <- 10^(seq(-2,2,length=numlambda))
p10 <- matrix(NA,5,2)
dimnames(p10) <- list(1:5, c("order", "best lambda"))
for (order in 1:5){
  mt <- get(paste("matrix_deg_sc",toString(order),sep=""))
  cv_ridge <- cv.glmnet(mt, scaled_trainingY, alpha = 0, lambda = lamb
das, nfolds = 10, type.measure="mse")
  optimal_lambda <- cv_ridge$lambda.min
  p10[order,] <- c(order, optimal_lambda)

  #cv_ridge$cvm[cv_ridge$lambda==cv_ridge$lambda.min] to show which is
the minimum MSE
  #Note that with cv_ridge$lambda that cv.glmnet gives the lambdas in
decreasing order, so from cv_ridge$cvm the last value is the one we ha
ve to look at, and it corresponds to the best value (0.01 in 4 out of
5 cases).
 }
p10
```

```
##   order best lambda
## 1     1  0.01000000
## 2     2  0.02782559
## 3     3  0.01000000
## 4     4  0.01000000
## 5     5  0.01000000
```

We can see that, both in the 5- and 10- fold cross-validation, different values of lambda are chosen, if compared to the bootstrap method. In particular, almost in every case the value 0.01 is the one that gives the best test MSE. We can see here that, apart from the first order, my function and the one that R provides give the same results 😊.

Everything here obviously depends on the values of lambda I use for cross validation, so if I had used a different set of values, I would have obtained a different chosen penalty factor. If I don't choose a set of lambdas beforehand, the functions will give the following values:

```
require(glmnet)
pr <- matrix(NA,5,2)
dimnames(pr) <- list(1:5, c("order", "best lambda"))
for (order in 1:5){
  mt <- get(paste("matrix_deg",toString(order),sep=""))
  cv_ridger <- cv.glmnet(mt, trainingY, alpha = 0, nfolds = 10, type.m
easure="mse")
  par(mfrow=c(1,2))
  plot(cv_ridger)
  plot(cv_ridger$glmnet.fit, xvar="lambda", label=5)
  optimal_lambdar <- cv_ridger$lambda.min
  pr[order,] <- c(order, optimal_lambdar)}

pr
```

```
##   order best lambda
## 1     1  0.01689327
## 2     2  0.01790554
## 3     3  0.01794467
## 4     4  0.01794467
## 5     5  0.01794467
```

As we can observe, using the R function with a different sequence of lambda lead to choose values that are different from 0.01.

Mean-Squared Error
0.02
0.04
0.06
0.08
2 2 2 2 2
-4
0
2
4
Log(λ)

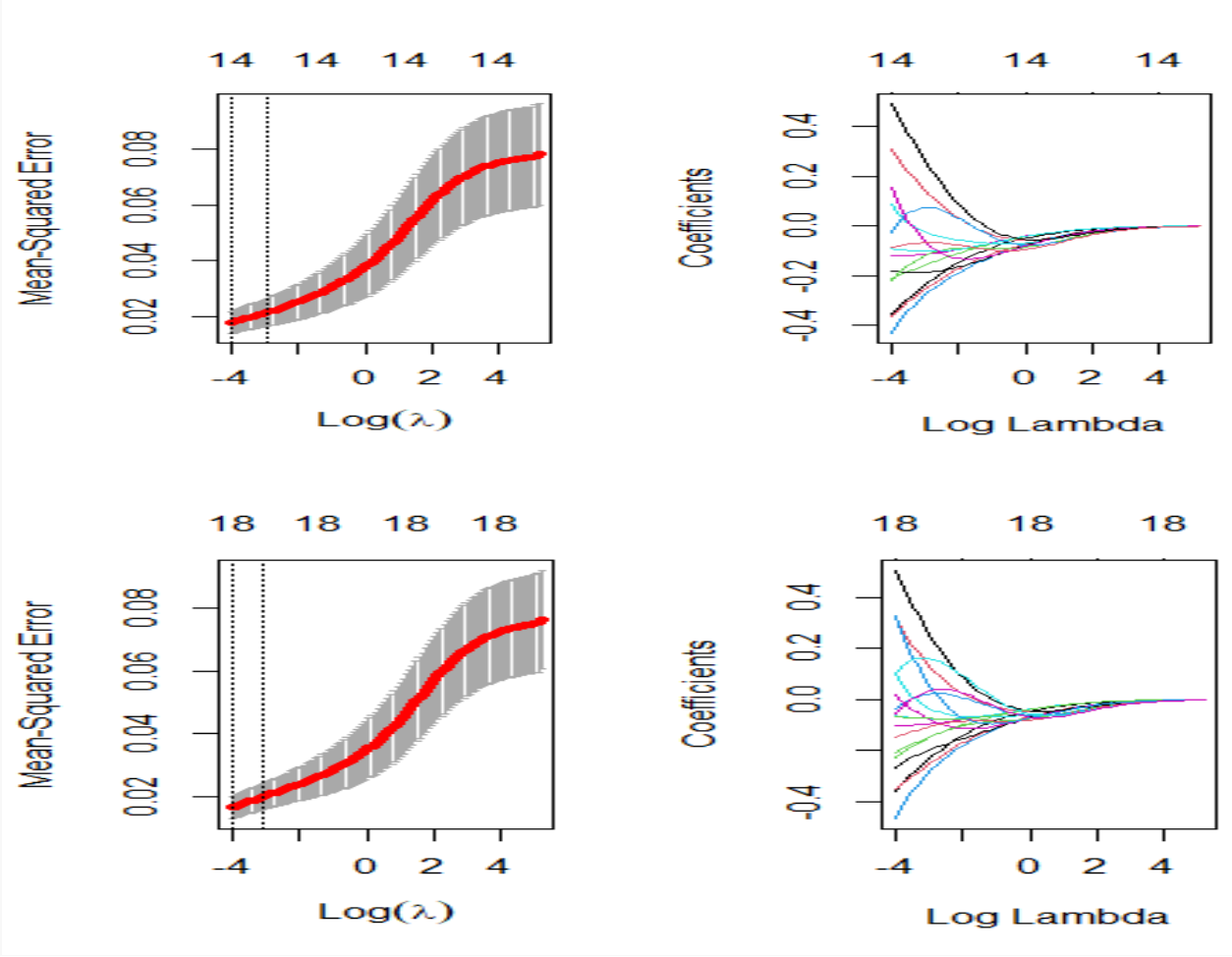
Coefficients
-0.7
-0.5
-0.3
-0.1
2 2 2 2 2
-4
0
2
4
Log Lambda

Mean-Squared Error
0.02
0.04
0.06
0.08
0.10
5 5 5 5 5
-4
0
2
4
Log(λ)

Coefficients
-0.6
-0.4
-0.2
0.0
0.2
0.4
5 5 5 5 5
-4
0
2
4
Log Lambda

Mean-Squared Error
0.02
0.04
0.06
0.08
9 9 9 9 9
-4
0
2
4
Log(λ)

Coefficients
-0.4
-0.2
0.0
0.2
0.4
9 9 9 9 9
-4
0
2
4
Log Lambda



What we can also observe from the left plots above is how the MSE changes (increases) as lambda increases and, from the ones on the right, we can check how the coefficients are driven to 0 as lambda increases.

PROBLEM 5

I am now asked to use Lasso regression, another regression method that can be used for regularization, since it introduces a penalization term as Ridge does. Additionally, Lasso also works as a variable selector since it enforces the $\beta's$ to be exactly zero when their correspondent variables are not dramatically important in explaining the response.

Once again, as in Ridge, cross-validation comes in help to choose the best lambda in terms of the correspondent test MSE. This lambda will give us the vector of coefficients $\beta_j's$, which we can use to obtain predictions, R squared, etc...

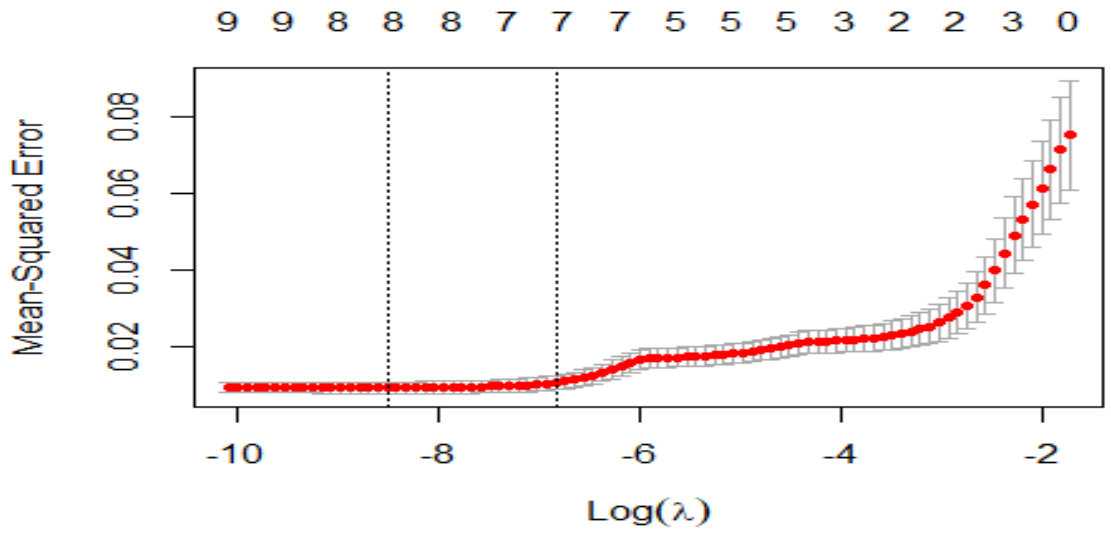
In this case I am using the built-in function of R contained in the library `glmnet`, which allows find the best lambdas for each degree of the polynomial.

```
library(glmnet)

pl <- matrix(NA,5,2)
dimnames(pl) <- list(1:5, c("order", "best lambda"))
for (order in 1:5){
  mt <- get(paste("matrix_deg",toString(order),sep=""))

  cv_model <- cv.glmnet(mt, trainingY, alpha = 1)
  best_lambda <- cv_model$lambda.min
  pl[order,] <- c(order, best_lambda)}
pl
```

```
##   order  best lambda
## 1     1 1.012725e-03
## 2     2 1.178065e-03
## 3     3 7.950602e-05
## 4     4 1.794467e-05
## 5     5 1.794467e-05
```

According to the default set of lambdas that the function analyzes, the values above are the best for the Lasso regression in terms of test MSEs. We can also observe how the MSE changes as the value of lambda increases. For this purpose, I now consider just one of the models (the third order one) to focus on the interpretation of the plot:

```
matt <- matrix_deg3
colnames(matt) <- c("int", "x1", "x2", "x1^2", "x2^2", "x1*x2", "x1^3"
, "x2^3", "(x1^2)*x2", "x1*(x2^2)")
cv_model3 <- cv.glmnet(matt, trainingY, alpha = 1)
plot(cv_model3)
```

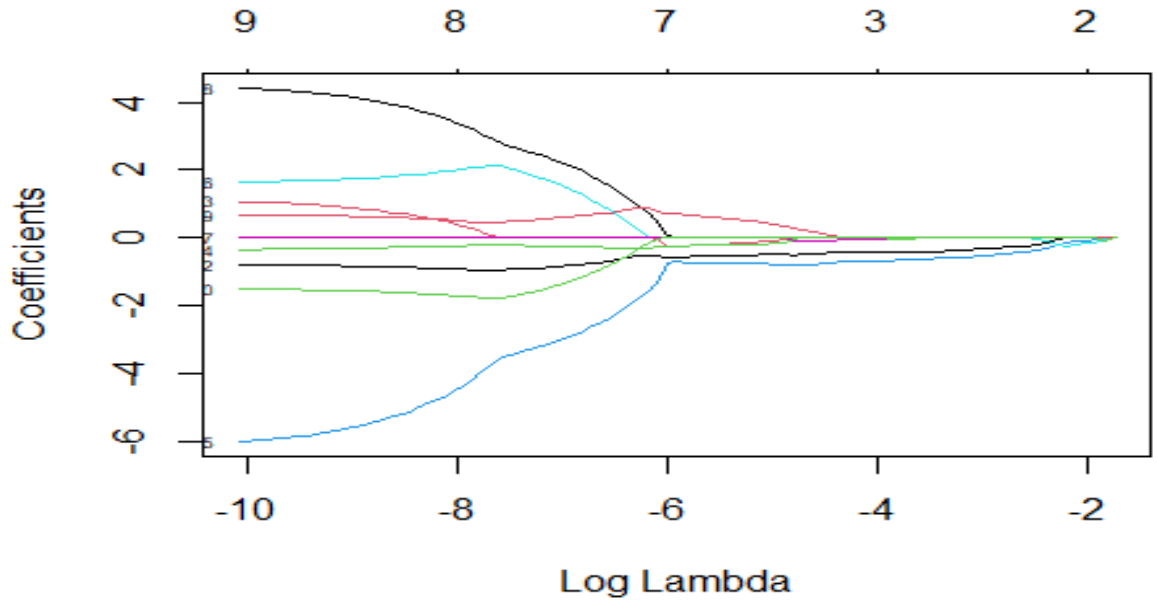


As we can see, as lambda increases (and so does its logarithm, which is what the function uses), the MSE decrease just in the very first place (reaching lambda min where the first dotted line is) and then increases monotonically until all the coefficients are equal to zero. When it happens (not in figure), the MSE obviously becomes constant. It is important to notice that when we penalize too much, we also remove variables that should be kept and, therefore, the predictions become less accurate and the test MSE increases.

It is also possible to notice that as lambda increases, the standard error of the MSE increases as well (the upper and lower bars for each point are the MSE +/- its SE), and so its estimate gets less precise.

We can also see what happens in terms of coefficients:

```
plot(cv_model3$glmnet.fit, xvar="lambda", label=5)
```



We can see here that as the value of lambda increases, more and more $\beta's$ are shrinked to zero. In particular we can see that as log(λ) get close to -6, almost all of the coefficients are shrunk to zero, almost at the same time. After that, only four are kept in the model, three of which stay for high values of lambda. Here we can see the power of Lasso, which is the ability of doing a variable selection while estimating the coefficients.

PROBLEM 6

The dataset I am using stores the latitude and longitude of Airbnb's located in New York in 2021, along with their prices. Each line corresponds to a different Airbnb. It may be useful to use this dataset to predict the cost of the Airbnb according to its location in the city of New York.

```
d <- read.table("C:/Users/cmira/Desktop/ny.csv",
                header = TRUE,
                row.names = 1,
                sep = ",")
df <- d[,c(1,2,3)]

require(skimr)

myskim2 <- skim_with(base = sfl(),
                     numeric = sfl(hist = NULL))
myskim2(df)
```

Data summary

Name	df
Number of rows	48895
Number of columns	3
_____	
Column type frequency:	
numeric	3
_____	
Group variables	None

Variable type: numeric

skim_variable	mean	sd	p0	p25	p50	p75	p100
latitude	40.73	0.05	40.50	40.69	40.72	40.76	40.91
longitude	-73.95	0.05	-74.24	-73.98	-73.96	-73.94	-73.71
price	152.72	240.15	0.00	69.00	106.00	175.00	10000.00

As we can see, the latitude goes from 40.5 to 40.9 while the longitude is from -74.2 to -73.7. These values correspond to the area of New York I'm studying. The price, which is the response variable, has a positive asymmetric distribution, as we can see, since 75% of the observations are smaller of very close to the mean value (153 USD), while there is a long tail that goes to 10000USD. Since the lowest price is 0, it is possible to say that this is probably due to an error made while filling the dataset.

We can also observe that, since the variable *price* has a different magnitude, its standard deviation is way higher than that of the other two variables.

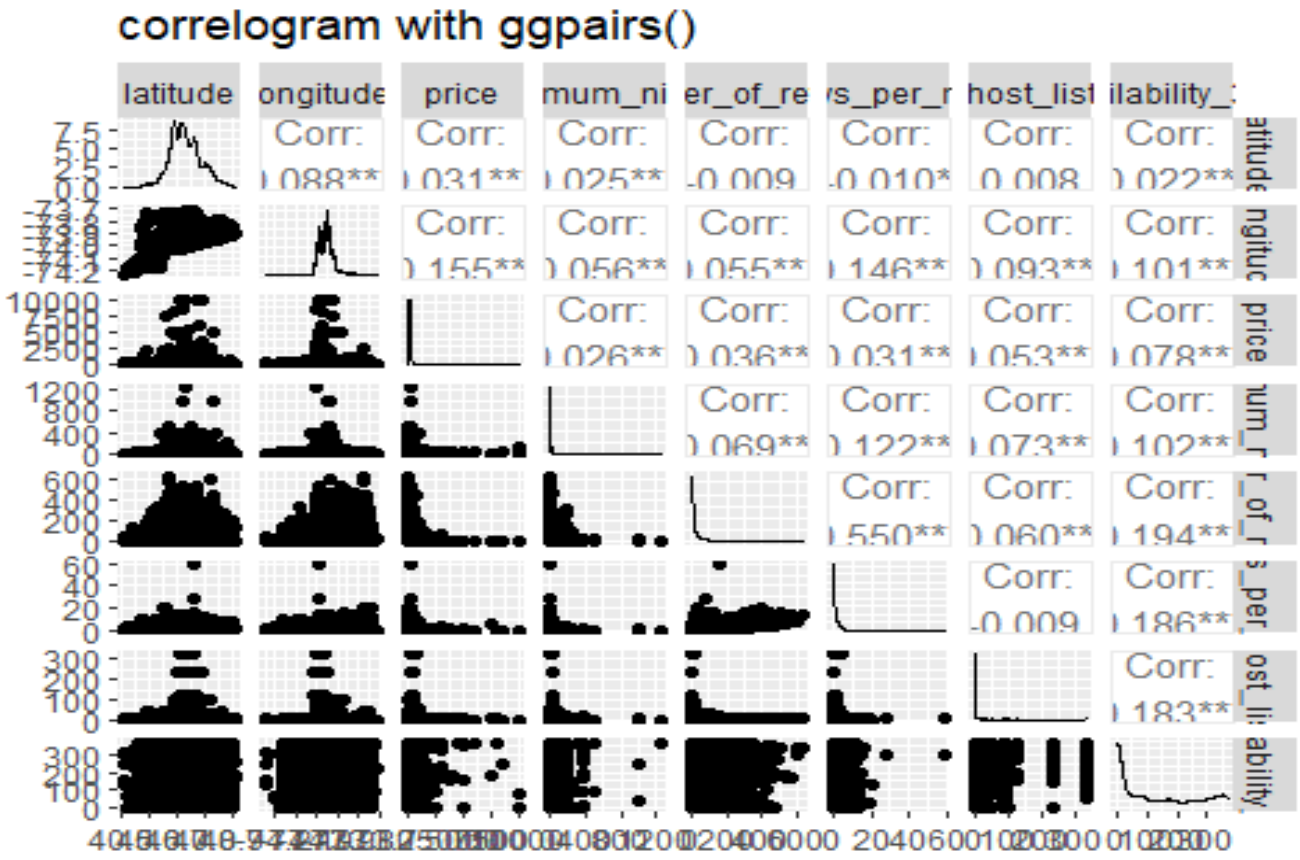
Just to explore the dataset, we can also check to which Airbnb corresponds the highest price.

```
df[which.max(df$price),]
```

```
##         latitude longitude price
## 7003697  40.7681 -73.91651 10000
```

Looking on Maps, we can check that the Airbnb with the highest price has the address "*25-64 35th Street, New York*".

```
require(GGally)

ggpairs(df_complete, title="correlogram with ggpairs()")
```



The above matrix is shown to have a visualization of the variables' distributions (on the diagonal), their correlations (upper right) and scatterplots (lower right). Looking at the variable *price*, we can see that it doesn't seem to be highly correlated to the used variables, if taken singularly, but we can show that using interactions and different polynomials will lead to good results (low test MSE).

Here is the scaled version I will use to obtain better results:

```
df2 <- as.data.frame(round(cbind(scaling(cbind(df$latitude)), scaling(
cbind(df$longitude)), scaling(cbind(df$price))),6))
dimnames(df2) <- list(rownames(df), colnames(df))

mysplitted <- train_test(df2[,c(1,2)],cbind(df2$price))

mytrainingX <- mysplitted[[1]]
mytrainingY <- mysplitted[[2]]
mytestX <- mysplitted[[3]]
mytestY <- mysplitted[[4]]

mymatrix1 <- model_matrix(mytrainingX[,1],mytrainingX[,2],1)
rownames(mymatrix1) <- translate(rownames(mytrainingX))
mytestX_1 <- as.data.frame(model_matrix(mytestX$V1,mytestX$V2,1))
mybeta1 <- betas(mymatrix1, mytrainingY)
mypred1 <- predictions(mytestX[,1],mytestX[,2],1,mybeta1)
mymse1 <- mean((mytestY - mypred1)^2)
myr21 <- 1 - sum((mytestY - mypred1)^2)/sum((mytestY - mean(mytestY))^
2)

mymatrix2 <- model_matrix(mytrainingX[,1],mytrainingX[,2],2)
rownames(mymatrix2) <- translate(rownames(mytrainingX))
mytestX_2 <- as.data.frame(model_matrix(mytestX$V1,mytestX$V2,2))
mybeta2 <- betas(mymatrix2, mytrainingY)
mypred2 <- predictions(mytestX[,1],mytestX[,2],2,mybeta2)
mymse2 <- mean((mytestY - mypred1)^2)
myr22 <- 1 - sum((mytestY - mypred2)^2)/sum((mytestY - mean(mytestY))^
2)

mymatrix3 <- model_matrix(mytrainingX[,1],mytrainingX[,2],3)
rownames(mymatrix3) <- translate(rownames(mytrainingX))
mytestX_3 <- as.data.frame(model_matrix(mytestX$V1,mytestX$V2,3))
mybeta3 <- betas(mymatrix3, mytrainingY)
mypred3 <- predictions(mytestX[,1],mytestX[,2],3,mybeta3)
mymse3 <- mean((mytestY - mypred3)^2)
myr23 <- 1 - sum((mytestY - mypred3)^2)/sum((mytestY - mean(mytestY))^
2)

mymatrix4 <- model_matrix(mytrainingX[,1],mytrainingX[,2],4)
rownames(mymatrix4) <- translate(rownames(mytrainingX))
mytestX_4 <- as.data.frame(model_matrix(mytestX$V1,mytestX$V2,4))
mybeta4 <- betas(mymatrix4, mytrainingY)
mypred4 <- predictions(mytestX[,1],mytestX[,2],4,mybeta4)
```

```
mymse4 <- mean((mytestY - mypred4)^2)
myr24 <- 1 - sum((mytestY - mypred4)^2)/sum((mytestY - mean(mytestY))^
2)

mymatrix5 <- model_matrix(mytrainingX[,1],mytrainingX[,2],5)
rownames(mymatrix5) <- translate(rownames(mytrainingX))
mytestX_5 <- as.data.frame(model_matrix(mytestX$V1,mytestX$V2,5))
mybeta5 <- betas(mymatrix5, mytrainingY)
mypred5 <- predictions(mytestX[,1],mytestX[,2],5,mybeta5)
mymse5 <- mean((mytestY - mypred5)^2)
myr25 <- 1 - sum((mytestY - mypred5)^2)/sum((mytestY - mean(mytestY))^
2)

myresults <- matrix(data=NA, nrow=2,ncol=5)
rownames(myresults)<-cbind("MSE","R2")
mynames<-c()
for (i in 1:5){
  mynames[i] <- paste("order", as.character(i))
}
colnames(myresults) <- mynames

myresults[1,] <- rbind(mymse1, mymse2, mymse3, mymse4, mymse5)
myresults[2,] <- rbind(myr21, myr22, myr23, myr24, myr25)

myresults

##        order 1   order 2    order 3    order 4    order 5
## MSE 1.02305207 1.0230521 0.99249256 0.98766670 0.98645598
## R2  0.02873098 0.0500922 0.05774369 0.06232528 0.06347472
```

From the data I have, the best model would probably be one of the last two. In terms of the values I obtained, it would be the 5th order polynomial, but the differences between that and the previous order are minimal, so it may be cleverer to use a “simpler” model.

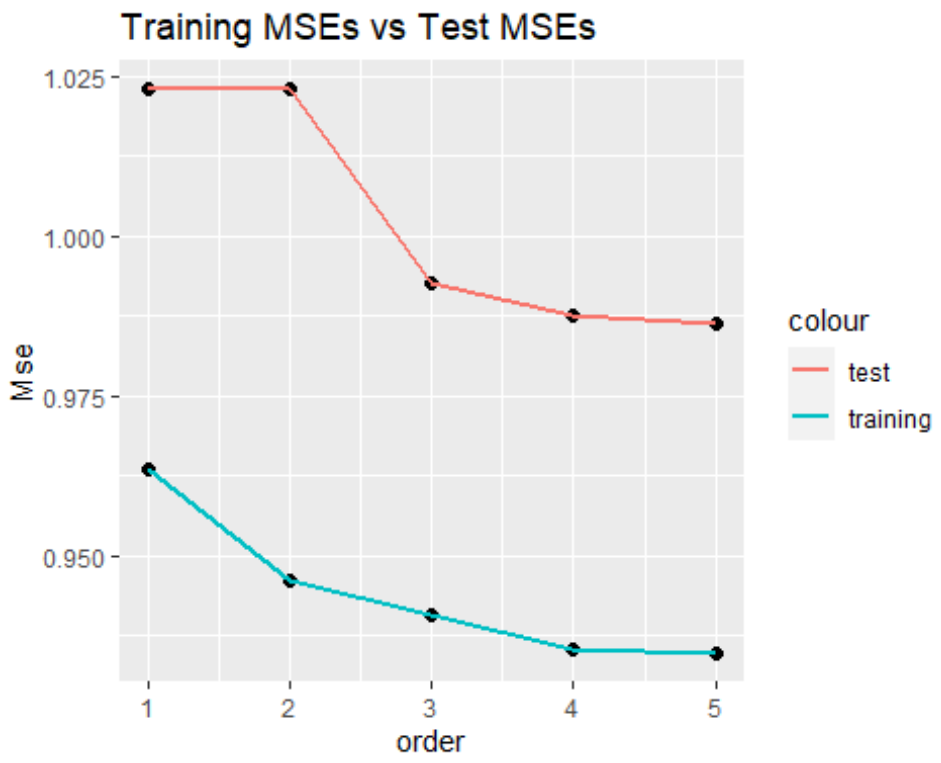
In particular, we can say that the last two models are able to predict around 62/63% (could be better) of the prices of the Airbnb’s of the city thanks to their latitude and longitude.

We can also see that the test MSE, which measures the errors done in predicting the test data with the model we used, are close to 1. This is not an optimal value, since the MSE has 0 as lower boundary, but it is not too high either, considering that the MSE has no upper limit.

We can now check if the model we used is a good one in terms of over or underfitting, that is, we can check if it sticks too much to the data I trained it on or if it is able to generalize the results.

```
mypredtrain1 <- predictions(mytrainingX[,1],mytrainingX[,2],1,mybeta1)
mymsetrain1 <- mean((mytrainingY - mypredtrain1)^2)
mypredtrain2 <- predictions(mytrainingX[,1],mytrainingX[,2],2,mybeta2)
```

```
mymsetrain2 <- mean((mytrainingY - mypredtrain2)^2)
mypredtrain3 <- predictions(mytrainingX[,1],mytrainingX[,2],3,mybeta3)
mymsetrain3 <- mean((mytrainingY - mypredtrain3)^2)
mypredtrain4 <- predictions(mytrainingX[,1],mytrainingX[,2],4,mybeta4)
mymsetrain4 <- mean((mytrainingY - mypredtrain4)^2)
mypredtrain5 <- predictions(mytrainingX[,1],mytrainingX[,2],5,mybeta5)
mymsetrain5 <- mean((mytrainingY - mypredtrain5)^2)

mymsematrix <- data.frame(1:5, rbind(mymse1,mymse2,mymse3,mymse4,mymse
5),rbind(mymsetrain1,mymsetrain2,mymsetrain3,mymsetrain4,mymsetrain5))
colnames(mymsematrix) <- c("order", "test", "train")
rownames(mymsematrix) <- c(1:5)

ggplot()+
  geom_point(data=mymsematrix, aes(x=order, y=test), size=2)+
  geom_line(data=mymsematrix, aes(x=order, y=test, colour="test"), siz
e=1)+
  geom_point(data=mymsematrix, aes(x=order, y=train), size=2)+
  geom_line(data=mymsematrix, aes(x=order, y=train, colour="training")
, size=1)+
  ylab("Mse")+ggtitle("Training MSEs vs Test MSEs")
```




```
mymsematrix
```

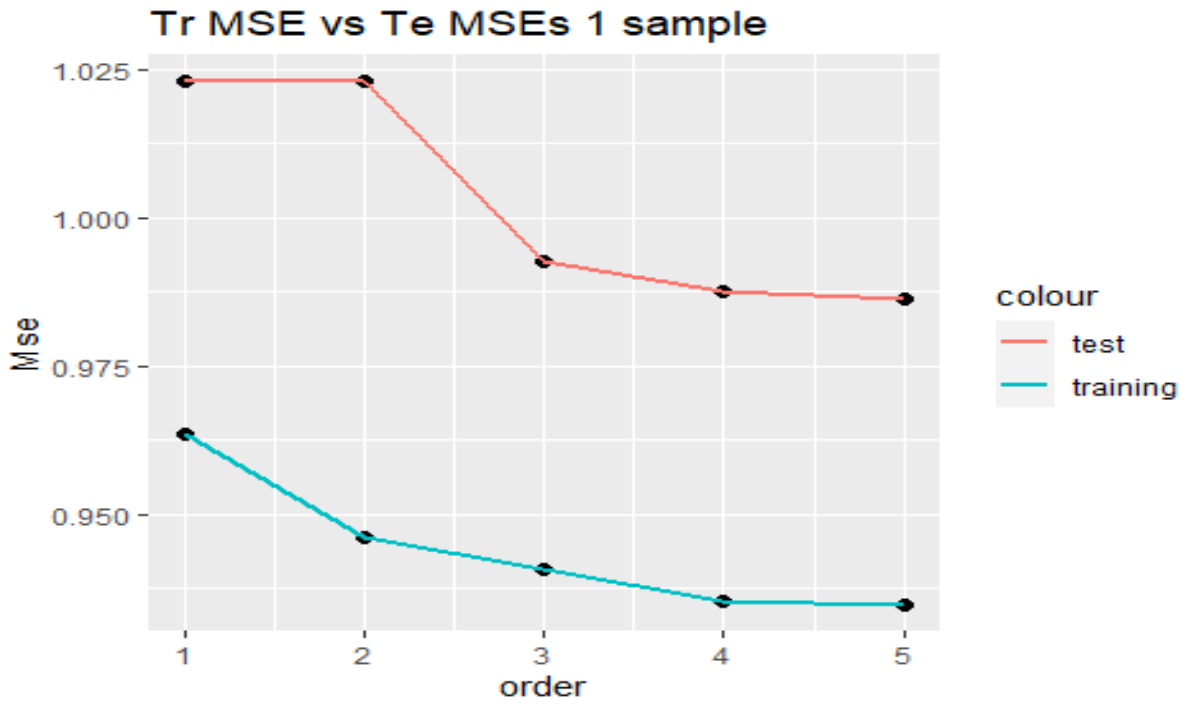
```
##   order      test     train
## 1     1 1.0230521 0.9633994
## 2     2 1.0230521 0.9461429
## 3     3 0.9924926 0.9406793
## 4     4 0.9876667 0.9353817
## 5     5 0.9864560 0.9346341
```

Even if the plot shows two lines that looks extremely far from one from another, it is essential to look at the values themselves. In fact, the graph might be misleading since the baseline is not set to 0 and, therefore, the two lines look further than they actually are. Looking at the values, we can see that, as "usual", the train MSEs are lower that the test ones, but they are not that far (they differ of 5% circa). As in the test situation, also for the train set the lowest MSE is obtained on the fifth order of the polynomial.

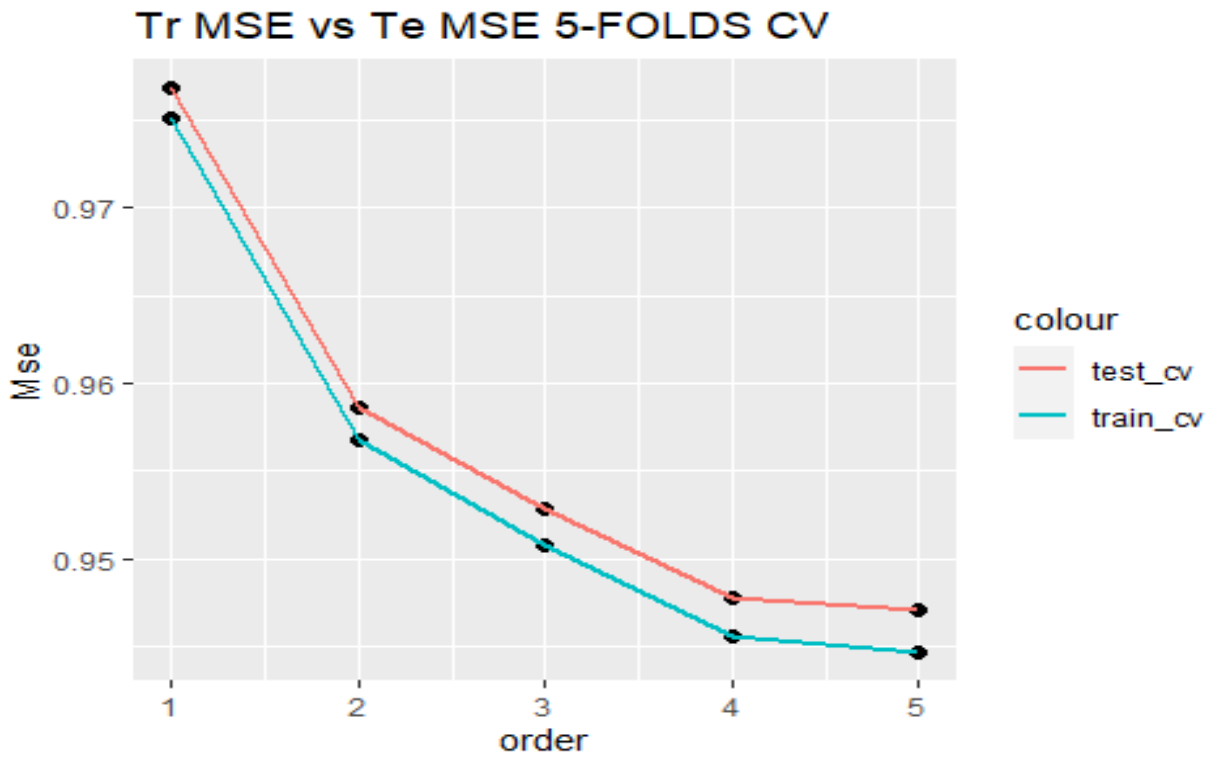
In order to have more precise results, I am now implementing the cross-validation method:

```
mynumfolds <- c(5,10)
mymatrixlist <- list()
mymsematrix_cv <- as.data.frame(matrix(NA, nrow = 5,3))
colnames(mymsematrix_cv) <- c("order", "test_cv", "train_cv")


for (k in 1:length(mynumfolds)){
  myfolds<-cut(seq(1,nrow(df2)),breaks=mynumfolds[k],labels=FALSE)

  for (degree in 1:5){

    mymsestest <- c()
    mymsestrain <- c()

    for(i in 1:mynumfolds[k]){

      mytestIndexes_cv <- which(myfolds==i,arr.ind=TRUE)
      mytestData_cv <- df2[mytestIndexes_cv, ]
      mytrainData_cv <- df2[-mytestIndexes_cv, ]
      mydataset_cv<-model_matrix(mytrainData_cv[,1],mytrainData_cv[,2]
,degree)
      mybeta_cv <- betas(mydataset_cv, mytrainData_cv[,3])
      mypredmse_cv <- predictions(mytestData_cv[,1],mytestData_cv[,2],
degree,mybeta_cv)
      mymsetest_cv <- mean((mytestData_cv[,3] - mypredmse_cv)^2)
      mypredtrain_cv<-predictions(mytrainData_cv[,1],mytrainData_cv[,2
],degree,mybeta_cv)
      mymsetrain_cv <- mean((mytrainData_cv[,3] - mypredtrain_cv)^2)
      mymsestest[i] <- mymsetest_cv
      mymsestrain[i] <- mymsetrain_cv
    }
```

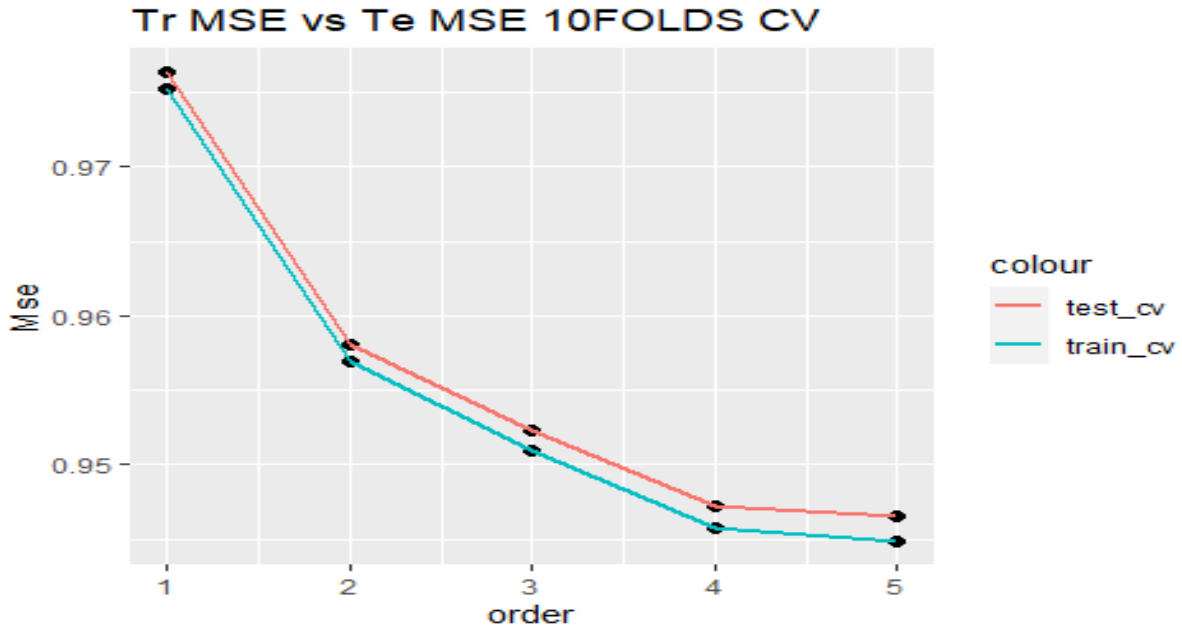
```
    mymsematrix_cv[degree,] <- c(degree, mean(mymsestest), mean(mymses
train))
  }

  mymatrixlist[[k]] <- mymsematrix_cv
}
myfivefolds<- mymatrixlist[[1]]
mytenfolds <- mymatrixlist[[2]]

ggplot()+  geom_point(data=mymsematrix, aes(x=order, y=test), size=2)+
geom_line(data=mymsematrix, aes(x=order, y=test, colour="test"), size=
1)+  geom_point(data=mymsematrix, aes(x=order, y=train), size=2)+  geo
m_line(data=mymsematrix, aes(x=order, y=train, colour="training"), siz
e=1)+  ylab("Mse")+ggtitle("Tr MSE vs Te MSEs 1 sample")
```




```
ggplot()+  geom_point(data=myfivefolds, aes(x=order, y=test_cv), size=
2)+  geom_line(data=myfivefolds, aes(x=order, y=test_cv, colour="test_
cv"), size=1)+  geom_point(data=myfivefolds, aes(x=order, y=train_cv),
size=2)+  geom_line(data=myfivefolds, aes(x=order, y=train_cv, colour=
"train_cv"), size=1)+  ylab("Mse")+ggtitle("Tr MSE vs Te MSE 5-FOLDS C
V")
```




```
ggplot()+  geom_point(data=mytenfolds, aes(x=order, y=test_cv), size=2
)+  geom_line(data=mytenfolds, aes(x=order, y=test_cv, colour="test_cv
"), size=1)+  geom_point(data=mytenfolds, aes(x=order, y=train_cv), si
ze=2)+  geom_line(data=mytenfolds, aes(x=order, y=train_cv, colour="tr
ain_cv"), size=1)+  ylab("Mse")+ggtitle("Tr MSE vs Te MSE 10FOLDS CV")
```



We can now see that, as the number of folds increases, the Test and Train MSEs get similar, and it shows that the models we are using are actually good, and that we "just needed more samples" to really evaluate the mean squared errors. As before, the train MSE is smaller than the test MSE for each polynomial, but they get closer and closer as the number of used folds increases. What we can also say is that, again, as the order of the polynomial increases, both the train and test MSE decrease, which means that a more complex model might be, in this case, a better choice in terms of explaining (train MSE) and predicting (test MSE) the response. Anyway, is still necessary to observe that, looking at the cross-validation results, even though the MSE are decreasing, the values obtained from the 2nd order to the 5th order polynomial are very close to each other. That is, I could also choose a simpler model to fit the data I am working with.

Note that I did not use the n-fold here since I had too many observations, but the values would be even more accurate in that scenario.

Here follows the Ridge regression and, after that, the Lasso as well:

```
ridgematrix <- matrix(NA,5,3)
dimnames(ridgematrix) <- list(1:5, c("order", "best lambda", "mse"))
for (order in 1:5){
  mymt <- get(paste("mymatrix",toString(order),sep=""))
  mycv_ridge <- cv.glmnet(mymt, mytrainingY, alpha = 0, nfolds = 10)
  myridgemse <- min(mycv_ridge$cvm)
  myoptimal_lambda <- mycv_ridge$lambda.min
  ridgematrix[order,] <- c(order, myoptimal_lambda, myridgemse)
}
ridgematrix
```

```
##   order best lambda       mse
## 1     1  0.01458301 0.9635402
## 2     2  0.01458301 0.9465037
## 3     3  0.01458301 0.9413602
## 4     4  0.01458301 0.9363552
## 5     5  0.01458301 0.9364569
```

```
lassomatrix <- matrix(NA,5,3)
dimnames(lassomatrix) <- list(1:5, c("order", "best lambda", "mse"))
for (order in 1:5){
  mymt <- get(paste("mymatrix",toString(order),sep=""))

  mycv_lasso <- cv.glmnet(mymt, mytrainingY, alpha = 1,  nfolds = 10)
  mylassomse <- min(mycv_lasso$cvm)
  mybest_lambda <- mycv_lasso$lambda.min
  lassomatrix[order,] <- c(order, mybest_lambda, mylassomse)
  }
lassomatrix
```

```
##   order  best lambda       mse
## 1     1 6.613218e-04 0.9635799
## 2     2 3.784327e-04 0.9465236
## 3     3 1.028803e-04 0.9412718
## 4     4 1.458301e-05 0.9360320
## 5     5 1.458301e-05 0.9354310
```

```
final <- cbind(mytenfolds$test_cv, ridgematrix[,"mse"], lassomatrix[,"
mse"])
colnames(final) <- c("OLS Mse", "Ridge Mse", "Lasso Mse")
final
```

```
##     OLS Mse Ridge Mse Lasso Mse
## 1 0.9763609 0.9635402 0.9635799
## 2 0.9580988 0.9465037 0.9465236
## 3 0.9523189 0.9413602 0.9412718
## 4 0.9472579 0.9363552 0.9360320
## 5 0.9465812 0.9364569 0.9354310
```

Summing up, we can see that as we introduce a penalizing term λ into the model, the final test MSE we obtain does decrease slightly. This means that if we accept to increase the bias of the model, we can reduce its variance. As we can see though, the MSE does not decrease much neither with Ridge nor with Lasso, which means that we do not significantly explain more variance than the one explained implementing the OLS.

In all the three situations we can see that the lowest test MSE is obtained for the 5^{th} order polynomial, and that the 4^{th} order has results that are actually very close to these. We can finally say that the penalizations that occurred in Ridge and Lasso are just minimal ones: the two variables - *latitude* and *longitude* - are kept in the five models almost at their "full potential". That is, the variables are quite important in explaining the behavior of the price of the Airbnb, and in predicting it as well.

To check if these results might differ, I am now implementing these methods once again, using linear regressions with a dataset that contains more variables. Along with latitude and longitude, now I also have 1) the number of minimum nights that one must spend in the Airbnb, 2) the number of reviews, 3) the number of reviews per month, 4) the number of properties for that Airbnb and 5) the number of available days during the year.

```
df_complete <- read.table("C:/Users/cmira/Desktop/AB_NYC_2019.csv",
                  header = TRUE,
                  row.names = 1,
                  sep = ",")
df_complete <- df_complete[complete.cases(df_complete), ]
head(df_complete)
```

```
##      latitude longitude price minimum_nights number_of_reviews
## 2539    40.65    -73.97   149              1                 9
## 2595    40.75    -73.98   225              1                45
## 3831    40.69    -73.96    89              1               270
## 5022    40.80    -73.94    80             10                 9
## 5099    40.75    -73.98   200              3                74
## 5121    40.69    -73.96    60             45                49
##      reviews_per_month calculated_host_listings_count availability_365
## 2539              0.21                               6              365
## 2595              0.38                               2              355
## 3831              4.64                               1              194
## 5022              0.10                               1                0
## 5099              0.59                               1              129
## 5121              0.40                               1                0
```

```
#scaling

df_complete2 <- as.data.frame(round(cbind(scaling(cbind(df_complete$la
titude)), scaling(cbind(df_complete$longitude)), scaling(cbind(df_comp
lete$price)), scaling(cbind(df_complete$minimum_nights)), scaling(cbin
d(df_complete$number_of_reviews)), scaling(cbind(df_complete$reviews_p
er_month)), scaling(cbind(df_complete$calculated_host_listings_count))
, scaling(cbind(df_complete$availability_365))),6))
dimnames(df_complete2) <- list(rownames(df_complete), colnames(df_comp
lete))

splitols <- train_test(df_complete2[,c(1,2,4,5,6,7,8)],cbind(df_comple
te2$price))
model <- lm(formula = splitols[[2]] ~ .-1, data = splitols[[1]])
mseols <- mean((splitols[[4]] - predict(model, newdata = splitols[[3]]
))^2)

Amynumfolds <- c(5,10)
Amylist <- list()
Amymsematrix_cv <- as.data.frame(matrix(NA, nrow = 5,3))
colnames(Amymsematrix_cv) <- c("order", "test_cv", "train_cv")

#cross-validation


for (k in 1:length(Amynumfolds)){
  Amyfolds<-cut(seq(1,nrow(df_complete2)),breaks=Amynumfolds[k],labels
=FALSE)

    Amymsestest <- c()
    Amymsestrain <- c()

    for(i in 1:Amynumfolds[k]){

      AmytestIndexes_cv <- which(Amyfolds==i,arr.ind=TRUE)
```

```
      AmytestData_cv <- df_complete2[AmytestIndexes_cv, ]
      AmytrainData_cv <- df_complete2[-AmytestIndexes_cv, ]
      mod <- lm(price ~.-1, data = AmytrainData_cv)
      Apredtrain_ <- predict(mod)
      Apredtest_ <- predict(mod, newdata = AmytestData_cv)
      Amymsetest_cv <- mean((AmytestData_cv[,3] - Apredtest_)^2)
      Amymsetrain_cv <- mean((AmytrainData_cv[,3] - Apredtrain_)^2)
      Amymsestest[i] <- Amymsetest_cv
      Amymsestrain[i] <- Amymsetrain_cv
    }

      Amylist[[k]] <- c(mean(Amymsestest), mean(Amymsestrain))
}
fivefolds_complete<- Amylist[1]
tenfolds_complete <- Amylist[2]

fivefolds_complete; tenfolds_complete
```

```
## [[1]]
## [1] 0.9668952 0.9620201

## [[1]]
## [1] 0.9645273 0.9622237
```

```
#ridge and lasso regression

X = model.matrix(price ~ .-1, data = df_complete2)
ridge_complete <- cv.glmnet(X, df_complete2$price, alpha = 0, nfolds =
10)
ridgecomp_mse <- min(ridge_complete$cvm)
ridgecom_optimal_lambda <- ridge_complete$lambda.min
ridgeresults <- c(ridgecom_optimal_lambda, ridgecomp_mse)

ridgeresults
```

```
## [1] 0.01553603 0.96321345
```

```
X = model.matrix(price ~ .-1, data = df_complete2)
lasso_complete <- cv.glmnet(X, df_complete2$price, alpha = 1, nfolds =
10)
lassocomp_mse <- min(lasso_complete$cvm)
lassocomp_optimal_lambda <- lasso_complete$lambda.min
lassoresults <- c(lassocomp_optimal_lambda, lassocomp_mse)
lassoresults
```

```
## [1] 0.004126506 0.962639124
```

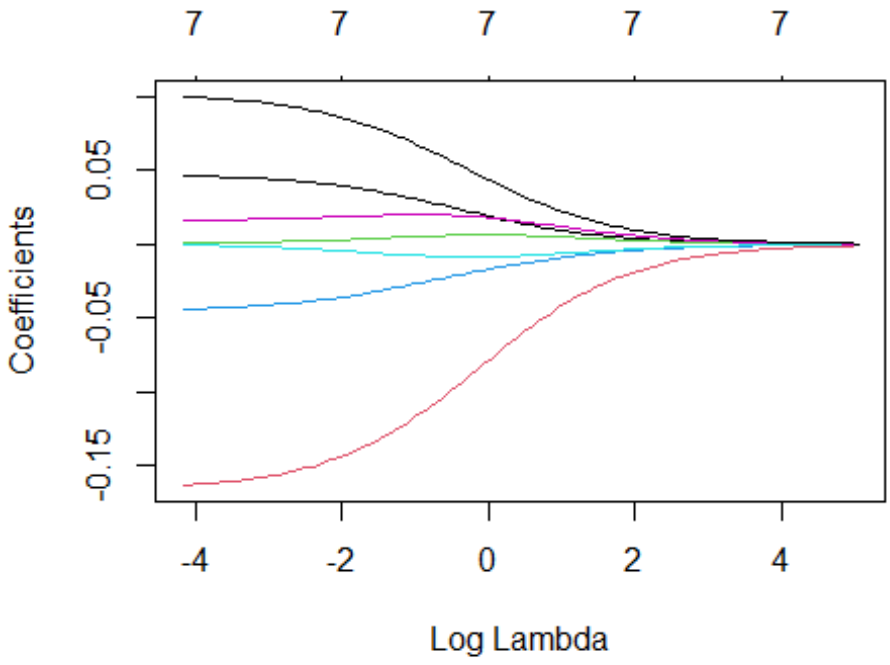
```
finalcomparisons <- cbind(tenfolds_complete[[1]][1], ridgeresults[2],
lassoresults[2])
colnames(finalcomparisons) <- c("cv 10folds", "ridge", "lasso");
finalcomparisons
```

```
##      cv 10folds     ridge     lasso
## [1,]  0.9645273 0.9632135 0.9626391
```

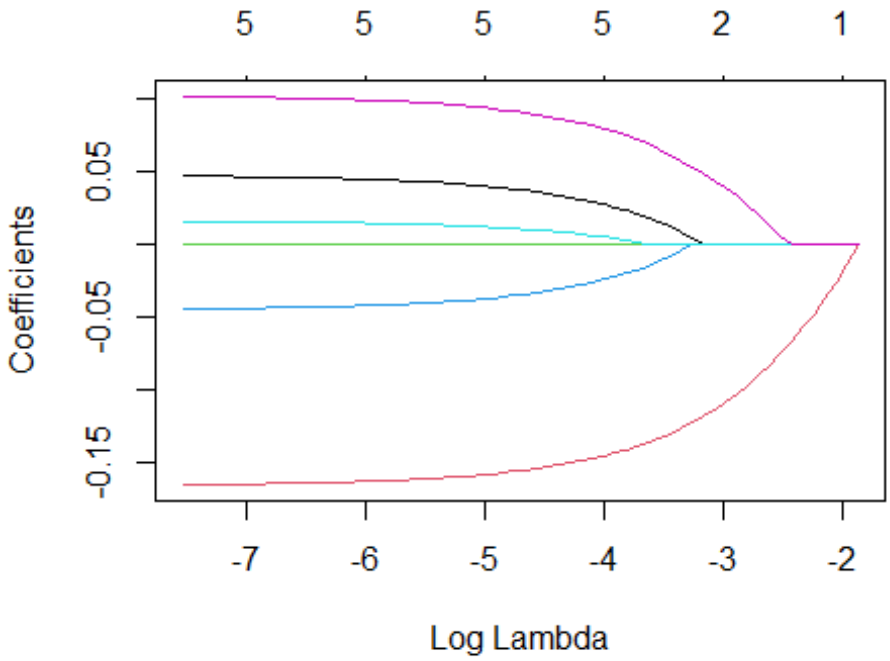
As we can see, the differences between the three methods here are, again, not significant.

As we can show in the plots below, there is no variable that is shrunk to 0 “straight away”, which means that they are all pretty useful to explain and predict the price of the Airbnb.

```
plot(Amycv_ridge $glmnet.fit, "lambda")
```




```
plot(Bmycv_lasso $glmnet.fit, "lambda")
```



In this case we can therefore say that these seven variables, taken at their first degree, make a good model in explaining the variable price. In short, this means that each of them gives some information about how much the price of the Airbnb, located in a certain position of New York, will be.

Knowing this can be very useful both for hosts and clients. For the host, for example, it might be clever to use these variables to understand how much the "competitors" charge the customers and, in this way, he can decide how high he can set the price of its apartment. Therefore, he will use this model to predict the price he should set.

On the other hand, these variables might be used by Airbnb to tell the customer how much he would pay hist staying according to the information inserted in the website page. Based on that, the customer can "act on these variables" and obtain a price that fits his desires.

Moving forward, the MSE here might be a symptom of how much the price of an Airbnb located in a certain place and with certain characteristics will differ from the one the guest would expect. This could either be positive or negative for the customer, who might find a real deal in a rich area, or a super expensive place in a "normal" area. On the other hand, for the host, using a model with a high MSE might make him set a price that is too high for the place the apartment is, and this will leave him with no guests.

The models I presented before, including the penalized ones, give MSEs that are not too high, and that is of course positive for the observation just made. Despite that, it is possible to imagine that the way some of the variables interact could lead to some changes in the predictions. Therefore, I think it could be useful to study some more models, in which I included

all the possible interactions between the variables, along with different degrees of them. Once I will find the best model in terms of test MSE, I will do I variable selection with the backward selection, to remove those variables that might give the same information, due to collinearity or other causes. For the record, I am not implementing cross-validation here, because it will take too much to do it with R.

```
nomi <- c(colnames(df_complete)[-3])
df_noprice <- df_complete[,-3]

n <- 1
newdf1 <- cbind.data.frame(df_complete$price, poly(as.matrix(df_nopric
e), degree = n, raw = TRUE))
store1 <- colnames(newdf1)
names(newdf1)[-1]<-paste0("x",c(1:7))
names(newdf1)[1]<-"prezzo"

newdf21<- round(as.data.frame(scale(newdf1)),3)

split1 <- train_test(newdf21[,c(2,3,4,5,6,7,8)],cbind(newdf21$prezzo))
trainx1 <- split1[[1]]
trainy1 <- split1[[2]]
testx1 <- split1[[3]]
testy1 <- split1[[4]]

polylm1 <- lm(trainy1 ~ ., trainx1)
prd1 <- predict(polylm1, newdata = testx1)
msee1 <- mean((testy1 - prd1)^2)
msee1

## [1] 0.5781675

n <- 2
newdf2 <- cbind.data.frame(df_complete$price, poly(as.matrix(df_nopric
e), degree = n, raw = TRUE))
store2 <- colnames(newdf2)
names(newdf2)[-1]<-paste0("x",c(1:35))
names(newdf2)[1]<-"prezzo"

newdf22<- as.data.frame(scale(newdf2))
split2 <- train_test(newdf22[,c(2:35)],cbind(newdf22$prezzo))
trainx2 <- split2[[1]]
trainy2 <- split2[[2]]
testx2 <- split2[[3]]
testy2 <- split2[[4]]

polylm2 <- lm(trainy2 ~ ., trainx2)
prd2 <- predict(polylm2, newdata = testx2)
```

```r
msee2 <- mean((testy2 - prd2)^2)
msee2
```

```
## [1] 0.5550931
```

```r
n <- 3
newdf3 <- cbind.data.frame(df_complete$price, poly(as.matrix(df_nopric
e), degree = n, raw = TRUE))
store3 <- colnames(newdf3)
names(newdf3)[-1]<-paste0("x",c(1:119))
names(newdf3)[1]<-"prezzo"

newdf23<- as.data.frame(scale(newdf3))
split3 <- train_test(newdf23[,c(2:119)],cbind(newdf23$prezzo))
trainx3 <- split3[[1]]
trainy3 <- split3[[2]]
testx3 <- split3[[3]]
testy3 <- split3[[4]]

polylm3 <- lm(trainy3 ~ ., trainx3)
prd3 <- predict(polylm3, newdata = testx3)
msee3 <- mean((testy3 - prd3)^2)
msee3
```

```
## [1] 0.549144
```

```r
n <- 4
newdf4 <- cbind.data.frame(df_complete$price, poly(as.matrix(df_nopric
e), degree = n, raw = TRUE))
store4 <- colnames(newdf4)
names(newdf4)[-1]<-paste0("x",c(1:329))
names(newdf4)[1]<-"prezzo"

newdf24<- as.data.frame(scale(newdf4))
split4 <- train_test(newdf24[,c(2:329)],cbind(newdf24$prezzo))
trainx4 <- split4[[1]]
trainy4 <- split4[[2]]
testx4 <- split4[[3]]
testy4 <- split4[[4]]

polylm4 <- lm(trainy4 ~ ., trainx4)
prd4 <- predict(polylm4, newdata = testx4)
msee4 <- mean((testy4 - prd4)^2)
msee4
```

```
## [1] 0.5541533
```

```
n <- 5
newdf5 <- cbind.data.frame(df_complete$price, poly(as.matrix(df_nopric
e), degree = n, raw = TRUE))
store5 <- colnames(newdf5)
names(newdf5)[-1]<-paste0("x",c(1:791))
names(newdf5)[1]<-"prezzo"

newdf25<- as.data.frame(scale(newdf5))
split5 <- train_test(newdf25[,c(2:791)],cbind(newdf25$prezzo))
trainx5 <- split5[[1]]
trainy5 <- split5[[2]]
testx5 <- split5[[3]]
testy5 <- split5[[4]]

polylm5 <- lm(trainy5 ~ ., trainx5)
prd5 <- predict(polylm5, newdata = testx5)
msee5 <- mean((testy5 - prd5)^2)
msee5

## [1] 2.057274

cbind(msee1,msee2,msee3,msee4,msee5)

##          msee1     msee2    msee3     msee4    msee5
## [1,] 0.5781675 0.5550931 0.549144 0.5541533 2.057274
```

We can see that among these five models, which have been obtained as multivariate polynomial regressions with interactions, the best models in terms of MSE in the third one. Since this model has 119 variables, I implemented a backward selection to find which are the best of these variable. (It took 25 minutes, so I would suggest not to try it).

This procedure studies different models obtained by removing one by one each variable and confronting the value obtained on the statistic AIC, which is used as a variable selector. The code is the following:

```
step(polylm3, direction = "backward")
```

The best model chosen with this procedure is the following:

```
finalpoly <- lm(trainy3 ~ x1 + x2 + x3 + x4 + x6 + x8 + x9 + x11 +
    x12 + x13 + x14 + x15 + x16 + x18 + x19 + x20 + x22 + x23 +
    x24 + x25 + x27 + x28 + x30 + x32 + x34 + x41 + x42 + x44 +
    x45 + x46 + x47 + x49 + x50 + x52 + x53 + x54 + x56 + x58 +
    x59 + x60 + x61 + x62 + x65 + x66 + x67 + x71 + x72 + x73 +
```

```
    x74 + x75 + x77 + x78 + x79 + x80 + x82 + x83 + x84 + x85 +
    x86 + x87 + x88 + x89 + x90 + x92 + x93 + x96 + x98 + x100 +
    x101 + x102 + x104 + x105 + x107 + x108 + x111 + x112 + x114 +
    x116 + x117, data = trainx3)
prdfin <- predict(finalpoly, newdata = testx3)
mseefin <- mean((testy3 - prdfin)^2)
mseefin
```

```
## [1] 0.5493735
```

Where these variables are obtained by multiplying different degrees of the seven predictors I was working with. For example, the variable x6 is obtained as “(latitude)*(longitude)^2”.

It is important to note that the MSE is a bit higher than the value obtained for the full third order polynomial (it was 0.549144), but now the model includes fewer variables and it’s easier to work with.

Being the order of the predictor 1=latitude, 2=longitude, 3=minimum nights, 4=number of reviews, 5=reviews per month, 6=calculated host listing count and 7=availability 365, the variables contained in the final model are those listed below. The numbers corresponding the position of each predictor tell whether it is included in the new variable or not. Additionally, if the number is greater than 1, it corresponds to the degree at which that variable is taken.

```
##  [1] "1.0.0.0.0.0.0" "2.0.0.0.0.0.0" "3.0.0.0.0.0.0" "0.1.0.0.0.0.0
"
##  [5] "2.1.0.0.0.0.0" "1.2.0.0.0.0.0" "0.3.0.0.0.0.0" "1.0.1.0.0.0.0
"
##  [9] "2.0.1.0.0.0.0" "0.1.1.0.0.0.0" "1.1.1.0.0.0.0" "0.2.1.0.0.0.0
"
## [13] "0.0.2.0.0.0.0" "0.1.2.0.0.0.0" "0.0.3.0.0.0.0" "0.0.0.1.0.0.0
"
## [17] "2.0.0.1.0.0.0" "0.1.0.1.0.0.0" "1.1.0.1.0.0.0" "0.2.0.1.0.0.0
"
## [21] "1.0.1.1.0.0.0" "0.1.1.1.0.0.0" "0.0.0.2.0.0.0" "0.1.0.2.0.0.0
"
## [25] "0.0.0.3.0.0.0" "0.0.1.0.1.0.0" "1.0.1.0.1.0.0" "0.0.2.0.1.0.0
"
## [29] "0.0.0.1.1.0.0" "1.0.0.1.1.0.0" "0.1.0.1.1.0.0" "0.0.0.2.1.0.0
"
## [33] "0.0.0.0.2.0.0" "0.1.0.0.2.0.0" "0.0.1.0.2.0.0" "0.0.0.1.2.0.0
"
## [37] "0.0.0.0.0.1.0" "2.0.0.0.0.1.0" "0.1.0.0.0.1.0" "1.1.0.0.0.1.0
"
## [41] "0.2.0.0.0.1.0" "0.0.1.0.0.1.0" "0.0.2.0.0.1.0" "0.0.0.1.0.1.0
"
## [45] "1.0.0.1.0.1.0" "0.0.0.0.1.1.0" "1.0.0.0.1.1.0" "0.1.0.0.1.1.0
"
```

```
## [49] "0.0.1.0.1.1.0" "0.0.0.1.1.1.0" "0.0.0.0.0.2.0" "1.0.0.0.0.2.0
"
## [53] "0.1.0.0.0.2.0" "0.0.1.0.0.2.0" "0.0.0.0.1.2.0" "0.0.0.0.0.3.0
"
## [57] "0.0.0.0.0.0.1" "1.0.0.0.0.0.1" "2.0.0.0.0.0.1" "0.1.0.0.0.0.1
"
## [61] "1.1.0.0.0.0.1" "0.2.0.0.0.0.1" "0.0.1.0.0.0.1" "0.1.1.0.0.0.1
"
## [65] "0.0.2.0.0.0.1" "0.1.0.1.0.0.1" "0.0.0.2.0.0.1" "1.0.0.0.1.0.1
"
## [69] "0.1.0.0.1.0.1" "0.0.1.0.1.0.1" "0.0.0.0.2.0.1" "0.0.0.0.0.1.1
"
## [73] "0.1.0.0.0.1.1" "0.0.1.0.0.1.1" "0.0.0.0.0.2.1" "0.0.0.0.0.0.2
"
## [77] "0.1.0.0.0.0.2" "0.0.0.1.0.0.2" "0.0.0.0.1.0.2"
```

We can check once again how the results change when using the penalization methods:

```
var <- c("x1", "x2" ,"x3" ,"x4" ,"x6" ,"x8" ,"x9", "x11" ,"x12" ,"x13"
, "x14", "x15" ,"x16", "x18" ,"x19" ,"x20" ,"x22" ,"x23","x24", "x25",
"x27", "x28","x30", "x32" ,"x34" ,"x41" ,"x42" ,"x44","x45", "x46" ,"x
47", "x49" ,"x50", "x52" ,"x53" ,"x54" ,"x56" ,"x58","x59", "x60" ,"x6
1", "x62" ,"x65", "x66" ,"x67" ,"x71" ,"x72" ,"x73","x74", "x75", "x77
", "x78" ,"x79", "x80" ,"x82" ,"x83" ,"x84" ,"x85","x86", "x87", "x88"
, "x89" ,"x90", "x92", "x93" ,"x96" ,"x98" ,"x100" ,  "x101" ,"x102" ,
"x104" ,"x105", "x107" ,"x108" ,"x111" ,"x112", "x114","x116" ,"x117")

finalridge <- cv.glmnet(as.matrix(trainx3[,var]), trainy3, alpha = 0,
nfolds = 10)
final_ridge_opt_lambda <- finalridge$lambda.min
ridge_pred <- predict(finalridge, s = final_ridge_opt_lambda, newx = a
s.matrix(testx3[,var]))
msefinridge <- mean((ridge_pred - testy3)^2)

finallasso <- cv.glmnet(as.matrix(trainx3[,var]), trainy3, alpha = 1,
nfolds = 10)
final_lasso_opt_lambda <- finallasso$lambda.min
lasso_pred <- predict(finallasso, s = final_lasso_opt_lambda, newx = a
s.matrix(testx3[,var]))
msefinlasso <- mean((lasso_pred - testy3)^2)
```

```
mseefin; msefinridge; msefinlasso
```

```
## [1] 0.5493735
```

```
## [1] 0.5778579
```

```
## [1] 0.5773003
```

In this case Lasso works slightly better than Ridge, but they both give a MSE that is higher than the one from OLS. This probably happens because the variables I included in this last model are the best I could use according to the backward selection and, thus, penalizing their use doesn't improve the results.

What has been shown is that penalizing the use of many variables doesn't always lead to a better model, but it is useful to check different regression methods in order to understand how the variables relate to the response and how they can be used to explain and predict it.

Here finished my project,

Thank you for your time,

Simone.