Lab 6

Jonathan Eng

11:59PM March 21, 2020

Load the Boston Housing data and create the vector `y`, design matrix `X` and let `n` and `p_plus_one` be the number of rows and columns.

```
y = MASS::Boston$medv
X = as.matrix(cbind(1, MASS::Boston[, 1 : 13]))
n = nrow(X)
p_plus_one = ncol(X)
```

Create a new matrix `Xjunk` by adding random columns to `X` to make the number of columns and rows the same.

```
Xjunk = X
for(j in (p_plus_one + 1) : n){
  Xjunk = cbind(Xjunk, rnorm(n))

}
dim(Xjunk)
```

```
## [1] 506 506
```

Test that the projection matrix onto $colsp[Xjunk]$ is the same as I_n:

```
pacman::p_load(testthat)
I_n = diag(n)
expect_equal(c(Xjunk %*% solve(t(Xjunk) %*% Xjunk) %*% t(Xjunk)), c(I_n))
```

Write a function spec'd as follows:

```
#' Orthogonal Projection
#'
#' Projects vector a onto v.
#'
#' @param a   the vector to project
#' @param v   the vector projected onto
#'
#' @returns   a list of two vectors, the orthogonal projection parallel to v named a_parallel,
#'            and the orthogonal error orthogonal to v called a_perpendicular
orthogonal_projection = function(a, v){
  a_parallel = (v %*% t(v) / sum(v^2)) %*% a
  a_perpendicular = a - a_parallel
  list(a_parallel = a_parallel, a_perpendicular = a_perpendicular)
}
```

Provide predictions for each of these computations and then run them to make sure you're correct.

```r
orthogonal_projection(c(1,2,3,4), c(1,2,3,4))
```

```
## $a_parallel
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
##
## $a_perpendicular
##      [,1]
## [1,]    0
## [2,]    0
## [3,]    0
## [4,]    0
```

```r
#prediction:
orthogonal_projection(c(1, 2, 3, 4), c(0, 2, 0, -1))
```

```
## $a_parallel
##      [,1]
## [1,]    0
## [2,]    0
## [3,]    0
## [4,]    0
##
## $a_perpendicular
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
```

```r
#prediction:
result = orthogonal_projection(c(2, 6, 7, 3), c(1, 3, 5, 7))
t(result$a_parallel) %*% result$a_perpendicular
```

```
##              [,1]
## [1,] -3.552714e-15
```

```r
#prediction:
result$a_parallel + result$a_perpendicular
```

```
##      [,1]
## [1,]    2
## [2,]    6
## [3,]    7
## [4,]    3
```

```r
#prediction:
result$a_parallel / c(1, 3, 5 ,7)
```

```
##           [,1]
## [1,] 0.9047619
## [2,] 0.9047619
## [3,] 0.9047619
## [4,] 0.9047619
```

```r
#prediction:
```

Try to orthogonally project onto the column space of X by projecting y on each vector of X individually and adding up the projections. You can use the function `orthogonal_projection`.

```r
sumProj <- 0
for (j in 1:p_plus_one){
  sumProj = sumProj + orthogonal_projection(y, X[ , j])$a_parallel
}
```

How much double counting occurred? Measure the magnitude relative to the true LS orthogonal projection.

```r
yhat = lm(y ~ X)$fitted.values
sqrt(sum(sumProj^2)) / sqrt(sum(yhat^2))
```

```
## [1] 8.997118
```

Convert X into V where V has the same column space as X but has orthogonal columns. You can use the function `orthogonal_projection`. This is the Gram-Schmidt orthogonalization algorithm.

```r
V = matrix(NA, nrow = nrow(X), ncol = ncol(X))
V[ , 1] <- X[ , 1]
for (j in 2:p_plus_one){
  V[ , j] <- X[ , j]
  for (k in 1:(j - 1)){
    V[ , j] <- V[ , j] - orthogonal_projection(X[ , j], V[ , k])$a_parallel
  }
}
t(V[ , 1]) %*% V[ , 2]
```

```
##               [,1]
## [1,] -1.544542e-12
```

Convert V into Q whose columns are the same except normalized

```r
Q = matrix(NA, nrow = nrow(X), ncol = ncol(X))
for(j in 1:p_plus_one){
  Q[ , j] = V[ , j] / sqrt(sum(V[ , j]^2))
}
```

Verify $Q^T Q$ is I_{p+1} i.e. Q is an orthonormal matrix.

```
expect_equal(t(Q) %*% Q, diag(p_plus_one))
```

Project y onto $colsp[Q]$ and verify it is the same as the OLS fit.

```
expect_equal(c(unname(Q %*% t(Q) %*% y)), unname(yhat))
```

Project Y onto the columns of Q one by one and verify it sums to be the projection onto the whole space.

```
sumProj <- 0
for (j in 1:p_plus_one){
  sumProj = sumProj + orthogonal_projection(y, Q[ , j])$a_parallel
}
```

Verify the sum of projections is $\hat{y}$

```
expect_equal(c(sumProj), unname(yhat))
```

Split the Boston Housing Data into a training set and a test set where the training set is 80% of the observations. Do so at random.

```
prop_train = 0.8
n_train = round(prop_train * n)
index_train = sample(1:n, n_train, replace = FALSE)
index_test = setdiff(1:n, index_train)

expect_equal(sort(c(index_test, index_train)), 1:n)

X_train = X[index_train, ]
y_train = y[index_train]

X_test = X[index_test, ]
y_test = y[index_test]
```

Find the s_e in sample and out of sample. Which one is greater? Note: we are now using s_e and not RMSE since RMSE has the $-(p+1)$ in the denominator which makes comparison more difficult when the n's are different.

```
get_insample_error = function(model){
  insample_error = sd(insample_model$residuals)

  insample_error
}

get_outsample_error = function(x_test, y_test, model){
  yhat = predict(model, data.frame(x_test))
  residuals = y_test - yhat
  outsample_error = sd(residuals)

  outsample_error
}
```

```r
insample_model = lm(y_train ~ X_train)

insample_error = get_insample_error(insample_model)
outsample_error = get_outsample_error(X_test, y_test,insample_model)

insample_error
```

```
## [1] 4.837134
```

```r
outsample_error
```

```
## [1] 11.32407
```

The out of sample error is greater than that of the in sample error

Do these two exercises 1,000 times and find the average difference between s_e and ooss_e. This is just `sd(e)` the standard deviation of the residuals.

```r
itter = 1000
for(i in 1:itter){
  insample_model = lm(y_train ~ X_train)
  insample_error = get_insample_error(insample_model)

  outsample_error = get_outsample_error(X_test, y_test,insample_model)
}

average_difference = (insample_error - outsample_error) / itter
average_difference
```

```
## [1] -0.006486938
```

Using `Xjunk` from above, divide the data into training and testing sets. Fit the model in-sample and calculate s_e in-sample by varying the number of columns used beginning with the first column. Keep the s_e values in the variable `s_es` which has length n. Show that it reaches 0 at n i.e. the model overfits.

```r
s_es = array(data = NA, dim = nrow(Xjunk))

prop_train = 0.8
n_train = round(prop_train * n)
index_train = sample(1:n, n_train, replace = FALSE)
index_test = setdiff(1:n, index_train)

X_train = Xjunk[index_train, ]
y_train = y[index_train]

X_test = Xjunk[index_test, ]
y_test = y[index_test]

for(i in 1:nrow(s_es)){
  insample_model = lm(y_train ~ X_train[ ,1:i])
  s_es[i] = get_insample_error(insample_model)
}

s_es
```

```
##   [1] 9.1255385 8.4508546 8.0935061 7.7292493 7.5616944 7.5559710 6.0054328
##   [8] 5.9978376 5.7156624 5.7069802 5.6615411 5.4544506 5.2752716 4.7283576
##  [15] 4.7125293 4.7117229 4.7117210 4.7117174 4.6990811 4.6965993 4.6965864
##  [22] 4.6956983 4.6822806 4.6800008 4.6729630 4.6729454 4.6725471 4.6710132
##  [29] 4.6659651 4.6658770 4.6649068 4.6241434 4.6226679 4.6157453 4.6149118
##  [36] 4.6149114 4.6066940 4.6034041 4.6004169 4.5880259 4.5619985 4.5606442
##  [43] 4.5583044 4.5415948 4.5410384 4.5340532 4.5333520 4.5333276 4.5327271
##  [50] 4.5183157 4.5147641 4.5138049 4.5136740 4.5094561 4.5032755 4.4896992
##  [57] 4.4797923 4.4673971 4.4453935 4.4321749 4.4301510 4.4300910 4.4298587
##  [64] 4.4267548 4.4245333 4.4014098 4.3796129 4.3617731 4.3589193 4.3524031
##  [71] 4.3523780 4.3369127 4.3366745 4.3160256 4.3144037 4.3140509 4.3089626
##  [78] 4.3067466 4.3034246 4.2851561 4.2746203 4.2675475 4.2387625 4.2372498
##  [85] 4.2355792 4.2355171 4.2354237 4.2346886 4.2141244 4.1826432 4.1722261
##  [92] 4.1445784 4.1336791 4.1270579 4.1258567 4.1245815 4.1105933 4.1097875
##  [99] 4.1087676 4.0939930 4.0903150 4.0902473 4.0771397 4.0744564 4.0590191
## [106] 4.0584776 4.0583050 4.0516858 4.0515944 4.0514129 4.0298762 4.0290292
## [113] 4.0288898 4.0255923 4.0252999 4.0250357 4.0075054 3.9851467 3.9851283
## [120] 3.8900698 3.8872544 3.8857469 3.8852831 3.8809873 3.8791314 3.8678361
## [127] 3.8387361 3.8385041 3.8345491 3.8313095 3.8209083 3.7983906 3.7973171
## [134] 3.7870370 3.7870204 3.7855658 3.7803677 3.7768063 3.7758950 3.7689119
## [141] 3.7670501 3.7444488 3.7377600 3.7344620 3.7339952 3.7179769 3.7001757
## [148] 3.6983012 3.6934015 3.6933991 3.6933951 3.6910593 3.6903100 3.6671028
## [155] 3.6670216 3.6667852 3.6450522 3.6375789 3.6160363 3.6143389 3.6097019
## [162] 3.5878661 3.5745216 3.5720450 3.5642788 3.5640916 3.5488574 3.5450835
## [169] 3.5447284 3.5424400 3.5293261 3.5289533 3.5280751 3.5254438 3.5242669
## [176] 3.5194654 3.5185366 3.5179650 3.5179038 3.5074514 3.4970221 3.4942420
## [183] 3.4927782 3.4893488 3.4834780 3.4712321 3.4701833 3.4490859 3.4466380
## [190] 3.4081129 3.3885496 3.3883600 3.3801995 3.3755237 3.3641363 3.3635458
## [197] 3.3596380 3.3554414 3.3283157 3.3273795 3.3110659 3.2831653 3.2831642
## [204] 3.2755973 3.2520864 3.2355717 3.2279730 3.2016092 3.1991486 3.1914318
## [211] 3.1699594 3.1698894 3.1602960 3.1419279 3.1376595 3.1333133 3.1255400
## [218] 3.1221831 3.1214182 3.0922265 3.0811890 3.0770997 3.0760988 3.0737507
## [225] 3.0488634 3.0334603 3.0253344 3.0180205 3.0144943 3.0131539 3.0121997
## [232] 2.9966750 2.9960622 2.9740256 2.9662756 2.9454902 2.9423121 2.9091820
## [239] 2.9073043 2.8953338 2.8931713 2.8704658 2.8689998 2.8645670 2.8566409
## [246] 2.8498410 2.8481605 2.8473175 2.8419152 2.8296132 2.8169466 2.8083703
## [253] 2.8079285 2.8012084 2.7884050 2.7847101 2.7774894 2.7569949 2.7565159
## [260] 2.7554003 2.7385672 2.7370465 2.7237562 2.7035839 2.6967368 2.6592697
## [267] 2.6564497 2.6553944 2.6548931 2.6535093 2.6534281 2.6517163 2.6256892
## [274] 2.6070804 2.6007008 2.5992964 2.5870634 2.5868146 2.5743206 2.5448901
## [281] 2.5339268 2.5288454 2.5157712 2.5134558 2.5132505 2.5078482 2.4769528
## [288] 2.4766357 2.4680223 2.4594966 2.4560819 2.4414579 2.4072861 2.4058792
## [295] 2.3650475 2.3644745 2.3557643 2.3540289 2.3315700 2.3087324 2.3024992
## [302] 2.2948733 2.2810123 2.2739794 2.2730627 2.2730087 2.2481689 2.2389165
## [309] 2.2174165 2.1995589 2.1754080 2.1693286 2.1556891 2.1542479 2.1481930
## [316] 2.1072461 2.0719751 2.0719237 2.0693999 2.0669462 2.0402365 2.0246686
## [323] 2.0131516 2.0043192 2.0027199 1.9580871 1.9385865 1.9384771 1.9369923
## [330] 1.9247532 1.9184523 1.8738437 1.8435134 1.8161968 1.7419728 1.7417595
## [337] 1.7416039 1.7404022 1.7395068 1.7385179 1.7066138 1.6965829 1.6953154
## [344] 1.6953154 1.6757738 1.6727334 1.6721477 1.6574563 1.6457714 1.6297069
## [351] 1.6296268 1.6296263 1.6290444 1.6290110 1.6202260 1.6151037 1.5517247
## [358] 1.5413527 1.5404832 1.5305703 1.5120911 1.4633261 1.4589468 1.4178993
## [365] 1.4077278 1.3705757 1.3499307 1.3484032 1.3466750 1.3441900 1.3235259
## [372] 1.3005948 1.2498395 1.2497982 1.2466946 1.2317919 1.2283970 1.2103027
```

```
## [379] 1.2102465 1.1856985 1.1814706 1.1812747 1.1748728 1.0133270 0.9972477
## [386] 0.9963037 0.8940897 0.8518287 0.7808789 0.7498689 0.7446609 0.7228450
## [393] 0.6883018 0.6695840 0.6397545 0.6190871 0.6172704 0.6170168 0.6062635
## [400] 0.5589897 0.5307171 0.2808234 0.2524483 0.2430335 0.0000000 0.0000000
## [407] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## [414] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## [421] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## [428] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## [435] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## [442] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## [449] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## [456] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## [463] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## [470] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## [477] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## [484] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## [491] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## [498] 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000 0.0000000
## [505] 0.0000000 0.0000000
```

Do the same thing but now calculate oos_{s_e}. Does this go to zero? What is the index corresponding to the best model?

```r
oos_e = array(data = NA, dim = nrow(Xjunk))

for(i in 1:nrow(oos_e)){
  insample_model = lm(y_train ~ X_train[ ,1:i])
  oos_e[i] = get_outsample_error(X_test, y_test, insample_model)

}

temp = oos_e[1]
j = 1

for(i in 2:nrow(oos_e)){
  if(oos_e[i] < temp){
    temp = oos_e[i]
    j = i
  }
}

j #Index of best model (smallest standard error)
```

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

```r
oos_e
```

```
##   [1]  9.417539 10.449658 10.678564 11.117757 11.259838 11.257233 12.168684
##   [8] 12.158385 12.260246 12.264890 12.303636 12.486753 12.584932 12.757686
##  [15] 12.760930 12.761155 12.761021 12.761081 12.749118 12.755395 12.755060
##  [22] 12.752079 12.748707 12.754306 12.752928 12.754223 12.753578 12.756589
##  [29] 12.754566 12.756752 12.752018 12.752102 12.749809 12.735593 12.735522
##  [36] 12.735707 12.752559 12.752106 12.749301 12.768383 12.780441 12.779792
```

```
##  [43] 12.781849 12.777791 12.778963 12.798559 12.801832 12.802261 12.802988
##  [50] 12.800576 12.804578 12.805811 12.804548 12.799196 12.796694 12.799596
##  [57] 12.804450 12.822835 12.825905 12.825155 12.825347 12.824742 12.824480
##  [64] 12.821293 12.822609 12.814094 12.836487 12.832449 12.840940 12.830862
##  [71] 12.830824 12.830509 12.831785 12.836081 12.834573 12.834197 12.838055
##  [78] 12.837811 12.838264 12.866478 12.886569 12.890921 12.869898 12.875481
##  [85] 12.882138 12.882968 12.882475 12.879918 12.856520 12.860227 12.856536
##  [92] 12.857117 12.858184 12.863638 12.866401 12.861143 12.868814 12.873153
##  [99] 12.870099 12.874107 12.866613 12.866799 12.858275 12.861494 12.881816
## [106] 12.879564 12.877954 12.876669 12.876817 12.877789 12.897551 12.896876
## [113] 12.894492 12.906317 12.908386 12.910667 12.923557 12.908298 12.907723
## [120] 12.910370 12.908274 12.903430 12.908851 12.916698 12.903765 12.903943
## [127] 12.906485 12.907481 12.894900 12.894652 12.919568 12.902187 12.905129
## [134] 12.910634 12.910399 12.913558 12.917007 12.920336 12.920222 12.924413
## [141] 12.925057 12.929102 12.937330 12.939400 12.945091 12.958782 12.970881
## [148] 12.972905 12.973113 12.973136 12.973128 12.970232 12.971441 12.978668
## [155] 12.978671 12.978799 13.002879 13.015452 13.020907 13.021697 13.012635
## [162] 13.017932 13.029917 13.037820 13.038454 13.040799 13.056364 13.064337
## [169] 13.065409 13.066269 13.060425 13.061582 13.060607 13.060360 13.064535
## [176] 13.065454 13.062968 13.063296 13.064028 13.056463 13.060555 13.066703
## [183] 13.073686 13.081624 13.085109 13.095401 13.094684 13.099043 13.100965
## [190] 13.125862 13.140195 13.138655 13.133389 13.134278 13.144274 13.146204
## [197] 13.148191 13.149428 13.162893 13.162510 13.180430 13.204515 13.204330
## [204] 13.208720 13.224537 13.233978 13.229291 13.238431 13.234548 13.230211
## [211] 13.226274 13.227477 13.234454 13.242503 13.235679 13.243037 13.238719
## [218] 13.239499 13.234541 13.232453 13.232020 13.233449 13.233552 13.230159
## [225] 13.222931 13.232826 13.231067 13.251232 13.245162 13.246061 13.242850
## [232] 13.250991 13.254410 13.263317 13.259161 13.255284 13.255565 13.262556
## [239] 13.262352 13.250315 13.248700 13.271474 13.276070 13.264615 13.268463
## [246] 13.258131 13.256679 13.256848 13.253875 13.266899 13.279233 13.279699
## [253] 13.277431 13.283231 13.279248 13.270331 13.268501 13.271020 13.267647
## [260] 13.266580 13.288805 13.289631 13.299866 13.320920 13.342595 13.351086
## [267] 13.347968 13.349937 13.349503 13.347864 13.347270 13.349718 13.374082
## [274] 13.374244 13.375444 13.372015 13.381108 13.383020 13.381859 13.372980
## [281] 13.367487 13.370530 13.361150 13.366717 13.364409 13.366399 13.362507
## [288] 13.365809 13.364354 13.376700 13.381599 13.388885 13.390927 13.386585
## [295] 13.394962 13.396347 13.399839 13.406998 13.425833 13.431107 13.431457
## [302] 13.435994 13.445729 13.452757 13.449268 13.449093 13.455438 13.445227
## [309] 13.446910 13.451020 13.452203 13.461709 13.479374 13.483370 13.480759
## [316] 13.505402 13.495164 13.495861 13.496823 13.496239 13.513394 13.510341
## [323] 13.512155 13.509061 13.506699 13.518433 13.522155 13.522017 13.526835
## [330] 13.516850 13.521017 13.532465 13.537235 13.550397 13.567904 13.569667
## [337] 13.569521 13.571278 13.571619 13.571585 13.595032 13.598366 13.603338
## [344] 13.603339 13.599368 13.594115 13.594940 13.601540 13.595750 13.589021
## [351] 13.588768 13.588734 13.587594 13.587638 13.589048 13.591740 13.605102
## [358] 13.611459 13.611485 13.613900 13.614126 13.610115 13.603524 13.593355
## [365] 13.590440 13.601069 13.611948 13.613042 13.614466 13.613061 13.611275
## [372] 13.597343 13.576099 13.576114 13.577346 13.579475 13.584940 13.579328
## [379] 13.579300 13.586538 13.585951 13.585477 13.579302 13.580449 13.579259
## [386] 13.578597 13.589958 13.604881 13.604964 13.610200 13.609968 13.613812
## [393] 13.609353 13.604928 13.603263 13.616757 13.614859 13.615559 13.613867
## [400] 13.632095 13.631418 13.617106 13.620035 13.622378 13.627252 13.627252
## [407] 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252
## [414] 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252
```

```
## [421] 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252
## [428] 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252
## [435] 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252
## [442] 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252
## [449] 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252
## [456] 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252
## [463] 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252
## [470] 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252
## [477] 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252
## [484] 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252
## [491] 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252
## [498] 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252 13.627252
## [505] 13.627252 13.627252
```

Index 1 contains the "best model"

Beginning with the Boston Housing Data matrix `X`, pull out the second column, the `crim` feature and call it `x2`. Then, use the `cut` function to bin each of its n values into two bins: the first is all values <= the median of `crim` and the second is all values > median of `crim`. Call it `x2bin`. Use the `table` function to ensure that half of the values are in the first group and half in the second group. This requires reading the documentation for `cut` carefully and using the `quantile` function carefully.

```r
x2 = X[ ,2]

x2bin = cut(x2, breaks = quantile(x2, c(0, .5, 1)), include.lowest = TRUE)
table(x2bin)
```

```
## x2bin
## [0.00632,0.257]      (0.257,89]
##             253             253
```

Now convert the factor variable `x2bin` to two dummies, `X2dummy`, a matrix of $n \times 2$ and verify the rowsums are all 1. They must be 1 because either the value is <= median or > median.

```r
X2dummy = model.matrix( ~ 0 + ., data.frame(x2bin))
table(rowSums(X2dummy))
```

```
##
##   1
## 506
```

Drop the first column of this matrix to arrive at `X2dummyfeatures`.

```r
X2dummyfeatures = X[ ,2:ncol(X)]
```

What you did with `crim`, do for all 13 variables in the Boston housing data, ie create `X2dummyfeatures` for all and then column bind them all together into a massive `Xdummy` matrix. Then run a regression of y on those features and report R^2.

```r
Xdummy = matrix(data = NA, nrow = nrow(X))
Xdummy[ ,1] = 1
```

```r
for(i in 2:ncol(Xdummy)){
  temp_dummyFeatures = X[ ,i:ncol(X)]
  Xdummy = cbind(Xdummy, temp_dummyFeatures)
}

model = lm(y ~ Xdummy)
summary(model)$r.sq
```

```
## [1] 0.7406427
```

Create a new `Xdummy` matrix. This time with two dummies for each variable: (1) between the 33%ile and 66%ile and (2) greater than the 66%ile. Run the regression and report R^2.

```r
#TODO
```

Keep doing this until each variable has 31 dummies for a total of $p = 403 + 1$ variables. Report all R^2;s. Why is it increasing and why is the last one so high?

```r
#TODO
```

Repeat this exercise with a 20% test set held out. Record in sample s_e's and ooss_e's. Do we see the canonical picture?

```r
#TODO
```

What is the optimal number of bins for each feature? That is what is the optimal complexity model of this set of models?

```r
#TODO
```