STA6703 SML Take-Home Prelim, Fall 2022

64.5/110

Christopher Marais

Helping functions

```
myRound <- function(x, acc=3) {mult = 10^acc; round(x*mult)/mult}
```

Problem 1

Case 1

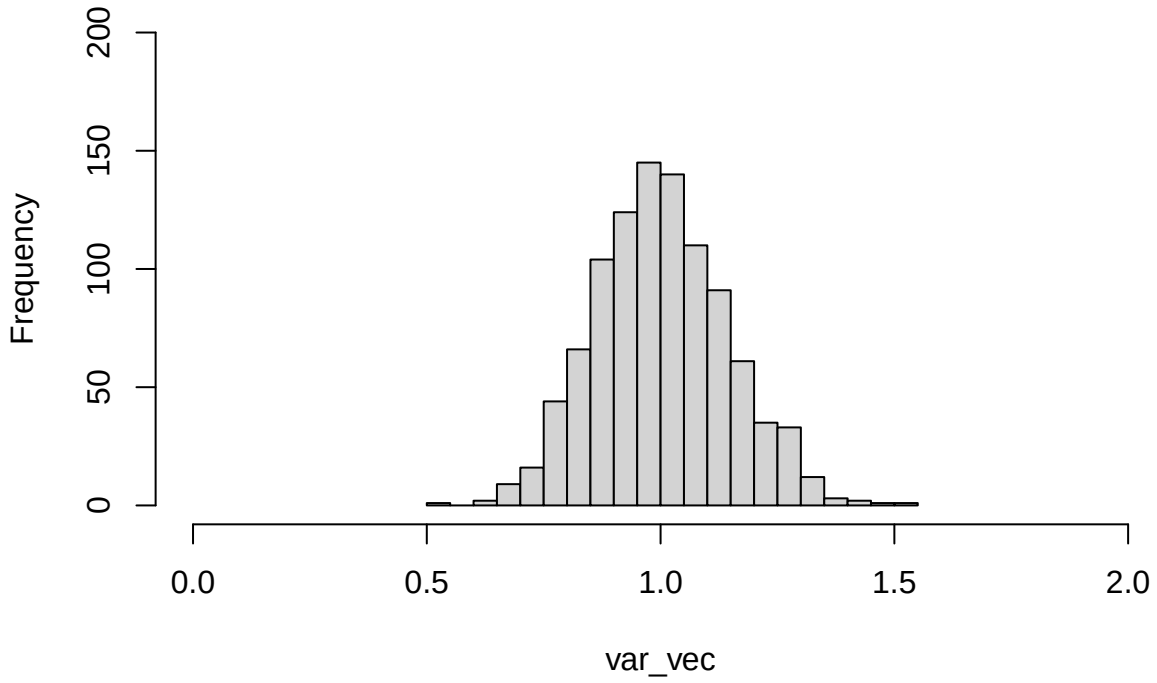
```
# Case 1
n = 100
m=1000
set.seed(0)
origData = rnorm(n) # case 1


# a) Player A
# generate 1000 data sets
set.seed(0)
z = rnorm(n*m)
mat = matrix(z,nrow=m)
#calculate variance for each row
var_vec = apply(mat,1,var)
samp_mean = mean(var_vec)
samp_var = var(var_vec)
print(paste("The sample mean is (Case 1: Player A): ", myRound(samp_mean)))
```

```
## [1] "The sample mean is (Case 1: Player A):  1.002"
```

```
print(paste("The sample variance is (Case 1: Player A): ", myRound(samp_var)))
```

```
## [1] "The sample variance is (Case 1: Player A):  0.02"
```

```
hist(var_vec,
     ylim=c(0,200),
     breaks=20,
     xlim=c(0,2),
     main="Sample variances (Case 1: Player A)")
```



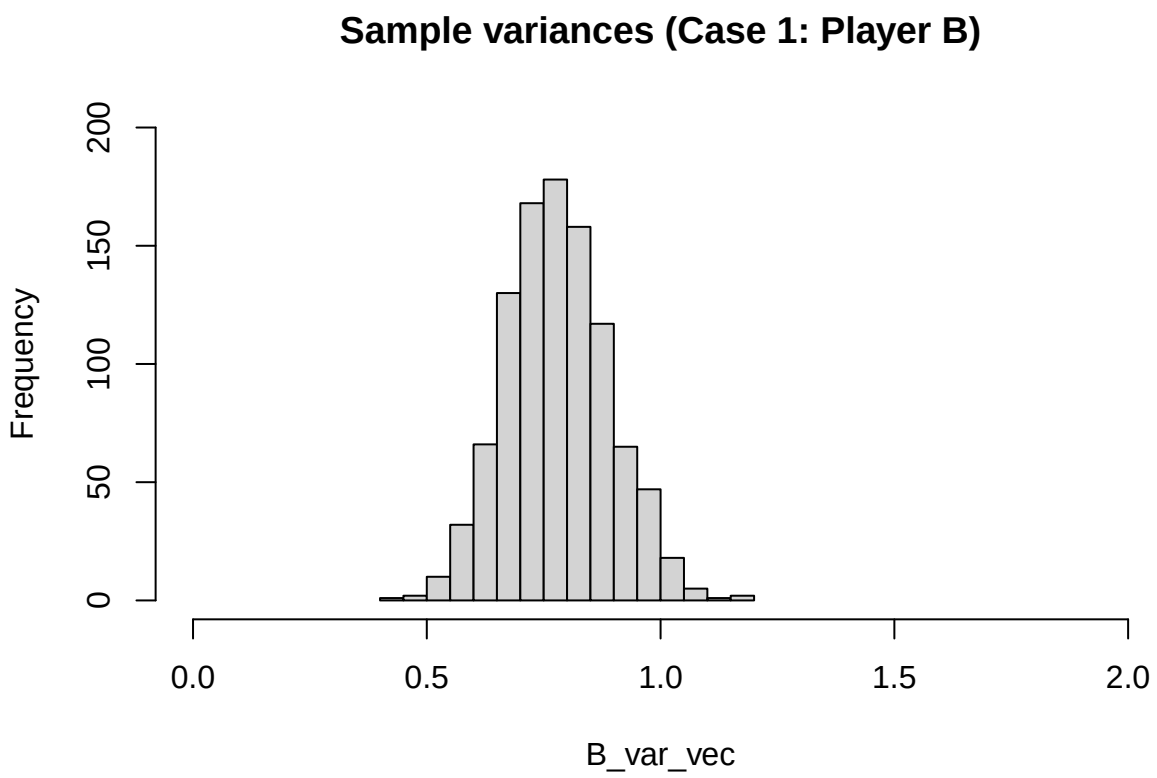

```
# b) Player B
# Estimate mean and variance from the sample data
B_var = var(origData)
B_mean = mean(origData)
# simulate new data with
set.seed(0)
B_mat = matrix(rnorm(n*m, mean = B_mean, sd = sqrt(B_var)),nrow=m)
B_var_vec = apply(B_mat,1,var)
B_samp_mean = mean(B_var_vec)
B_samp_var = var(B_var_vec)
print(paste("The sample mean is (Case 1: Player B): ", myRound(B_samp_mean)))
```

```
## [1] "The sample mean is (Case 1: Player B):  0.781"
```

```
print(paste("The sample variance is (Case 1: Player B): ", myRound(B_samp_var)))
```

```
## [1] "The sample variance is (Case 1: Player B):  0.012"
```

```
hist(B_var_vec,
     ylim=c(0,200),
     breaks=20,
     xlim=c(0,2),
     main="Sample variances (Case 1: Player B)")
```



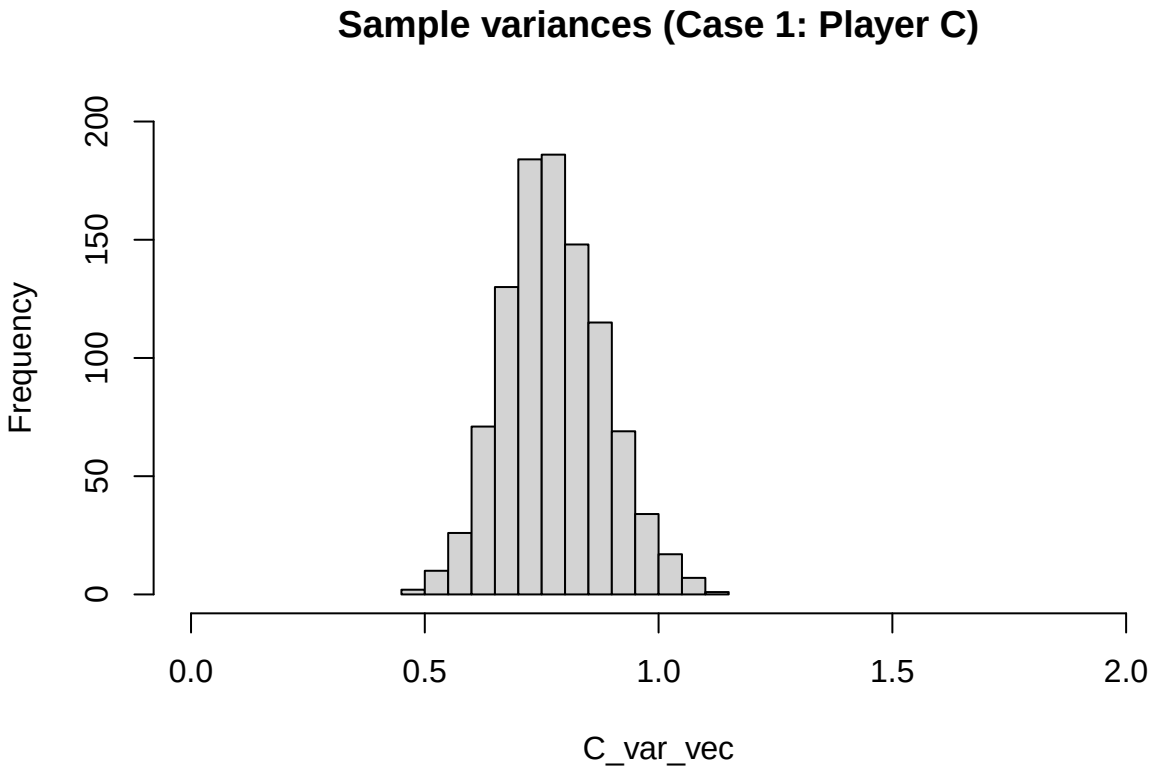

```
# c) Player C
# sample new data
z=sample(x=origData, size=n*m, replace=TRUE)
C_mat = matrix(z,nrow=m)
C_var_vec = apply(C_mat,1,var)
C_samp_mean = mean(C_var_vec)
C_samp_var = var(C_var_vec)
print(paste("The sample mean is (Case 1: Player C): ", myRound(C_samp_mean)))
```

```
## [1] "The sample mean is (Case 1: Player C):  0.776"
```

```
print(paste("The sample variance is (Case 1: Player C): ", myRound(C_samp_var)))
```

```
## [1] "The sample variance is (Case 1: Player C):  0.011"
```

```
hist(C_var_vec,
     ylim=c(0,200),
     breaks=20,
     xlim=c(0,2),
     main="Sample variances (Case 1: Player C)")
```



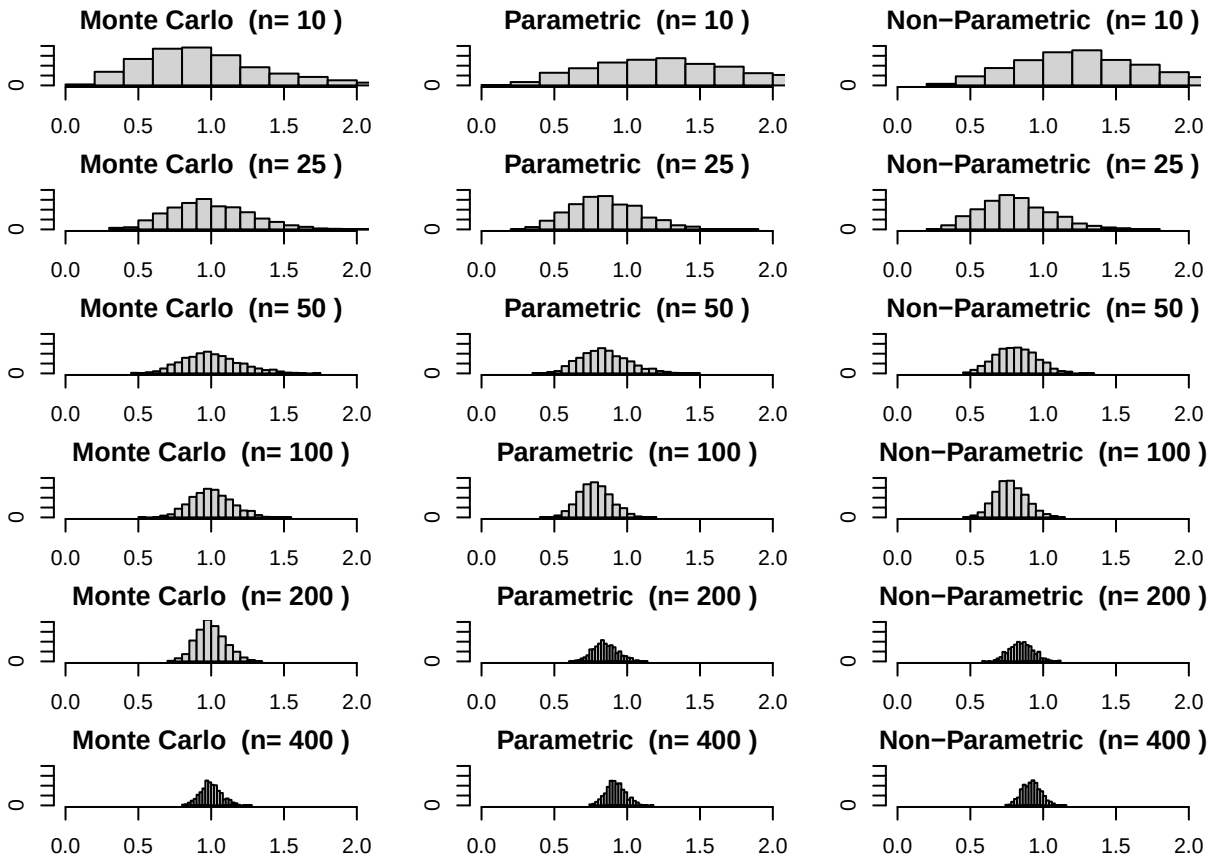
d)

For case 1 we can see that the bootstrapping worked reasonably well. Naturally the Monte Carlo method performed the best and has a estimated mean and variance closest to the true variance and mean. Parametric and non-parametric bootstrapping seem to perform similarly in this case. neither are perfectly accurate, but they produce similar estimates of the mean and variance. In case 1 I would say that both methods had an equally similar bias and variance I would classify both these methods as having a high bias and a low variance. Both methods seem to struggle in estimating the true mean accurately.

```
# e)
n_vec=c(10, 25, 50, 100, 200, 400)
methods_vec = c("Monte Carlo", "Parametric", "Non-Parametric")
m=1000
set.seed(0)
data400 = rnorm(400)

# save results in tables
mean_res_df = data.frame(matrix(ncol=7, nrow=3))
var_res_df = data.frame(matrix(ncol=7, nrow=3))
colnames(mean_res_df) = c("Method", n_vec)
colnames(var_res_df) = c("Method", n_vec)
mean_res_df["Method"] = methods_vec
var_res_df["Method"] = methods_vec

par(mfrow=c(6,3),mar=c(2,2,2,2))
for(n in n_vec){
  mean_res_vec = c()
```

```
var_res_vec = c()

#Monte Carlo
set.seed(0)
z = rnorm(n*m)
mat = matrix(z,nrow=m)
var_vec = apply(mat,1,var)
samp_mean = myRound(mean(var_vec))
samp_var = myRound(var(var_vec))
mean_res_vec = c(mean_res_vec, samp_mean)
var_res_vec = c(var_res_vec, samp_var)
hist(var_vec,
     ylim=c(0,200),
     breaks=20,
     xlim=c(0,2),
     main=paste("Monte Carlo ",
                "(n=",n,")"))

# Parametric
# Estimate mean and variance from the sample data
z = data400[1:n]
B_var = var(z)
B_mean = mean(z)
# simulate new data with
set.seed(0)
B_mat = matrix(rnorm(n*m, mean = B_mean, sd = sqrt(B_var)),nrow=m)
B_var_vec = apply(B_mat,1,var)
B_samp_mean = myRound(mean(B_var_vec))
B_samp_var = myRound(var(B_var_vec))
mean_res_vec = c(mean_res_vec, B_samp_mean)
var_res_vec = c(var_res_vec, B_samp_var)
hist(B_var_vec,
     ylim=c(0,200),
     breaks=20,
     xlim=c(0,2),
     main=paste("Parametric ",
                "(n=",n,")"))

# c) Player C
# sample new data
z=sample(x=data400[1:n], size=n*m, replace=TRUE)
C_mat = matrix(z,nrow=m)
C_var_vec = apply(C_mat,1,var)
C_samp_mean = myRound(mean(C_var_vec))
C_samp_var = myRound(var(C_var_vec))
mean_res_vec = c(mean_res_vec, C_samp_mean)
var_res_vec = c(var_res_vec, C_samp_var)
hist(C_var_vec,
     ylim=c(0,200),
```

```
        breaks=20,
        xlim=c(0,2),
        main=paste("Non-Parametric ",
                   "(n=",n,")"))

  mean_res_df[as.character(n)] = mean_res_vec
  var_res_df[as.character(n)] = var_res_vec
}
```




```
# display tables
knitr::kable(mean_res_df,
             format = "markdown",
             caption = "Mean for different n values (Case 1).")
```

Table 1: Mean for different n values (Case 1).

Method	10	25	50	100	200	400
Monte Carlo	0.987	0.995	1.005	1.002	0.998	0.998
Parametric	1.434	0.856	0.844	0.781	0.851	0.921
Non-Parametric	1.305	0.818	0.815	0.776	0.851	0.920

```
knitr::kable(var_res_df,
             format = "markdown",
             caption = "Variance for different n values (Case 1).")
```

Table 2: Variance for different n values (Case 1).

Method	10	25	50	100	200	400
Monte Carlo	0.230	0.080	0.040	0.020	0.010	0.005
Parametric	0.486	0.059	0.028	0.012	0.007	0.004
Non-Parametric	0.225	0.061	0.020	0.011	0.007	0.004

e) From this experiment we can see that the performance of all the methods increase as `n` increases. Once again for case 1 it seems as though the parametric and non-parametric methods perform almost equally well. They are not as good as the Monte Carlo method, but get closer in performance as the size of `n` increases. We can see that the variance decreases as `n` increases and also that the bias decreases with an increase in `n` for case 1.

Case 2

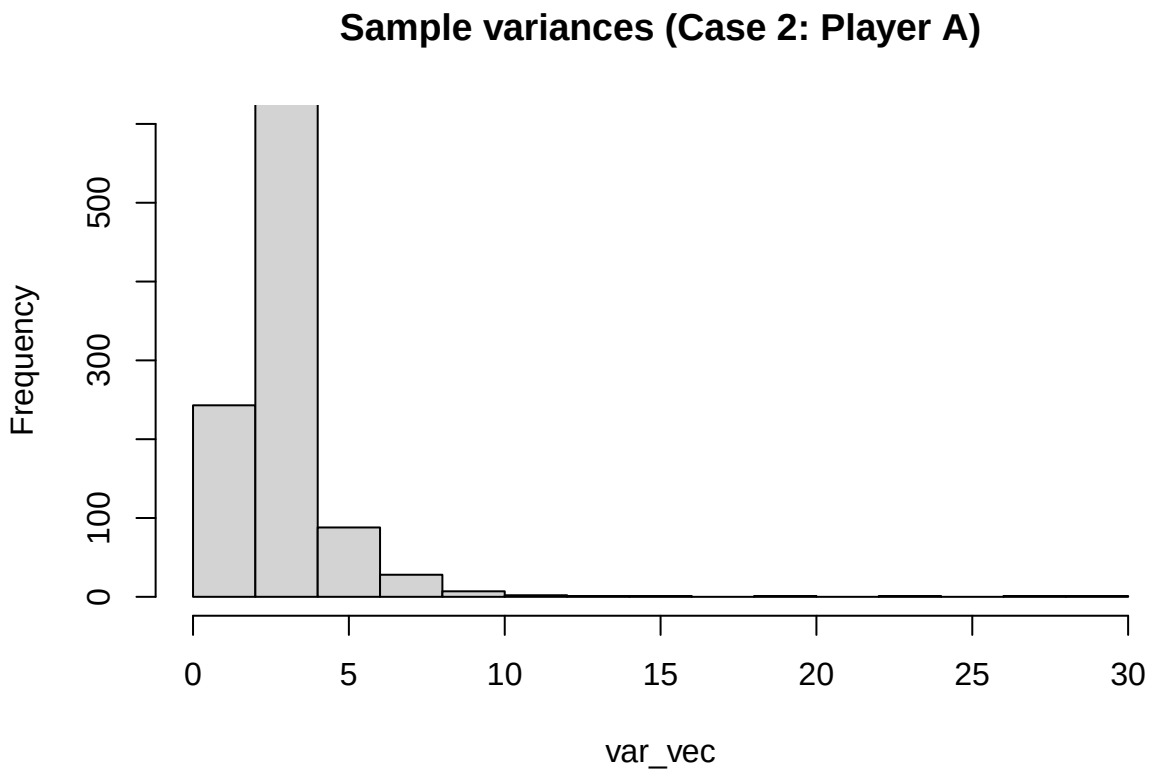
```
# Case 2
n = 100
m=1000
set.seed(0)
origData = rt(n,df=3); # case 2


# a) Player A
# generate 1000 data sets
set.seed(0)
z = rt(n*m,df=3)
mat = matrix(z,nrow=m)
#calculate variance for each row
var_vec = apply(mat,1,var)
samp_mean = mean(var_vec)
samp_var = var(var_vec)
print(paste("The sample mean is (Case 2: Player A): ", myRound(samp_mean)))
```

```
## [1] "The sample mean is (Case 2: Player A):  2.917"
```

```
print(paste("The sample variance is (Case 2: Player A): ", myRound(samp_var)))
```

```
## [1] "The sample variance is (Case 2: Player A):  3.897"
```

```r
hist(var_vec,
     ylim=c(0,600),
     # breaks=20,
     xlim=c(0,30),
     main="Sample variances (Case 2: Player A)")
```



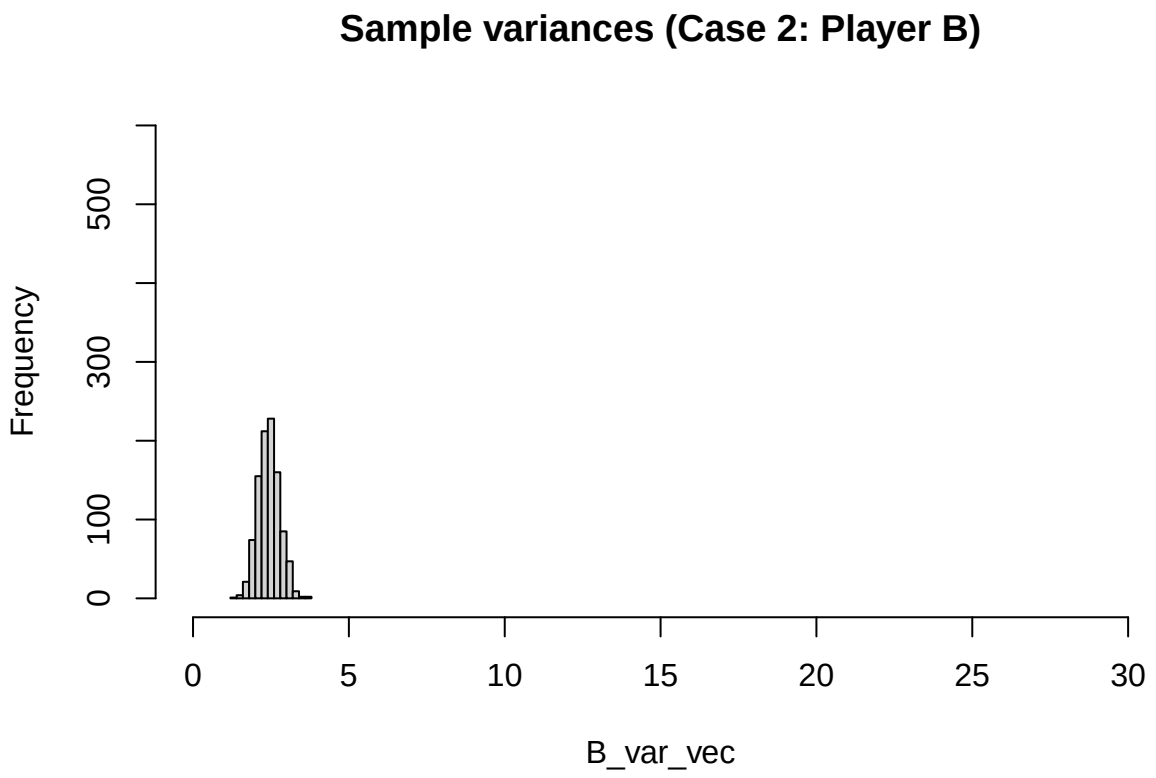

```r
# b) Player B
# Estimate mean and variance from the sample data
B_var = var(origData)
B_mean = mean(origData)
# simulate new data with
set.seed(0)
B_mat = matrix(rnorm(n*m, mean = B_mean, sd = sqrt(B_var)),nrow=m)
B_var_vec = apply(B_mat,1,var)
B_samp_mean = mean(B_var_vec)
B_samp_var = var(B_var_vec)
print(paste("The sample mean is (Case 2: Player B): ", myRound(B_samp_mean)))
```

```
## [1] "The sample mean is (Case 2: Player B):  2.434"
```

```r
print(paste("The sample variance is (Case 2: Player B): ", myRound(B_samp_var)))
```

```
## [1] "The sample variance is (Case 2: Player B):  0.118"
```

```r
hist(B_var_vec,
     ylim=c(0,600),
     # breaks=20,
     xlim=c(0,30),
     main="Sample variances (Case 2: Player B)")
```



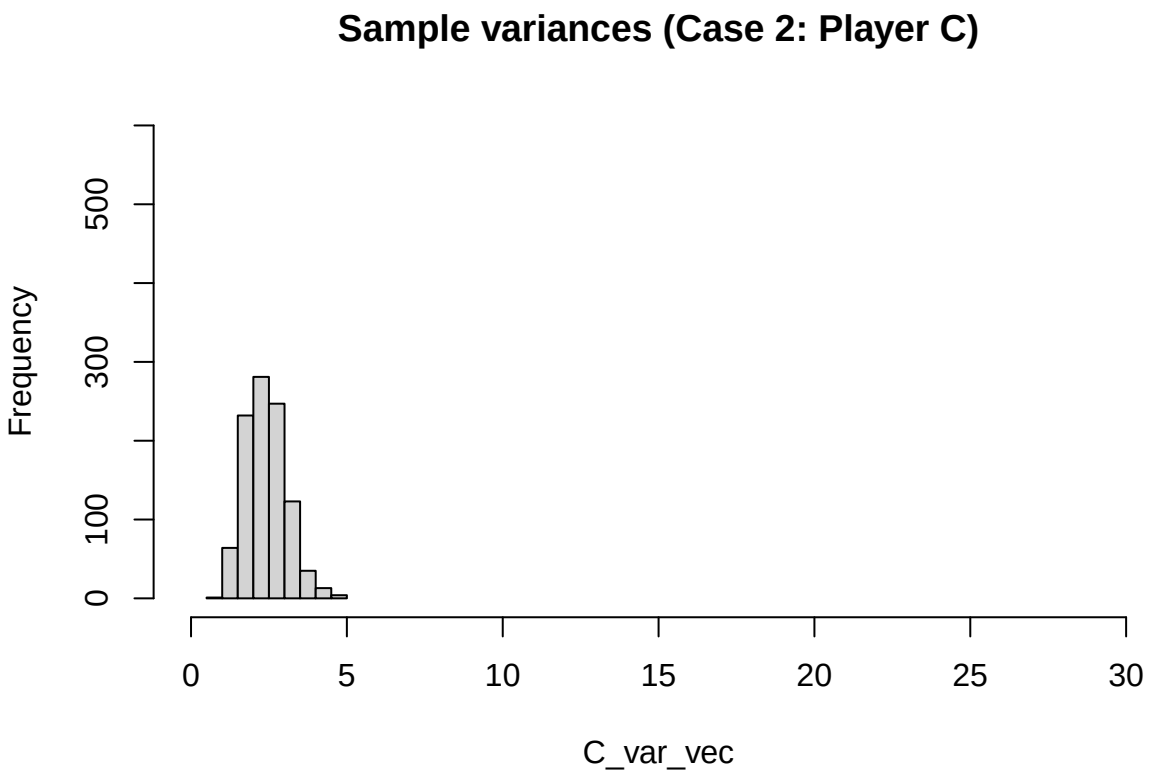

```r
# c) Player C
# sample new data
z=sample(x=origData, size=n*m, replace=TRUE)
C_mat = matrix(z,nrow=m)
C_var_vec = apply(C_mat,1,var)
C_samp_mean = mean(C_var_vec)
C_samp_var = var(C_var_vec)
print(paste("The sample mean is (Case 2: Player C): ", myRound(C_samp_mean)))
```

```
## [1] "The sample mean is (Case 2: Player C):  2.41"
```

```r
print(paste("The sample variance is (Case 2: Player C): ", myRound(C_samp_var)))
```

```
## [1] "The sample variance is (Case 2: Player C):  0.422"
```

```r
hist(C_var_vec,
     ylim=c(0,600),
     # breaks=20,
     xlim=c(0,30),
     main="Sample variances (Case 2: Player C)")
```



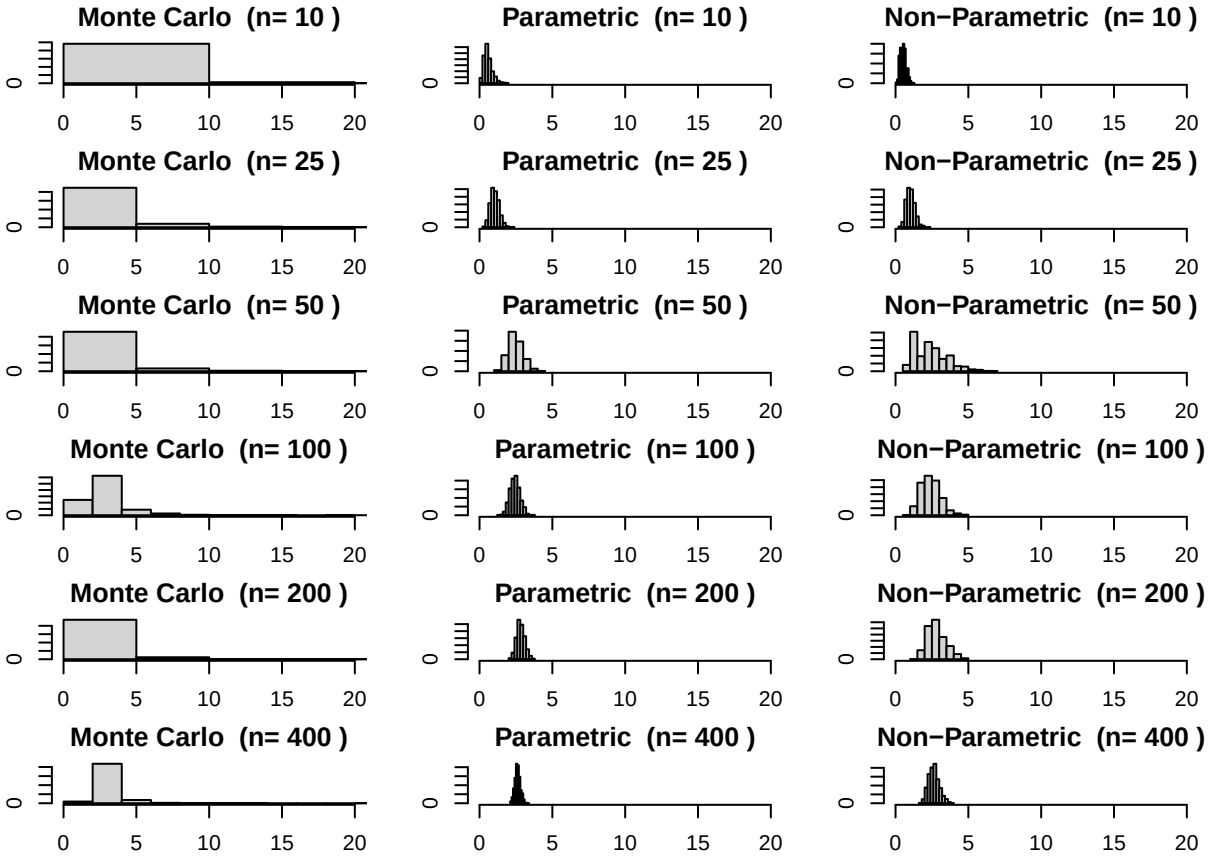
d)

For case 2 we can see something different than what we saw in case 1. Both the Parametric and N-n parametric seem to be accurately estimate the mean of the original distribution, but they are not as accurate with estimating the variance. However, the non-parametric approach seems to be better at estimating the variance than the parametric approach and this is likely because the distribution assumed in the parametric approach is different from the one of the original data. This is a definite drawback of parametric bootstrapping. Assumptions have to be made of the original distribution and this could cause discrepancies. In this case i would say that the parametric and non-parametric approaches have a low bias and a high variance.

```r
# e)
n_vec=c(10, 25, 50, 100, 200, 400)
methods_vec = c("Monte Carlo", "Parametric", "Non-Parametric")
m=1000
set.seed(0)
data400 = rt(400,df=3)

# save results in tables
mean_res_df = data.frame(matrix(ncol=7, nrow=3))
var_res_df = data.frame(matrix(ncol=7, nrow=3))
```

```r
colnames(mean_res_df) = c("Method", n_vec)
colnames(var_res_df) = c("Method", n_vec)
mean_res_df["Method"] = methods_vec
var_res_df["Method"] = methods_vec

par(mfrow=c(6,3),mar=c(2,2,2,2))
for(n in n_vec){
  mean_res_vec = c()
  var_res_vec = c()

  #Monte Carlo
  set.seed(0)
  z = rt(n*m, df=3)
  mat = matrix(z,nrow=m)
  var_vec = apply(mat,1,var)
  samp_mean = myRound(mean(var_vec))
  samp_var = myRound(var(var_vec))
  mean_res_vec = c(mean_res_vec, samp_mean)
  var_res_vec = c(var_res_vec, samp_var)
  hist(var_vec,
       # ylim=c(0,500),
       # breaks=20,
       xlim=c(0,20),
       main=paste("Monte Carlo ",
                  "(n=",n,")"))

  # Parametric
  # Estimate mean and variance from the sample data
  z = data400[1:n]
  B_var = var(z)
  B_mean = mean(z)
  # simulate new data with
  set.seed(0)
  B_mat = matrix(rnorm(n*m, mean = B_mean, sd = sqrt(B_var)),nrow=m)
  B_var_vec = apply(B_mat,1,var)
  B_samp_mean = myRound(mean(B_var_vec))
  B_samp_var = myRound(var(B_var_vec))
  mean_res_vec = c(mean_res_vec, B_samp_mean)
  var_res_vec = c(var_res_vec, B_samp_var)
  hist(B_var_vec,
       # ylim=c(0,500),
       # breaks=20,
       xlim=c(0,20),
       main=paste("Parametric ",
                  "(n=",n,")"))

  # c) Player C
  # sample new data
  z=sample(x=data400[1:n], size=n*m, replace=TRUE)
```

```
  C_mat = matrix(z,nrow=m)
  C_var_vec = apply(C_mat,1,var)
  C_samp_mean = myRound(mean(C_var_vec))
  C_samp_var = myRound(var(C_var_vec))
  mean_res_vec = c(mean_res_vec, C_samp_mean)
  var_res_vec = c(var_res_vec, C_samp_var)
  hist(C_var_vec,
       # ylim=c(0,500),
       # breaks=20,
       xlim=c(0,20),
       main=paste("Non-Parametric ",
                  "(n=",n,")"))

  mean_res_df[as.character(n)] = mean_res_vec
  var_res_df[as.character(n)] = var_res_vec
}
```




```
# display tables
knitr::kable(mean_res_df,
             format = "markdown",
             caption = "Mean for different n values (Case 2).")
```

Table 3: Mean for different n values (Case 2).

Method	10	25	50	100	200	400
Monte Carlo	3.061	3.021	2.940	2.917	3.004	2.976
Parametric	0.583	1.050	2.471	2.434	2.828	2.614
Non-Parametric	0.528	1.031	2.407	2.410	2.816	2.633

```
knitr::kable(var_res_df,
             format = "markdown",
             caption = "Variance for different n values (Case 2).")
```

Table 4: Variance for different n values (Case 2).

Method	10	25	50	100	200	400
Monte Carlo	41.988	12.041	8.706	3.897	3.778	2.057
Parametric	0.080	0.089	0.242	0.118	0.079	0.036
Non-Parametric	0.044	0.079	1.281	0.422	0.409	0.136

e) Similar to Case 1 we see here that when `n` increases we see a better estimate of the mean for the parametric and non-parametric approaches. This is however not true for the variance estimate as it does not improve as much with increasing `n` values. Once again the non-parametric approach performed slightly better than the parametric approach in this case.

Case 3

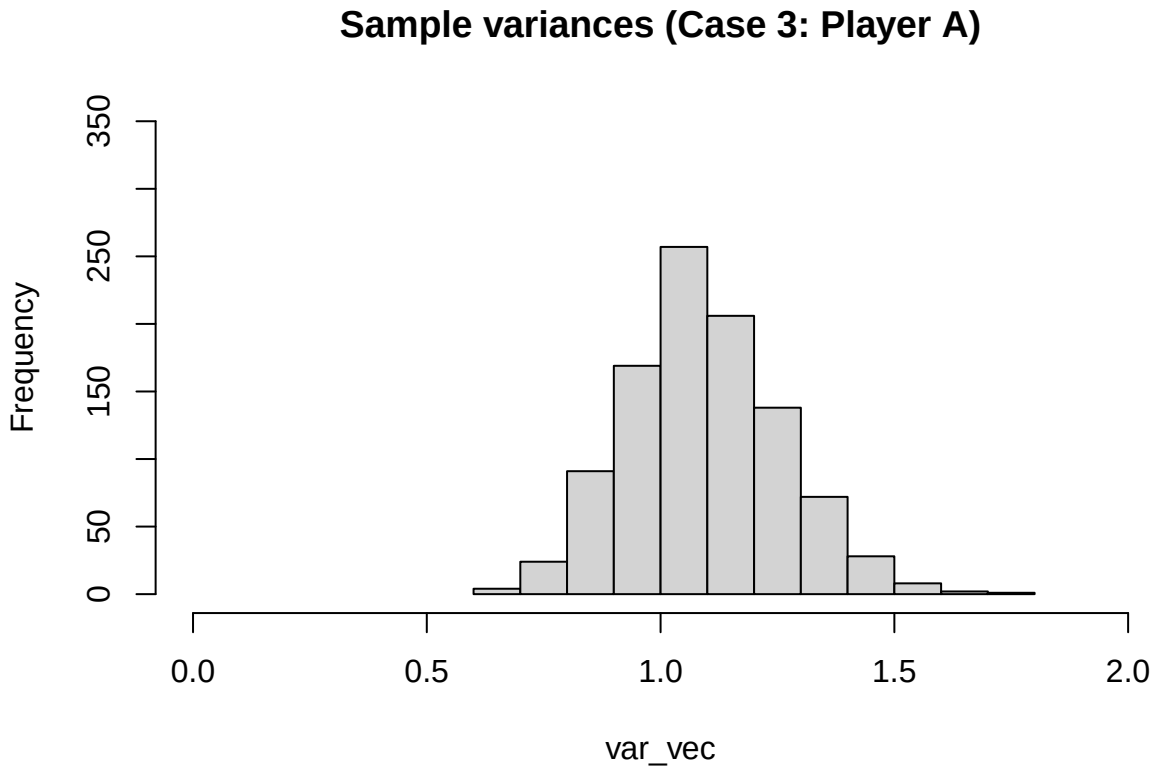
```
# Case 3
n = 100
m=1000
set.seed(0)
origData = rt(n,df=25); # case 3


# a) Player A
# generate 1000 data sets
set.seed(0)
z = rt(n*m,df=25)
mat = matrix(z,nrow=m)
#calculate variance for each row
var_vec = apply(mat,1,var)
samp_mean = mean(var_vec)
samp_var = var(var_vec)
print(paste("The sample mean is (Case 3: Player A): ", myRound(samp_mean)))
```

```
## [1] "The sample mean is (Case 3: Player A):  1.094"
```

```
print(paste("The sample variance is (Case 3: Player A): ", myRound(samp_var)))
```

```
## [1] "The sample variance is (Case 3: Player A):  0.027"
```

```
hist(var_vec,
     ylim=c(0,350),
     # breaks=20,
     xlim=c(0,2),
     main="Sample variances (Case 3: Player A)")
```



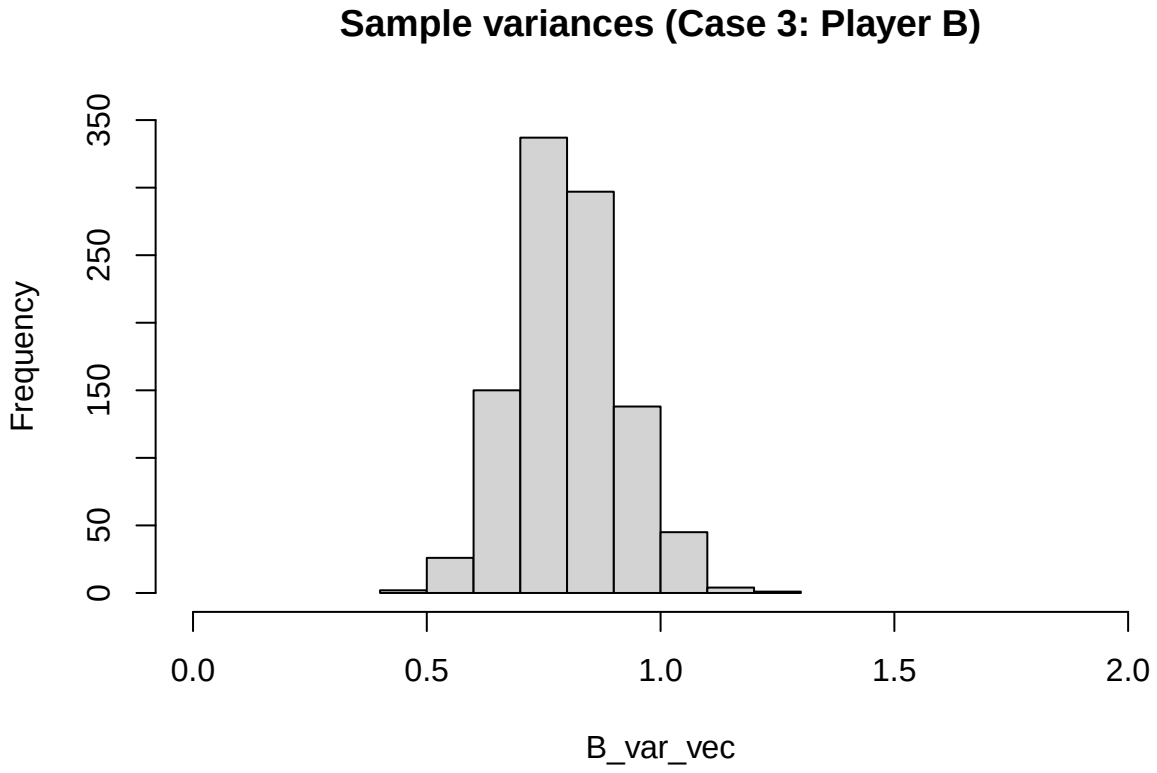

```
# b) Player B
# Estimate mean and variance from the sample data
B_var = var(origData)
B_mean = mean(origData)
# simulate new data with
set.seed(0)
B_mat = matrix(rnorm(n*m, mean = B_mean, sd = sqrt(B_var)),nrow=m)
B_var_vec = apply(B_mat,1,var)
B_samp_mean = mean(B_var_vec)
B_samp_var = var(B_var_vec)
print(paste("The sample mean is (Case 3: Player B): ", myRound(B_samp_mean)))
```

```
## [1] "The sample mean is (Case 3: Player B):  0.801"
```

```r
print(paste("The sample variance is (Case 3: Player B): ", myRound(B_samp_var)))
```

```
## [1] "The sample variance is (Case 3: Player B):  0.013"
```

```r
hist(B_var_vec,
     ylim=c(0,350),
     # breaks=20,
     xlim=c(0,2),
     main="Sample variances (Case 3: Player B)")
```



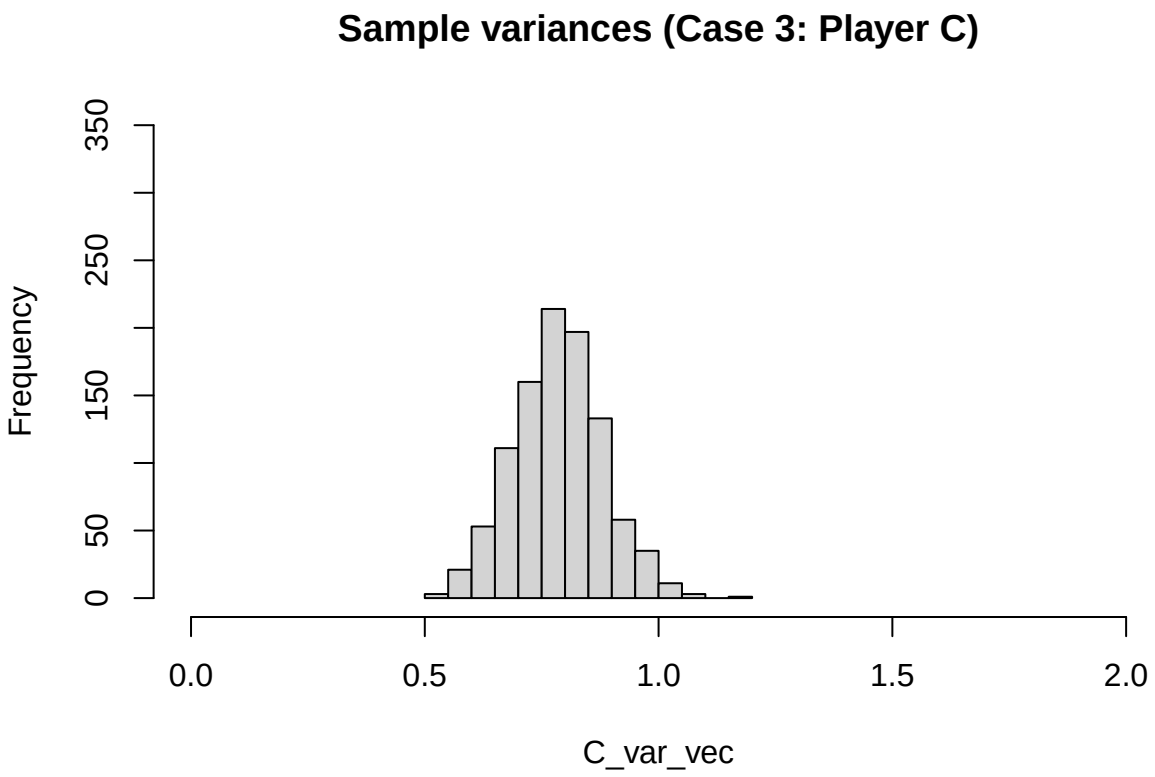

```r
# c) Player C
# sample new data
z=sample(x=origData, size=n*m, replace=TRUE)
C_mat = matrix(z,nrow=m)
C_var_vec = apply(C_mat,1,var)
C_samp_mean = mean(C_var_vec)
C_samp_var = var(C_var_vec)
print(paste("The sample mean is (Case 3: Player C): ", myRound(C_samp_mean)))
```

```
## [1] "The sample mean is (Case 3: Player C):  0.786"
```

```r
print(paste("The sample variance is (Case 3: Player C): ", myRound(C_samp_var)))
```

```
## [1] "The sample variance is (Case 3: Player C):  0.009"
```

```r
hist(C_var_vec,
     ylim=c(0,350),
     # breaks=20,
     xlim=c(0,2),
     main="Sample variances (Case 3: Player C)")
```

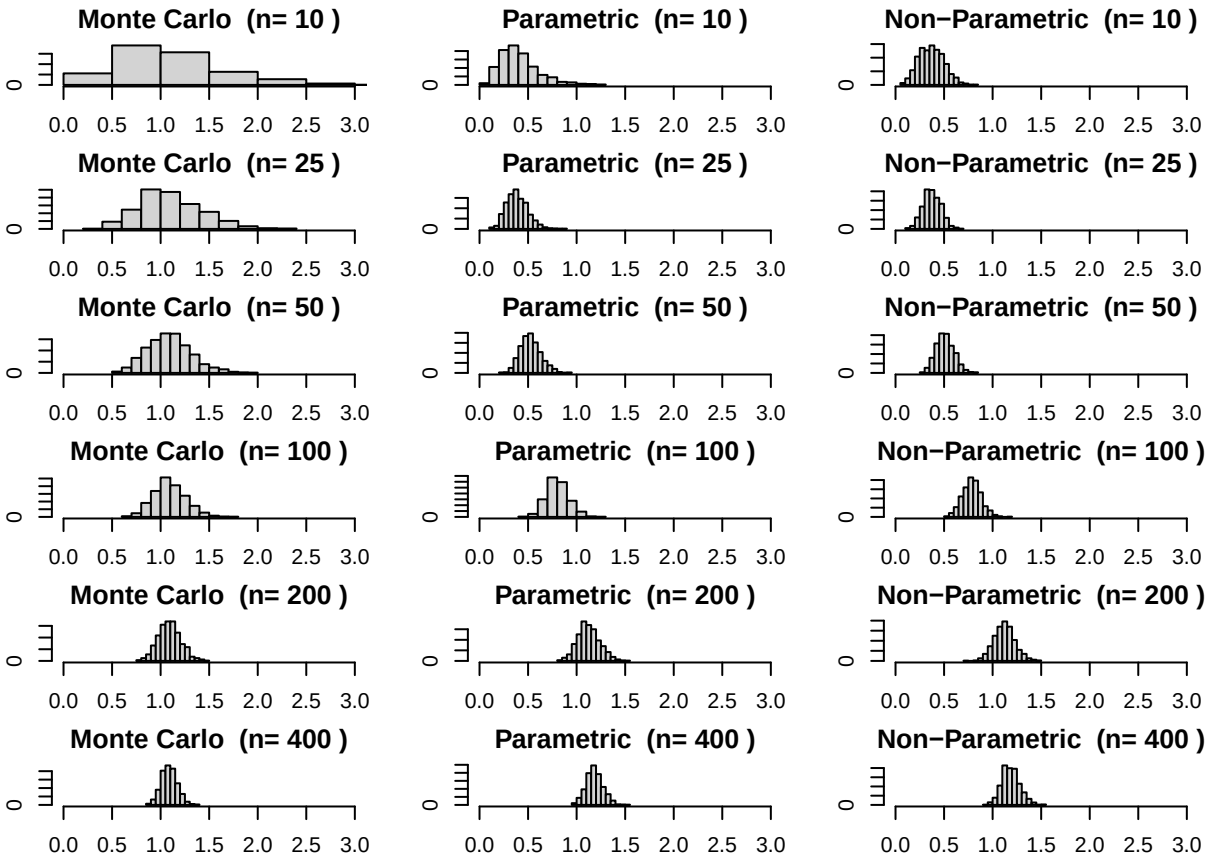


d)

For this case the performance of the parametric and non-parametric methods are better than what was seen in case 2. This is likely because the larger degrees of freedom in the true distribution. In this case the parametric approach slightly outperformed the non-parametric approach. This is likely due to the increased degrees of freedom. With a higher `df` value a t distribution looks more similar to a normal distribution. Therefore the effects of assuming a normal distribution in the parametric approach is reduced when compared to case 2. For this case I would say that the bias is low and the variance is low. This shows us that a drawback of the parametric approach is that it is very dependent on the assumption about the original data. If the assumption is more similar to the original data it can outperform the non-parametric approach. However, when the assumption is incorrect the non-parametric approach might be a better option. Additionally, we can note that a drawback for the non-parametric approach is that it is very dependent on the sample size of the original data set and more importantly how well it represents the true population, when the sample size is large and we are confident that the sample represents the population well then the non-parametric approach might be viable option. However, when the sample size is small and not representative of the true population a parametric approach might be better.

```r
# e)
n_vec=c(10, 25, 50, 100, 200, 400)
methods_vec = c("Monte Carlo", "Parametric", "Non-Parametric")
m=1000
set.seed(0)
data400 = rt(400,df=25)

# save results in tables
mean_res_df = data.frame(matrix(ncol=7, nrow=3))
var_res_df = data.frame(matrix(ncol=7, nrow=3))
colnames(mean_res_df) = c("Method", n_vec)
colnames(var_res_df) = c("Method", n_vec)
mean_res_df["Method"] = methods_vec
var_res_df["Method"] = methods_vec

par(mfrow=c(6,3),mar=c(2,2,2,2))
for(n in n_vec){
  mean_res_vec = c()
  var_res_vec = c()

  #Monte Carlo
  set.seed(0)
  z = rt(n*m, df=25)
  mat = matrix(z,nrow=m)
  var_vec = apply(mat,1,var)
  samp_mean = myRound(mean(var_vec))
  samp_var = myRound(var(var_vec))
  mean_res_vec = c(mean_res_vec, samp_mean)
  var_res_vec = c(var_res_vec, samp_var)
  hist(var_vec,
       # ylim=c(0,500),
       # breaks=20,
       xlim=c(0,3),
       main=paste("Monte Carlo ",
                  "(n=",n,")"))

  # Parametric
  # Estimate mean and variance from the sample data
  z = data400[1:n]
  B_var = var(z)
  B_mean = mean(z)
  # simulate new data with
  set.seed(0)
  B_mat = matrix(rnorm(n*m, mean = B_mean, sd = sqrt(B_var)),nrow=m)
  B_var_vec = apply(B_mat,1,var)
  B_samp_mean = myRound(mean(B_var_vec))
  B_samp_var = myRound(var(B_var_vec))
  mean_res_vec = c(mean_res_vec, B_samp_mean)
  var_res_vec = c(var_res_vec, B_samp_var)
```

```
  hist(B_var_vec,
       # ylim=c(0,500),
       # breaks=20,
       xlim=c(0,3),
       main=paste("Parametric ",
                  "(n=",n,")"))

  # c) Player C
  # sample new data
  z=sample(x=data400[1:n], size=n*m, replace=TRUE)
  C_mat = matrix(z,nrow=m)
  C_var_vec = apply(C_mat,1,var)
  C_samp_mean = myRound(mean(C_var_vec))
  C_samp_var = myRound(var(C_var_vec))
  mean_res_vec = c(mean_res_vec, C_samp_mean)
  var_res_vec = c(var_res_vec, C_samp_var)
  hist(C_var_vec,
       # ylim=c(0,500),
       # breaks=20,
       xlim=c(0,3),
       main=paste("Non-Parametric ",
                  "(n=",n,")"))

  mean_res_df[as.character(n)] = mean_res_vec
  var_res_df[as.character(n)] = var_res_vec
}
```




```
# display tables
knitr::kable(mean_res_df,
             format = "markdown",
             caption = "Mean for different n values (Case 3).")
```

Table 5: Mean for different n values (Case 3).

Method	10	25	50	100	200	400
Monte Carlo	1.089	1.095	1.090	1.094	1.091	1.088
Parametric	0.409	0.387	0.527	0.801	1.124	1.184
Non-Parametric	0.370	0.376	0.511	0.786	1.121	1.187

```
knitr::kable(var_res_df,
             format = "markdown",
             caption = "Variance for different n values (Case 3).")
```

Table 6: Variance for different n values (Case 3).

Method	10	25	50	100	200	400
Monte Carlo	0.283	0.103	0.051	0.027	0.013	0.007
Parametric	0.040	0.012	0.011	0.013	0.012	0.007

Method	10	25	50	100	200	400
Non-Parametric	0.016	0.008	0.009	0.009	0.011	0.008

e) For an increase in `n` we can see accuracy for both the parametric and non-parametric approaches improve to be very similar to what is produced by the Monte Carlo method at n=400. This is a very similar performance as to what was seen in case 1.

Problem 2

```
myCVids <- function(n, K, seed=0) {
# balanced subsets generation (subset sizes differ by at most 1)
# n is the number of observations/rows in the training set
# K is the desired number of folds (e.g., 5 or 10)
set.seed(seed);
t = floor(n/K); r = n-t*K;
id0 = rep((1:K),times=t)
ids = sample(id0,t*K)
if (r > 0) {ids = c(ids, sample(K,r))}
ids
}

# two-sample t test
column_t_test <- function(features, target){
  tests = lapply(seq(1,ncol(features)),function(x){t.test(features[,x]~target)})
  pval_lst = lapply(seq(1,length(tests)),function(x){tests[[x]]$p.value})
  return(list(tests, pval_lst))
}


# keep features with p-value <= 0.05
top_features <- function(pval_lst, p_val, p_min){
  bool_lst = lapply(seq(1,length(pval_lst)),function(x){pval_lst[[x]] < p_val})
  if(sum(as.integer(bool_lst))<=p_min){
    pval_df = t(data.frame(pval_lst))
    row.names(pval_df) <- NULL
    bool_df=data.frame(pval_sig=matrix(unlist(bool_lst), nrow=length(bool_lst),
                                       byrow=TRUE))
    bool_df$p_val <- pval_df[,1]
    selected_df = bool_df[bool_df$pval_sig==TRUE,]
    feat_index_vec = as.numeric(rownames(selected_df))
    return(feat_index_vec)
  }else{
    pval_df = t(data.frame(pval_lst))
    row.names(pval_df) <- NULL
    bool_df=data.frame(pval_sig=matrix(unlist(bool_lst), nrow=length(bool_lst),
                                       byrow=TRUE))
```

```
    bool_df$p_val <- pval_df[,1]
    bool_df$p_top = FALSE
    sorted_res = sort(bool_df$p_val, index.return=TRUE)
    bool_df[head(sorted_res$ix,p_min),]$p_top = TRUE
    bool_df$p_select = as.logical(bool_df$pval_sig*bool_df$p_top)
    selected_df = bool_df[bool_df$p_select==TRUE,]
    feat_index_vec = as.numeric(rownames(selected_df))
    return(feat_index_vec)
  }


}

# Mis-classification ratio calculation
MCR <- function(true_vals, pred_probs, threshold=0.5){
  if(length(true_vals)!=length(pred_probs)){
    print("ERROR: predictions and true values not of same shape")
  }else{
    pred_vals = as.integer((pred_probs > threshold))
    mcr = sum(pred_vals != true_vals)/length(true_vals)
    return(mcr)
  }
}
```

```
# Generate data
set.seed(0) # set seed
n = 25 # number of samples in each class
nr = n*2 # total number of samples
Y = c(rep(1,n),rep(0,n)) # target values
k=10 # k value for cross validation
p_star_vec = c(5,10,20,40) # p* values to select the top features in the data
i_vec = seq(5) #different values of i for different sized data sets

# create matrix where results will be stored
mcr_i_pstar_df = data.frame(matrix(0,
                                   nrow = length(p_star_vec),
                                   ncol = length(i_vec)+1))
mcr_i_pstar_df[,1] = p_star_vec # add p* values to results table

# loop over values of i to make different data sets
for(i in i_vec){
  nc = 200*2^i # nc = 6400
  M = matrix(rnorm(nr*nc),nrow=nr)
  X = M[,1:nc] # features of data
  # calculate cross validation indexes
  # this is applied only after feature selection
  ids = myCVids(n=nr, K=k, seed=0)
  # feature selection
  # apply two-sample t-test
```

```
t_test_results = column_t_test(features=X, target=Y)
# get all features according to p-value at differing values of p_star
# create vector to store mean mcr values for each p* subset
mean_pstar_mcr_vec = c()

# loop over p* values to create a subset of selected features
for(p_star in p_star_vec){
  feat_index_vec = c(top_features(pval_lst=t_test_results[[2]],
                                  p_val=0.05,
                                  p_min=p_star))

  X_pstar = X[,feat_index_vec]
  k_mcr_vec = c() # store all MCR values for each k

  # loop over k to calculate the MCR for each fold
  for( k in seq(k)){
    isk = (ids == k) # k varies from 1 to K
    valid.k = which(isk) # test data index
    train.k = which(!isk) # train data index
    # get all training data in single data frame
    train_df = data.frame(Y=Y[train.k], X_pstar[train.k,])
    # get all testing data in single data frame
    val_df = data.frame(Y=Y[valid.k], X_pstar[valid.k,])

    # train a Logistic regression model
    LR = glm(Y ~ .,
            data=train_df,
            family="binomial")

    # get estimated probabilities on the test data
    LR_probs = data.frame(
                  predict(LR,
                  val_df,
                  type ="response"
                  )
                )

    # use the probabilities to calculate the MCR
    mcr = MCR(
            true_vals=val_df$Y,
            pred_probs=LR_probs[,1],
            threshold=0.5)

    k_mcr_vec = c(k_mcr_vec, mcr)
  }
  mean_pstar_mcr = mean(k_mcr_vec) # get the mean MCR for all values of k

  mean_pstar_mcr_vec = c(mean_pstar_mcr_vec, mean_pstar_mcr)
}
```

```
  mcr_i_pstar_df[,i+1] = mean_pstar_mcr_vec
}
# add all data to a data frame and transform to be in the correct format
mcr_i_pstar_df = t(mcr_i_pstar_df)
rownames(mcr_i_pstar_df) = c("p*", i_vec)
knitr::kable(mcr_i_pstar_df, format = "markdown") # display table
```

p*	5.00	10.00	20.00	40.00
1	0.30	0.26	0.26	0.26
2	0.24	0.16	0.20	0.32
3	0.16	0.10	0.10	0.26
4	0.18	0.06	0.04	0.18
5	0.16	0.02	0.04	0.30

We can see that the error estimates are quite low for what we expect to see. Furthermore we can see that as the size of the original set of features increases we see a general steep drop in error rate. This is especially true when lower numbers of features are kept during selection. This is likely because only the best correlating features are selected and then the same data that was used to select these features are sued to test with again in cross-validation. This is wrong, we can tell that by performing feature selection before splitting the data into training and test sets that we do not get a true representation fo the performance as the selection process is part of training and has an influence on the test set when done before splitting the data.

Problem 3

```
# Generate data
set.seed(0) # set seed
n = 25 # number of samples in each class
nr = n*2 # total number of samples
Y = c(rep(1,n),rep(0,n)) # target values
k=10 # k value for cross validation
p_star_vec = c(5,10,20,40) # p* values to select the top features in the data
i_vec = seq(5) #different values of i for different sized data sets

# create matrix where results will be stored
mcr_i_pstar_df = data.frame(matrix(0,
                                   nrow = length(p_star_vec),
                                   ncol = length(i_vec)+1))
mcr_i_pstar_df[,1] = p_star_vec # add p* values to results table
progress=0
# loop over values of i to make different data sets
for(i in i_vec){
  nc = 200*2^i # nc = 6400
  M = matrix(rnorm(nr*nc),nrow=nr)
  X = M[,1:nc] # features of data
  # calculate cross validation indexes
```

```
# this is applied only after feature selection
ids = myCVids(n=nr, K=k, seed=0)

# get all features according to p-value at differing values of p_star
# create vector to store mean mcr values for each p* subset
mean_pstar_mcr_vec = c()

# loop over p* values to create a subset of selected features
for(p_star in p_star_vec){
  k_mcr_vec = c() # store all mcr values for each k

  # k-fold cross validation
  # loop over k to calculate the MCR for each fold
  for( k in seq(k)){
    progress=progress+1
    # print(myRound(progress/200)*100)
    isk = (ids == k) # k varies from 1 to K
    valid.k = which(isk) # test data index
    train.k = which(!isk) # train data index

    # feature selection
    # apply two-sample t-test
    t_test_results = column_t_test(features=X[train.k,], target=Y[train.k])
    feat_index_vec = c(top_features(pval_lst=t_test_results[[2]],
                                    p_val=0.05,
                                    p_min=p_star))
    X_pstar = X[,feat_index_vec]

    # get all training data in single data frame
    train_df = data.frame(Y=Y[train.k], X_pstar[train.k,])
    # get all testing data in single data frame
    val_df = data.frame(Y=Y[valid.k], X_pstar[valid.k,])

    # train a Logistic regression model
    LR = glm(Y ~ .,
             data=train_df,
             family="binomial")

    # get estimated probabilities on the test data
    LR_probs = data.frame(
                  predict(LR,
                  val_df,
                  type ="response"
                  )
                )

    # use the probabilities to calculate the MCR
    mcr = MCR(
             true_vals=val_df$Y,
```

```
              pred_probs=LR_probs[,1],
              threshold=0.5)

      k_mcr_vec = c(k_mcr_vec, mcr)
    }
    mean_pstar_mcr = mean(k_mcr_vec) # get the mean MCR for all values of k

    mean_pstar_mcr_vec = c(mean_pstar_mcr_vec, mean_pstar_mcr)
  }
  mcr_i_pstar_df[,i+1] = mean_pstar_mcr_vec
}
# add all data to a data frame and transform to be in the correct format
mcr_i_pstar_df = t(mcr_i_pstar_df)
rownames(mcr_i_pstar_df) = c("p*", i_vec)
knitr::kable(mcr_i_pstar_df, format = "markdown") # display table
```

p*	5.00	10.00	20.00	40.00
1	0.46	0.48	0.48	0.46
2	0.48	0.48	0.52	0.62
3	0.38	0.36	0.32	0.48
4	0.38	0.42	0.32	0.42
5	0.38	0.48	0.56	0.48

From these error estimates or mis-classifications rates (MCR) we can see that they are larger than what was seen in Problem 2. This is more in line with what was expected wit this type of data and classifier. Additionally, we do not see the same steep drop in error rate as the size of original features increases, similarly we do not see the same general decrease when lower numbers of features are selected as we saw in Problem 2. Even though these results are not as appealing as the ones seen in Problem 2, these are much more trustworthy as we do not interfere with the data before splitting it. Furthermore, it is worth noting that the approach in Problem 3 takes much more time to process than the one in Problem 2. It is clear that for reliable and more trustworthy results we pay the price of computational time.

Problem 4

```
# functions
myCVids <- function(n, K, seed=0) {
# balanced subsets generation (subset sizes differ by at most 1)
# n is the number of observations/rows in the training set
# K is the desired number of folds (e.g., 5 or 10)
set.seed(seed);
t = floor(n/K); r = n-t*K;
id0 = rep((1:K),times=t)
ids = sample(id0,t*K)
if (r > 0) {ids = c(ids, sample(K,r))}
ids
}
```

```r
genData <- function(n, seed=0) {
set.seed(seed)
x = seq(-1,1,length.out=n)
y = x - x^2 + 2*rnorm(n) # true sigma = 2;
out.df = data.frame(x=x, y=y)
out.df
}
```

4.1

```r
# parameters
set.seed(100)
d_vec = seq(0,4)
# generate data
train.df = genData(n=200,seed=100)
test.df = genData(400)

# create k-fold indexes
k=5
inds.part = myCVids(n=nrow(train.df),K=k)
# create a data frame to save results into
M = matrix(0, nrow = (length(d_vec)), ncol = k+1)
KFOLD_df = data.frame(M)
KFOLD_df[,1] = d_vec # add degrees to results table
colnames(KFOLD_df) = c("Degree", seq(k))

# loop over k-folds
for( k in seq(k)){
  isk = (inds.part == k) # k varies from 1 to K
  valid.k = which(isk) # test data index
  train.k = which(!isk)
  # split data
  train_sub_df = train.df[train.k,]
  valid_df = train.df[valid.k,]

  d_mse_vec = c()
  for(d in d_vec){
    if(d==0){
      # train a polynomial model
      PR = lm(y ~ 1, data=train_sub_df)
    }else{
     # train a polynomial model
      PR = lm(y ~ poly(x, d, raw = TRUE), data=train_sub_df)
    }
    # get predicted values to calculate MSE
    pred_val = predict(PR, valid_df, type="response")
    pred_true_val_df = data.frame(pred = pred_val, actual = valid_df$y)
```

```
    #calculate MSE
    mse = mean((pred_true_val_df$actual - pred_true_val_df$pred)^2)
    d_mse_vec = c(d_mse_vec, mse)
  }
  KFOLD_df[,k+1] = d_mse_vec
}

# tally of MSE
KFOLD_df$Total_MSE = rowSums(KFOLD_df[2:ncol(KFOLD_df)])
knitr::kable(KFOLD_df,
             format = "markdown",
             caption = "MSE by 5 folds at different degrees on validation data.") # display table
```

Table 9: MSE by 5 folds at different degrees on validation data.

Degree	1	2	3	4	5	Total_MSE
0	2.533138	2.984333	4.243087	4.513501	4.206231	18.48029
1	2.228435	2.853997	3.653187	4.190807	3.876836	16.80326
2	2.204849	2.797685	3.639950	4.126926	3.967901	16.73731
3	2.250757	2.872660	3.635394	4.190780	3.961687	16.91128
4	2.241189	2.894667	3.626206	4.490320	3.955490	17.20787

```
# print best d value
print(paste("The best K-fold degree is: ",
            as.character(KFOLD_df[which.min(KFOLD_df$Total_MSE),]$Degree)))
```

```
## [1] "The best K-fold degree is:  2"
```

The optimal tuning parameter for d in this case is 2. when d=2 the total MSE for all 5 folds in cross validation is the lowest. However, the difference between d=1 and d=2 is not very large and it might be more effective to go with d=1 in line wiht Occam's razor.

4.2

```
# create k-fold indexes
k=nrow(train.df)
inds.part = myCVids(n=nrow(train.df),K=k)
# create a data frame to save results into
M = matrix(0, nrow = (length(d_vec)), ncol = k+1)
LOOCV_df = data.frame(M)
LOOCV_df[,1] = d_vec # add degrees to results table
colnames(LOOCV_df) = c("Degree", seq(k))

# loop over k-folds
for( k in seq(k)){
```

```
  isk = (inds.part == k) # k varies from 1 to K
  valid.k = which(isk) # test data index
  train.k = which(!isk)
  # split data
  train_sub_df = train.df[train.k,]
  valid_df = train.df[valid.k,]

  d_mse_vec = c()
  for(d in d_vec){
    if(d==0){
      # train a polynomial model
      PR = lm(y ~ 1, data=train_sub_df)
    }else{
     # train a polynomial model
      PR = lm(y ~ poly(x, d, raw = TRUE), data=train_sub_df)
    }
    # get predicted values to calculate MSE
    pred_val = predict(PR, valid_df, type="response")
    pred_true_val_df = data.frame(pred = pred_val, actual = valid_df$y)
    #calculate MSE
    mse = mean((pred_true_val_df$actual - pred_true_val_df$pred)^2)
    d_mse_vec = c(d_mse_vec, mse)
  }
  LOOCV_df[,k+1] = d_mse_vec
}

# estimate test data MSE
test_d_vec = c()
for(d in d_vec){
  if(d==0){
    # train a polynomial model
    PR = lm(y ~ 1, data=train.df)
  }else{
   # train a polynomial model
    PR = lm(y ~ poly(x, d, raw = TRUE), data=train.df)
  }
  # get predicted values to calculate MSE
  pred_val = predict(PR, test.df, type="response")
  pred_true_val_df = data.frame(pred = pred_val, actual = test.df$y)
  #calculate MSE
  mse = mean((pred_true_val_df$actual - pred_true_val_df$pred)^2)
  test_d_vec = c(test_d_vec, mse)
}

# tally of MSE
LOOCV_df$Total_validation_MSE = rowSums(LOOCV_df[2:ncol(LOOCV_df)])
LOOCV_df$Test_MSE = test_d_vec
knitr::kable(LOOCV_df[,c(1,k+2,k+3)],
             format = "markdown",
```

```
            caption = "Total MSE of LOOCV at different degrees on validation data.") # display t
```

Table 10: Total MSE of LOOCV at different degrees on validation data.

Degree	Total_validation_MSE	Test_MSE
0	740.7244	4.139705
1	676.5361	3.784582
2	677.7304	3.709157
3	683.7229	3.718672
4	689.2117	3.727459

```
# print best d value
print(paste("The best LOOCV degree is: ",
            as.character(LOOCV_df[which.min(LOOCV_df$Total_validation_MSE),]$Degree)))
```

```
## [1] "The best LOOCV degree is:  1"
```

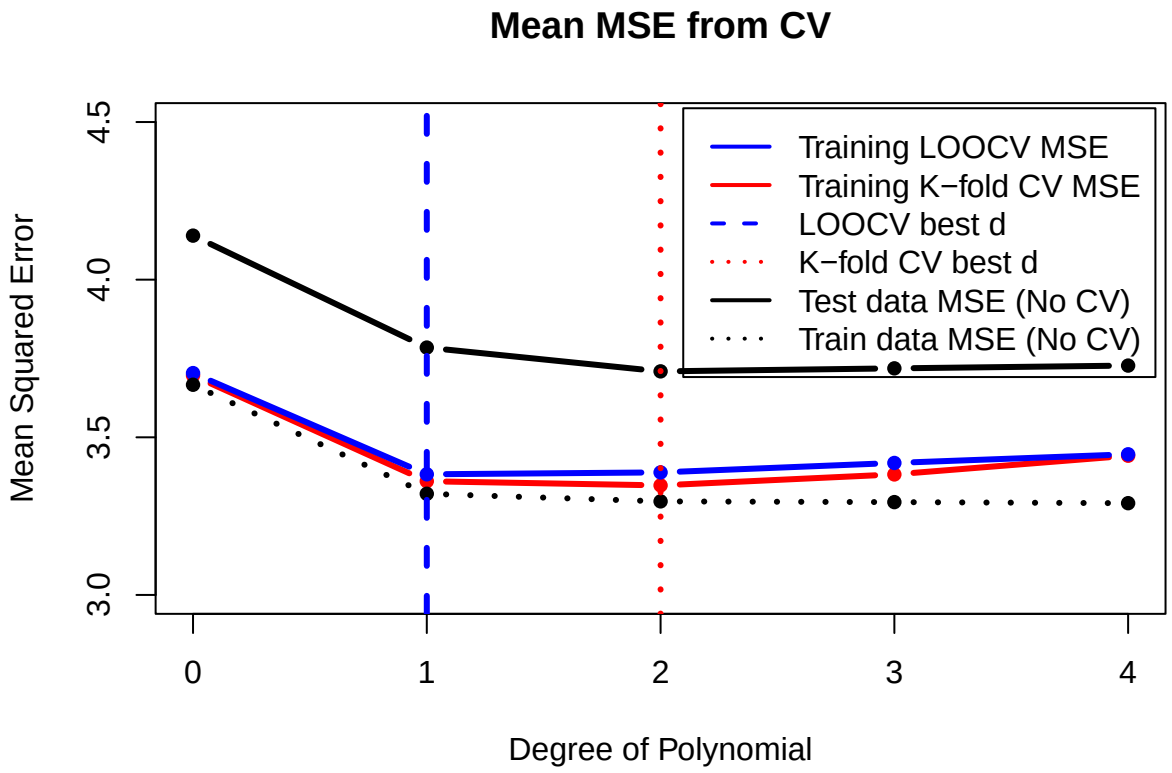
From these results we can see that the lowest total MSE on the validation data shows that a degree of 1 leads to the best model. However from the test MSE we can see that the degree of 1 has the best performance. However, the difference in performance between degrees 1, 2, 3, and 4 is not much and almost negligible and it is still satisfactory to settle for a degree of 1 when following the rule of Occam's razor. This means that when a bunch of models perform equally well it is best to go with the simplest model. We can see here that LOOCV gives us the same answer as we got from 5-fold CV. However, it might still be better to opt for 5-fold CV as LOOCV does not scramble the data enough with each fold highly correlated to the others leading to a high variance. Usually LOOCV should also be much more computationally expensive, however that is not the case when done on polynomial regression.

4.3

```
# estimate test data MSE
train_d_vec = c()
test_d_vec = c()
for(d in d_vec){
  if(d==0){
    # train a polynomial model
    PR = lm(y ~ 1, data=train.df)
  }else{
   # train a polynomial model
    PR = lm(y ~ poly(x, d, raw = TRUE), data=train.df)
  }
  # get predicted values to calculate MSE
  pred_val = predict(PR, test.df, type="response")
  pred_true_val_df = data.frame(pred = pred_val, actual = test.df$y)
  #calculate MSE
  mse = mean((pred_true_val_df$actual - pred_true_val_df$pred)^2)
```

```
  test_d_vec = c(test_d_vec, mse)

  # get predicted values to calculate MSE
  pred_val = predict(PR, train.df, type="response")
  pred_true_val_df = data.frame(pred = pred_val, actual = train.df$y)
  #calculate MSE
  mse = mean((pred_true_val_df$actual - pred_true_val_df$pred)^2)
  train_d_vec = c(train_d_vec, mse)
}

# visualize LOOCV and K-fold CV
# calculate max and min limits of data KFOLD
# kfold_max_vec = apply(KFOLD_df[, 2:(ncol(KFOLD_df)-1)], 1, max)
# kfold_min_vec = apply(KFOLD_df[, 2:(ncol(KFOLD_df)-1)], 1, min)
kfold_mean_vec = apply(KFOLD_df[, 2:(ncol(KFOLD_df)-1)], 1, mean)
# calculate max and min limits of data for LOOCV
# loocv_max_vec = apply(LOOCV_df[, 2:(ncol(LOOCV_df)-1)], 1, max)
# loocv_min_vec = apply(LOOCV_df[, 2:(ncol(LOOCV_df)-1)], 1, min)
loocv_mean_vec = apply(LOOCV_df[, 2:(ncol(LOOCV_df)-2)], 1, mean)



{plot(x=KFOLD_df$Degree,
     y=kfold_mean_vec,
     ylab="Mean Squared Error",
     main="Mean MSE from CV",
     xlab="Degree of Polynomial",
     type='b',
     col='red',
     pch = 16,
     lwd=3,
     ylim=c(3,4.5)
     )
lines(x=LOOCV_df$Degree,
     y=loocv_mean_vec,
     type='b',
     col='blue',
     pch = 16,
     lwd=3)
lines(x=LOOCV_df$Degree,
     y=test_d_vec,
     type='b',
     col='black',
     pch = 16,
     lwd=3)
lines(x=LOOCV_df$Degree,
     y=train_d_vec,
     type='b',
     col='black',
```

```
      lty=3,
      pch = 16,
      lwd=3)
abline(v=KFOLD_df[which.min(KFOLD_df$Total_MSE),]$Degree,
      col='red',
      pch = 16,
      lty=3,
      lwd=3)
abline(v=LOOCV_df[which.min(LOOCV_df$Total_validation_MSE),]$Degree,
      col='blue',
      pch = 16,
      lty=2,
      lwd=3)
legend("topright",
       inset = 0.01,
       legend = c("Training LOOCV MSE",
                  "Training K-fold CV MSE",
                  "LOOCV best d",
                  "K-fold CV best d",
                  "Test data MSE (No CV)",
                  "Train data MSE (No CV)"),
       lty = c(1,1,2,3,1,3),
       col = c("blue", "red","blue", "red", "black", "black"),
       lwd = 2)}
```



On this data set we can see that the LOOCV and 5-fold CV have a very similar trend when looking at

average MSE for each fold at each degree. This is not entirely in line with the test set of data, but it is following the same trend. The CV approaches give a MSE that is closer to the test MSE than what the training MSE would give. This indicates that these approaches are indeed useful to estimate parameters even though they reduce training data set size.

4.4

```
# parameters
set.seed(100)
d_vec = seq(0,4)
# generate data
train.df = genData(n=400,seed=100)
test.df = genData(400)

# create k-fold indexes
k=5
inds.part = myCVids(n=nrow(train.df),K=k)
# create a data frame to save results into
M = matrix(0, nrow = (length(d_vec)), ncol = k+1)
KFOLD_df = data.frame(M)
KFOLD_df[,1] = d_vec # add degrees to results table
colnames(KFOLD_df) = c("Degree", seq(k))

# loop over k-folds
for( k in seq(k)){
  isk = (inds.part == k) # k varies from 1 to K
  valid.k = which(isk) # test data index
  train.k = which(!isk)
  # split data
  train_sub_df = train.df[train.k,]
  valid_df = train.df[valid.k,]

  d_mse_vec = c()
  for(d in d_vec){
    if(d==0){
      # train a polynomial model
      PR = lm(y ~ 1, data=train_sub_df)
    }else{
     # train a polynomial model
      PR = lm(y ~ poly(x, d, raw = TRUE), data=train_sub_df)
    }
    # get predicted values to calculate MSE
    pred_val = predict(PR, valid_df, type="response")
    pred_true_val_df = data.frame(pred = pred_val, actual = valid_df$y)
    #calculate MSE
    mse = mean((pred_true_val_df$actual - pred_true_val_df$pred)^2)
    d_mse_vec = c(d_mse_vec, mse)
  }
```

```
  KFOLD_df[,k+1] = d_mse_vec
}

# tally of MSE
KFOLD_df$Total_MSE = rowSums(KFOLD_df[2:ncol(KFOLD_df)])
knitr::kable(KFOLD_df,
             format = "markdown",
             caption = "MSE by 5 folds at different degrees on validation data.") # display table
```

Table 11: MSE by 5 folds at different degrees on validation data.

Degree	1	2	3	4	5	Total_MSE
0	3.438751	4.942601	3.672744	4.779153	4.645821	21.47907
1	3.584766	4.483129	3.614016	4.226312	4.321004	20.22923
2	3.427265	4.326183	3.499535	4.504805	4.064654	19.82244
3	3.381529	4.333047	3.556816	4.469806	4.060612	19.80181
4	3.375970	4.375542	3.547353	4.476218	4.128512	19.90359

```
# print best d value
print(paste("The best K-fold degree is: ",
            as.character(KFOLD_df[which.min(KFOLD_df$Total_MSE),]$Degree)))
```

```
## [1] "The best K-fold degree is:  3"
```

```
# create k-fold indexes
k=nrow(train.df)
inds.part = myCVids(n=nrow(train.df),K=k)
# create a data frame to save results into
M = matrix(0, nrow = (length(d_vec)), ncol = k+1)
LOOCV_df = data.frame(M)
LOOCV_df[,1] = d_vec # add degrees to results table
colnames(LOOCV_df) = c("Degree", seq(k))

# loop over k-folds
for( k in seq(k)){
  isk = (inds.part == k) # k varies from 1 to K
  valid.k = which(isk) # test data index
  train.k = which(!isk)
  # split data
  train_sub_df = train.df[train.k,]
  valid_df = train.df[valid.k,]

  d_mse_vec = c()
  for(d in d_vec){
    if(d==0){
      # train a polynomial model
```

```
      PR = lm(y ~ 1, data=train_sub_df)
    }else{
     # train a polynomial model
      PR = lm(y ~ poly(x, d, raw = TRUE), data=train_sub_df)
    }
    # get predicted values to calculate MSE
    pred_val = predict(PR, valid_df, type="response")
    pred_true_val_df = data.frame(pred = pred_val, actual = valid_df$y)
    #calculate MSE
    mse = mean((pred_true_val_df$actual - pred_true_val_df$pred)^2)
    d_mse_vec = c(d_mse_vec, mse)
  }
  LOOCV_df[,k+1] = d_mse_vec
}

# estimate test data MSE
test_d_vec = c()
for(d in d_vec){
  if(d==0){
    # train a polynomial model
    PR = lm(y ~ 1, data=train.df)
  }else{
   # train a polynomial model
    PR = lm(y ~ poly(x, d, raw = TRUE), data=train.df)
  }
  # get predicted values to calculate MSE
  pred_val = predict(PR, test.df, type="response")
  pred_true_val_df = data.frame(pred = pred_val, actual = test.df$y)
  #calculate MSE
  mse = mean((pred_true_val_df$actual - pred_true_val_df$pred)^2)
  test_d_vec = c(test_d_vec, mse)
}

# tally of MSE
LOOCV_df$Total_validation_MSE = rowSums(LOOCV_df[2:ncol(LOOCV_df)])
LOOCV_df$Test_MSE = test_d_vec
knitr::kable(LOOCV_df[,c(1,k+2,k+3)],
             format = "markdown",
             caption = "Total MSE of LOOCV at different degrees on validation data.") # display t
```

Table 12: Total MSE of LOOCV at different degrees on validation data.

Degree	Total_validation_MSE	Test_MSE
0	1722.641	4.142277
1	1614.560	3.790363
2	1578.588	3.687165
3	1578.030	3.684597

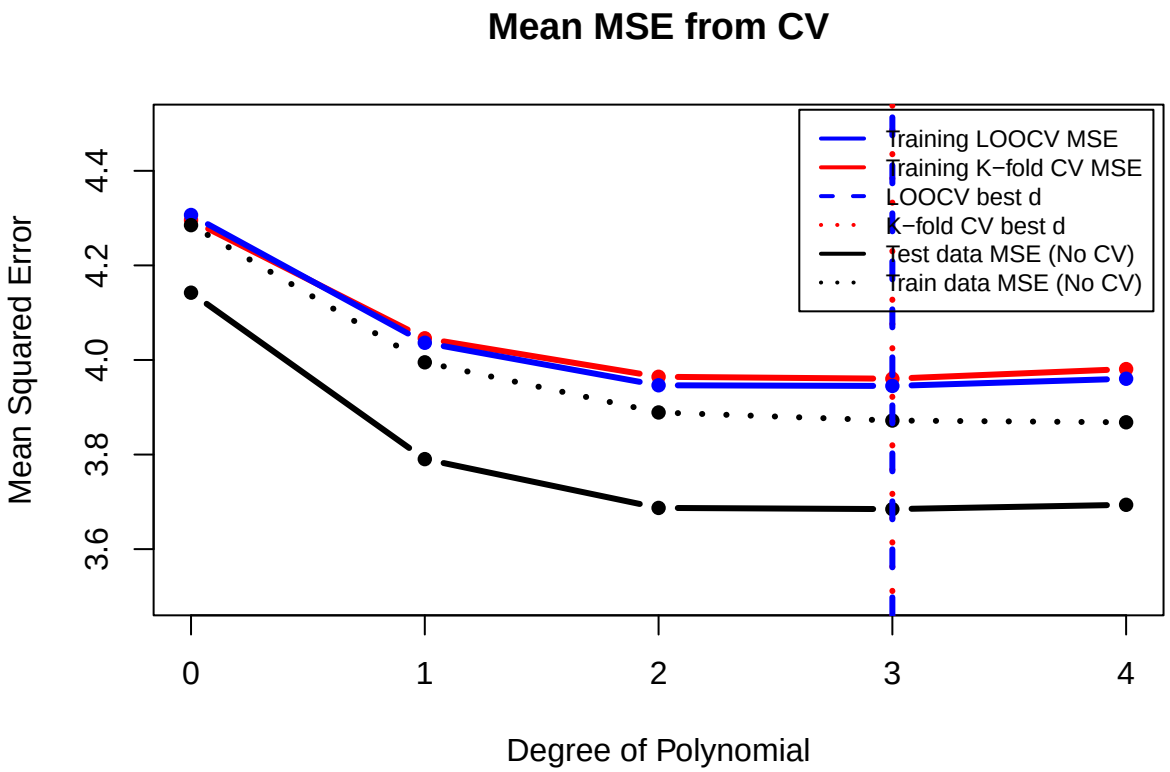
Degree	Total_validation_MSE	Test_MSE
4	1584.124	3.693714

```
# print best d value
print(paste("The best LOOCV degree is: ",
            as.character(LOOCV_df[which.min(LOOCV_df$Total_validation_MSE),]$Degree)))
```

```
## [1] "The best LOOCV degree is:  3"
```

```
# estimate test data MSE
train_d_vec = c()
test_d_vec = c()
for(d in d_vec){
  if(d==0){
    # train a polynomial model
    PR = lm(y ~ 1, data=train.df)
  }else{
   # train a polynomial model
    PR = lm(y ~ poly(x, d, raw = TRUE), data=train.df)
  }
  # get predicted values to calculate MSE
  pred_val = predict(PR, test.df, type="response")
  pred_true_val_df = data.frame(pred = pred_val, actual = test.df$y)
  #calculate MSE
  mse = mean((pred_true_val_df$actual - pred_true_val_df$pred)^2)
  test_d_vec = c(test_d_vec, mse)

  # get predicted values to calculate MSE
  pred_val = predict(PR, train.df, type="response")
  pred_true_val_df = data.frame(pred = pred_val, actual = train.df$y)
  #calculate MSE
  mse = mean((pred_true_val_df$actual - pred_true_val_df$pred)^2)
  train_d_vec = c(train_d_vec, mse)
}

# visualize LOOCV and K-fold CV
# calculate max and min limits of data KFOLD
# kfold_max_vec = apply(KFOLD_df[, 2:(ncol(KFOLD_df)-1)], 1, max)
# kfold_min_vec = apply(KFOLD_df[, 2:(ncol(KFOLD_df)-1)], 1, min)
kfold_mean_vec = apply(KFOLD_df[, 2:(ncol(KFOLD_df)-1)], 1, mean)
# calculate max and min limits of data for LOOCV
# loocv_max_vec = apply(LOOCV_df[, 2:(ncol(LOOCV_df)-1)], 1, max)
# loocv_min_vec = apply(LOOCV_df[, 2:(ncol(LOOCV_df)-1)], 1, min)
loocv_mean_vec = apply(LOOCV_df[, 2:(ncol(LOOCV_df)-2)], 1, mean)

{plot(x=KFOLD_df$Degree,
     y=kfold_mean_vec,
     ylab="Mean Squared Error",
```

```
     main="Mean MSE from CV",
     xlab="Degree of Polynomial",
     type='b',
     col='red',
     pch = 16,
     lwd=3,
     ylim=c(3.5,4.5)
     )
lines(x=LOOCV_df$Degree,
     y=loocv_mean_vec,
     type='b',
     col='blue',
     pch = 16,
     lwd=3)
lines(x=LOOCV_df$Degree,
     y=test_d_vec,
     type='b',
     col='black',
     pch = 16,
     lwd=3)
lines(x=LOOCV_df$Degree,
     y=train_d_vec,
     type='b',
     col='black',
     lty=3,
     pch = 16,
     lwd=3)
abline(v=KFOLD_df[which.min(KFOLD_df$Total_MSE),]$Degree,
     col='red',
     pch = 16,
     lty=3,
     lwd=3)
abline(v=LOOCV_df[which.min(LOOCV_df$Total_validation_MSE),]$Degree,
     col='blue',
     pch = 16,
     lty=2,
     lwd=3)
legend("topright",
       inset = 0.01,
       legend = c("Training LOOCV MSE",
                  "Training K-fold CV MSE",
                  "LOOCV best d",
                  "K-fold CV best d",
                  "Test data MSE (No CV)",
                  "Train data MSE (No CV)"),
       lty = c(1,1,2,3,1,3),
       col = c("blue", "red","blue", "red", "black", "black"),
       lwd = 2,
       cex = 0.75)}
```



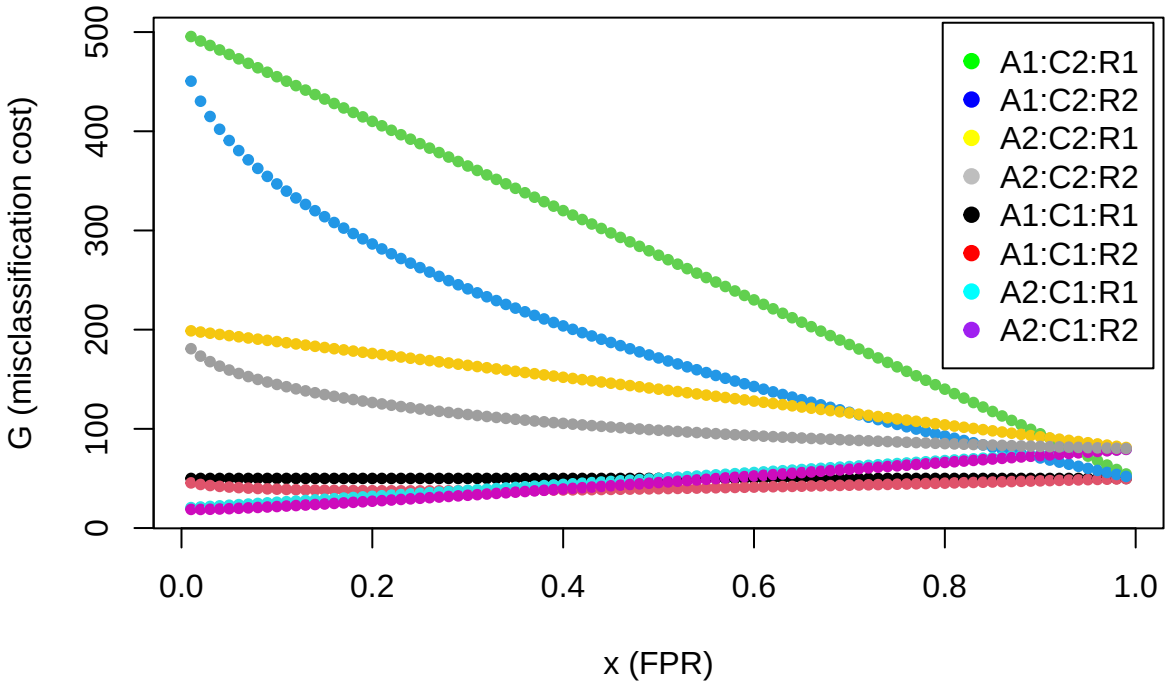
In this example with a larger data set we can see that the optimal degree is 3 for both CV approaches. This is accurate to what is represented in the test data. The test data also actually presents a MSE that is lower than the training data MSE. This is likely just an artifact of the random data set generation and is quite unexpected. Even though a degree of 3 is chosen here for both LOOCV and 5-fold CV it might be better to opt for a degree of 2 as the decrease in MSE from a degree of two to 3 is not as much and might not warrant the added complexity of the model.

Problem 5

```
# assume values for x and cfp
# x = 0.25
cfp=1
n=100


x_vec=c()
A_vec=c()
C_vec=c()
R_vec=c()
FPR_vec=c()
TPR_vec=c()
G_vec=c()
for (x in seq(0.01,0.99,0.01)) {
  A=c(0.5, 0.2)
```

```
  C=c(cfp, 10*cfp)
  R=c(x, sqrt(x))

  Ai=0
  for(q in A){
    Ai = Ai+1
    P=n*q
    N=n*(1-q)

    Ri=0
    for(tpr in R){
      Ri=Ri+1
      # calculate TP, FP, TN, and FN with regards to x
      TP = tpr*P
      FN = P-TP
      FP = x*N
      TN = N-FP
      FPR = x
      TPR = tpr

      Ci=0
      for(cfn in C){
        Ci=Ci+1
        G = cfn*FN + cfp*FP

        x_vec=c(x_vec,x)
        A_vec=c(A_vec,Ai)
        C_vec=c(C_vec,Ci)
        R_vec=c(R_vec,Ri)
        FPR_vec=c(FPR_vec,FPR)
        TPR_vec=c(TPR_vec,TPR)
        G_vec=c(G_vec,G)

        # print(paste("(A:",as.character(Ai),")"))
        # print(paste("(C:",as.character(Ci),")"))
        # print(paste("(R:",as.character(Ri),")"))
        # print(paste("FPR = ", FPR))
        # print(paste("TPR = ", TPR))
        # print(paste("G = ", G))
        # print("----------------------------")


      }
    }

  }
}

results_df = data.frame(x_vec,
```

```
                          A_vec,
                          C_vec,
                          R_vec,
                          FPR_vec,
                          TPR_vec,
                          G_vec)

results_df$design_id <- paste(results_df$A_vec,
                              results_df$C_vec,
                              results_df$R_vec)

# Objective function score visualization
{plot(x=results_df$x_vec,
     y=results_df$G_vec,
     col=factor(results_df$design_id),
     ylab="G (misclassification cost)",
     xlab="x (FPR)",
     pch=20)
legend("topright",
     inset = 0.01,
     pch=c(19,19,19,19),
     legend = c("A1:C2:R1",
                "A1:C2:R2",
                "A2:C2:R1",
                "A2:C2:R2",
                "A1:C1:R1",
                "A1:C1:R2",
                "A2:C1:R1",
                "A2:C1:R2"),
     col = c("green",
             "blue",
             "yellow",
             "grey",
             "black",
             "red",
             "cyan",
             "purple"))}
```

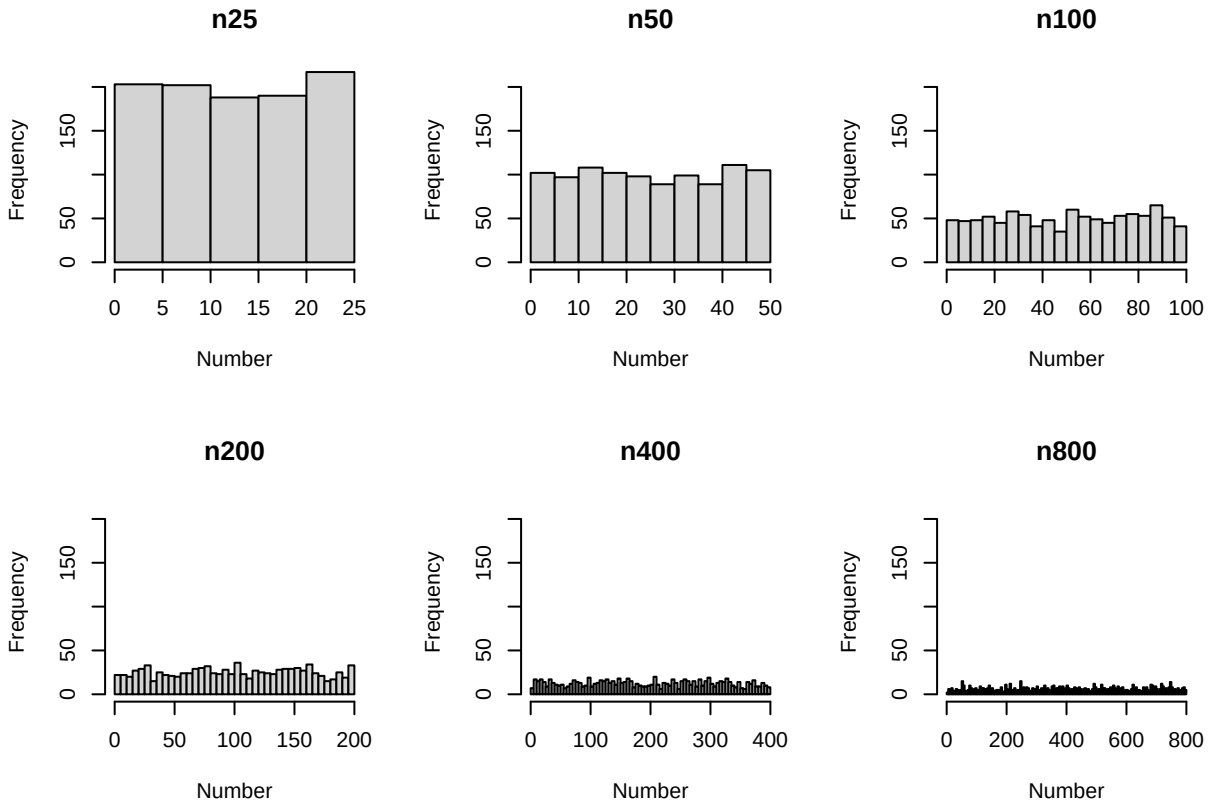


The best classifier is the one that minimizes the objective function the best over values of x. In this case it was one with the design combinations of (A:1 C1: R:2) or (A:2 C1: R:2). For higher values of x an unbalanced population gives higher values from the objective function. A1 is thus better with even populations when. This is because with an uneven class ratio the mis-classification rate increases for higher values of x because a higher x value means a higher false positive count. Even when the false negative count is low. This classifier is therefore not stable for all values of x and we choose the (A:1 C1: R:2) combination over the (A:2 C1: R:2) combination. Even though the (A:2 C1: R:2) combination clearly performs better at lower values of x. This in turn increased the mis-classifications calculated in the objective function. All classifiers that had a cfn that was 10 times the cfp resulted in an objective function that was orders of magnitude larger than the other classifiers. This makes intuitive senses as it dramatically increases the net cost of mis-classifications. With a TPR that is square rooted the objective function consistently returns a lower mis-classifications score. R2 is thus better. This is because, when the square root of the TPR is used to derive True positives they are higher than than when the TPR is not square rooted. This in turn decreases the amount of mis-classifications.

Problem 6

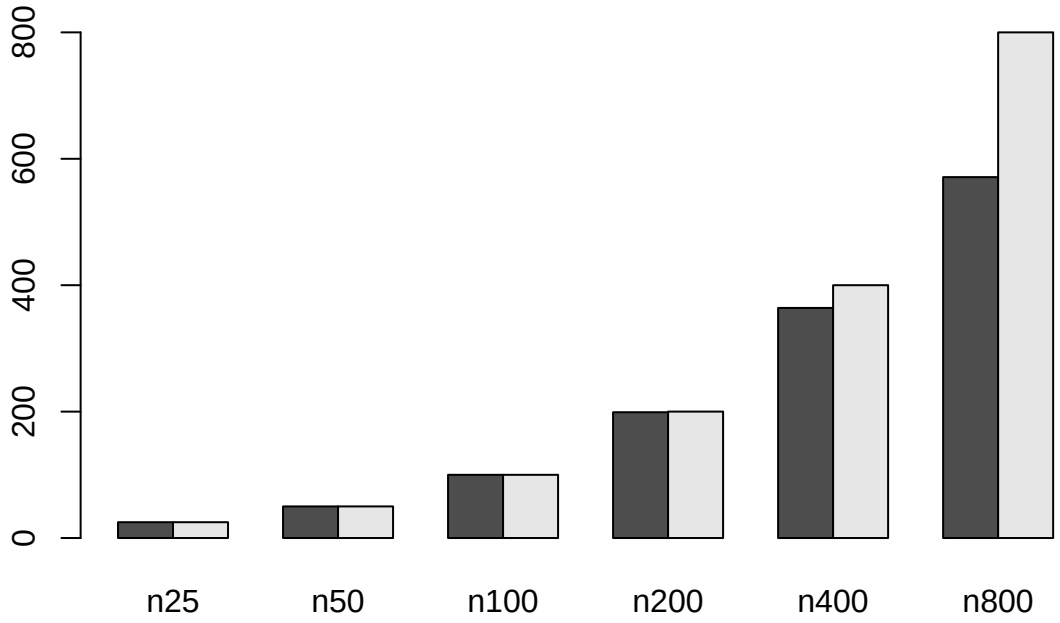
```
# Generate data
set.seed(0)
n_vec = c(25, 50, 100, 200, 400, 800)
m = 1000
df = data.frame(m=1:m)
table_df = data.frame(n=n_vec) # results table
mean_uniq_vec = c()
```

```r
sd_uniq_vec = c()
for (n in n_vec){
  I = seq(n)
  samp_vec = c(sample(x=I, size=m ,replace=TRUE))
  df[paste("n",n , sep="")] = samp_vec

  mean_uniq_vec = c(mean_uniq_vec, mean(unique(samp_vec)))
  sd_uniq_vec = c(sd_uniq_vec, sd(unique(samp_vec)))
}

# #visualize data
{par(mfrow=c(2,3))
for(i in names(df)[2:7]){
  hist(df[[i]],
       xlab = "Number",
       main=i,
       ylim=c(0,210),
       breaks = as.numeric(substr(i,2,nchar(i)))/5)
}}
```




```r
# bar plot of of unique values for different values of n
unique_df = t(data.frame(unique_sample_count = apply(df[, 2:(ncol(df))], 2, FUN=function(x){lengt
                         n=n_vec))
{par(mfrow=c(1,1))
```

```
  barplot(
  unique_df,
  beside=TRUE
  )}
```




```
# make table of results
table_df$Mean_Unique = myRound(mean_uniq_vec, acc=2)
table_df$SD_Unique = myRound(sd_uniq_vec, acc=2)
table_df = data.frame(t(table_df))

knitr::kable(table_df, format = "markdown")
```

	X1	X2	X3	X4	X5	X6
n	25.00	50.00	100.00	200.00	400.00	800.00
Mean_Unique	13.00	25.50	50.50	100.75	200.02	408.84
SD_Unique	7.36	14.58	29.01	57.92	116.29	230.78

We can see that as our number of replications stay constant at 1000 that the larger our sampling range gets that there is a lower frequency of each of the possible unique numbers that gets sampled. The frequency for each unique value to be sampled from `n` reduces as `n` increases. We can similarly see in the bar plot that the amount of unique numbers sampled is approximately the same as `n` for increasing values of `n` up to the point where `n`=400. From `n`=400 and `n`=800 we can see that there were some numbers

that were not even sampled. The amount of numbers that do not get sampled seem to increase with the size of `n`.

In the table we can see that the mean unique numbers are all approximately half the size of `n`. It also shows that the standard deviation is approximately a third of the size of `n`. The mean and standard deviation and mean is kept constant in proportion to the sample size for the nonparametric bootstrap method.