Homework 1, STATS 315A

Stanford University, Winter 2019

Joe Higgins

Question 1

Linear versus Knn: you are to run a simulation to compare KNN and linear regression in terms of their performance as a classifier, in the presence of an increasing number of noise variables. We will use a binary response variable Y taking values $\{0, 1\}$, and initially X in R^2. The joint distribution for (Y, X) is $(1-\pi)f_0(x)$ if $y = 0$ $h_{YX}(y, x) = \pi f_1(x)$ if $y = 1$ where $f_0(x)$ and $f_1(x)$ are each a mixture of K Gaussians: $f_j(x) = \sum_{k=1}^{K} \omega_{kj}\phi(x; \mu_{kj}, \Sigma_{kj}), j = 0, 1$ (1) $\phi(x; \mu, \Sigma)$ is the density function for a bivariate Gaussian with mean vector μ and covariance matrix Σ, and the $0 < \omega_{kj} < 1$ are the mixing proportions, with $\sum_k \omega_{kj} = 1$.

(a) We will use $\pi = .5$, $K = 6$, $\omega_{kj} = \frac{1}{6}$ and $\Sigma_{kj} = \sigma^2 I = 0.2 = I$, $\forall k, j$. The six location vectors in each class are simulated once and then fixed. Use a standard bivariate gaussian with covariance I, and mean-vector $(0, 1)$ for class 0 and $(1, 0)$ for class 1 to generate the 12 location vectors.

```
#Set up sampling parameters and location vectors
rm(list = ls())
pi <- 0.5
K <- 6
J <- 1
num_classes <- J+1
omega <- matrix(1/6, nrow = K, ncol = J)
sigma2 <- 0.2

mu_0 <- c(0, 1)
mu_1 <- c(1, 0)
location_vectors_0 <- mvrnorm(K, mu = mu_0, Sigma = diag(2))
location_vectors_1 <- mvrnorm(K, mu = mu_1, Sigma = diag(2))

location_vectors <- array(c(location_vectors_0, location_vectors_1), dim = c(K,num_classes,num_classes))
```

(b) Write a function to generate a sample of N points from a density as in (1), with the parameter settings as in 1(a). The function takes as inputs a centroid matrix, N, *pi*, *omega* (vector) and *sigma*2 (σ^2), and outputs a matrix X.

```
mixture_of_Gaussians <- function(j, centroid, omega, sigma2){
  k <- sample((nrow(omega)), 1, prob=omega)
  location <- mvrnorm(1, mu = c(centroid[k,1,j+1], centroid[k,2,j+1]), Sigma = sqrt(sigma2)*diag(2))
  return(location)
}

generate_sample <- function(centroid, N, pi, omega, sigma2){
  X <- matrix(NA, nrow=N, ncol=3)
  for(i in 1:N){
    y <- rbinom(1, 1, pi)
    x <- mixture_of_Gaussians(y, centroid, omega, sigma2)
    X[i,] <- c(y, x)
  }
  return(X)
}
```

(c) Use your function to generate a training set of size 300 from h_{YX} , as well as a test set of size 20K. This should leave you with `xtrain`, `ytrain`, `xtest` and `ytest`.

```
train_size <- 300
test_size <- 20000

X <- generate_sample(location_vectors, train_size, pi, omega, sigma2)
y_train <- X[,1]
x_train <- X[,2:3]
X <- generate_sample(location_vectors, test_size, pi, omega, sigma2)
y_test <- X[,1]
x_test <- X[,2:3]
```

(d) Write a function to compute the Bayes classifier for this setup. It should take as input the 12×2 matrix of means, sigma, and an input matrix X. Your function should classifiy all the rows of X.

```
likelihood <- function(x, j, centroid, sigma2){
  l <- 0
  for(k in seq(1,K,1)){
    l <- l + omega[k] * dmvnorm(x, mean = c(centroid[k,1,j+1], centroid[k,2,j+1]), sigma = sqrt(sigma2)
  }
  return(l)
}

bayes_classification <- function(centroid, sigma2, pi, X){
  y_hats <- c()
  for(i in 1:nrow(X)){
    x <- X[i,]
    p_0 <- likelihood(x, 0, centroid, sigma2)*(1-pi)
    p_1 <- likelihood(x, 1, centroid, sigma2)*(pi)
    classification <- which.max(c(p_0, p_1)) - 1
    y_hats <- c(y_hats, classification)
  }
  return(y_hats)
}

get_accuracy <- function(y_hat, y){
  correct <- y_hat == y
  pct_correct = sum(correct)/length(correct)
  return(pct_correct)
}

y_hat_bayes <- bayes_classification(location_vectors, sigma2, pi, x_train)
get_accuracy(y_hat_bayes, y_train)
```

```
## [1] 0.7733333
```

(e) Write an evaluation function that takes as input your training and test data. In addition it should take as input a vector of values for k, the KNN neighborhood size parameter. Your function should

i. Estimate the Bayes error using the test data.
ii. Estimate the test error of a linear classifier fit by least squares.
iii. Estimate the test errors for KNN at each of the values of k (the R package class has a knn() function). Run your function using k = 1, 3, 5, 7, 9, 11, 13, 15.

```
evaluation <- function(y_train, x_train, y_test, x_test, ks){
```

```
  train <- data.frame(cbind(y_train,x_train))
  test <- data.frame(cbind(y_test,x_test))

  #Bayesian Error
  y_hat_bayes <- bayes_classification(location_vectors, sigma2, pi, x_test)
  bayes_error <- 1 - get_accuracy(y_hat_bayes, y_test)
  cat("Bayes error: ",bayes_error,"\n")

  #Linear Classifier Error
  linear_classifier <- glm(y_train ~ ., data = train, family = "binomial")
  y_hat_linear_p <- predict(linear_classifier, test, type = "response")
  y_hat_linear <- y_hat_linear_p
  y_hat_linear[y_hat_linear_p < .50] <- 0
  y_hat_linear[y_hat_linear_p >= .50] <- 1
  linear_error <- 1 - get_accuracy(y_hat_linear, y_test)
  cat("Linear classifier error: ",linear_error,"\n")

  #KNN Error
  for(k in ks){
    y_hat_knn <- knn(x_train, x_test, y_train, k = k, prob = FALSE)
    knn_error <- 1 - get_accuracy(y_hat_knn, y_test)
    cat("KNN",k, " classifier error: ",knn_error,"\n")
  }
}

ks <- c(1, 3, 5, 7, 9, 11, 13, 15)
evaluation(y_train, x_train, y_test, x_test, ks)
```

```
## Bayes error:  0.23755
## Linear classifier error:  0.2684
## KNN 1  classifier error:  0.3044
## KNN 3  classifier error:  0.27685
## KNN 5  classifier error:  0.26845
## KNN 7  classifier error:  0.26815
## KNN 9  classifier error:  0.2644
## KNN 11  classifier error:  0.25985
## KNN 13  classifier error:  0.2606
## KNN 15  classifier error:  0.25715
```

(f) Modify your function in the previous step to take two extra parameters: the number noise of noise variables, and sigma.noise. The idea is to add additional Gaussian noise columns to xtrain and xtest. Once again your function should produce the list of errors as before.

```
evaluation <- function(y_train, x_train, y_test, x_test, ks, num_noise_vars, sigma.noise){

  #Bayesian Error
  y_hat_bayes <- bayes_classification(location_vectors, sigma2, pi, x_test)
  bayes_error <- 1 - get_accuracy(y_hat_bayes, y_test)
  cat("Bayes error: ",bayes_error,"\n")

  #Add noise
  noise_train <- matrix( rnorm(train_size*num_noise_vars,mean=0,sd=sigma.noise), train_size, num_noise_
  noise_test <- matrix( rnorm(test_size*num_noise_vars,mean=0,sd=sigma.noise), test_size, num_noise_vars

  x_train <- cbind(x_train, noise_train)
```

```
  x_test <- cbind(x_test, noise_test)

  train <- data.frame(cbind(y_train,x_train))
  test <- data.frame(cbind(y_test,x_test))

  #Linear Classifier Error
  linear_classifier <- glm(y_train ~ ., data = train, family = "binomial")
  y_hat_linear_p <- predict(linear_classifier, test, type = "response")
  y_hat_linear <- y_hat_linear_p
  y_hat_linear[y_hat_linear_p < .50] <- 0
  y_hat_linear[y_hat_linear_p >= .50] <- 1
  linear_error <- 1 - get_accuracy(y_hat_linear, y_test)
  cat("Linear classifier error: ",linear_error,"\n")

  #KNN Error
  for(k in ks){
    y_hat_knn <- knn(x_train, x_test, y_train, k = k, prob = FALSE)
    knn_error <- 1 - get_accuracy(y_hat_knn, y_test)
    cat("KNN",k, " classifier error: ",knn_error,"\n")
  }
}
```

(g) Run your function with 1,2,...,10 noise variables, with `sigma.noise=1`. Summarize what you have learned.

Summary: Both methods decrease accuracy with additional noise variables, but linear is more robust than KNN to additional noise variables.

```
num_noise_vars <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
for(n in num_noise_vars){
  cat(n,"noise variables\n")
  evaluation(y_train, x_train, y_test, x_test, ks, n, 1)
}
```

```
## 1 noise variables
## Bayes error:  0.23755
## Linear classifier error:  0.26905
## KNN 1  classifier error:  0.30635
## KNN 3  classifier error:  0.2882
## KNN 5  classifier error:  0.27995
## KNN 7  classifier error:  0.2746
## KNN 9  classifier error:  0.26605
## KNN 11  classifier error:  0.2629
## KNN 13  classifier error:  0.25855
## KNN 15  classifier error:  0.25455
## 2 noise variables
## Bayes error:  0.23755
## Linear classifier error:  0.2712
## KNN 1  classifier error:  0.3134
## KNN 3  classifier error:  0.2929
## KNN 5  classifier error:  0.2831
## KNN 7  classifier error:  0.27575
## KNN 9  classifier error:  0.27375
## KNN 11  classifier error:  0.2675
## KNN 13  classifier error:  0.26695
```

```
## KNN 15  classifier error:  0.26455
## 3 noise variables
## Bayes error:  0.23755
## Linear classifier error:  0.2761
## KNN 1  classifier error:  0.3184
## KNN 3  classifier error:  0.29625
## KNN 5  classifier error:  0.2832
## KNN 7  classifier error:  0.2769
## KNN 9  classifier error:  0.2781
## KNN 11  classifier error:  0.27565
## KNN 13  classifier error:  0.27445
## KNN 15  classifier error:  0.27205
## 4 noise variables
## Bayes error:  0.23755
## Linear classifier error:  0.2733
## KNN 1  classifier error:  0.3385
## KNN 3  classifier error:  0.3127
## KNN 5  classifier error:  0.30075
## KNN 7  classifier error:  0.2884
## KNN 9  classifier error:  0.2822
## KNN 11  classifier error:  0.28185
## KNN 13  classifier error:  0.27795
## KNN 15  classifier error:  0.2766
## 5 noise variables
## Bayes error:  0.23755
## Linear classifier error:  0.2804
## KNN 1  classifier error:  0.3359
## KNN 3  classifier error:  0.3114
## KNN 5  classifier error:  0.3031
## KNN 7  classifier error:  0.2959
## KNN 9  classifier error:  0.2898
## KNN 11  classifier error:  0.28685
## KNN 13  classifier error:  0.28305
## KNN 15  classifier error:  0.2825
## 6 noise variables
## Bayes error:  0.23755
## Linear classifier error:  0.27595
## KNN 1  classifier error:  0.3497
## KNN 3  classifier error:  0.3301
## KNN 5  classifier error:  0.31725
## KNN 7  classifier error:  0.31485
## KNN 9  classifier error:  0.31045
## KNN 11  classifier error:  0.31115
## KNN 13  classifier error:  0.30855
## KNN 15  classifier error:  0.30615
## 7 noise variables
## Bayes error:  0.23755
## Linear classifier error:  0.28
## KNN 1  classifier error:  0.3469
## KNN 3  classifier error:  0.32255
## KNN 5  classifier error:  0.30795
## KNN 7  classifier error:  0.30095
## KNN 9  classifier error:  0.2987
## KNN 11  classifier error:  0.29575
```

```
## KNN 13  classifier error:  0.2938
## KNN 15  classifier error:  0.2904
## 8 noise variables
## Bayes error:  0.23755
## Linear classifier error:  0.27165
## KNN 1  classifier error:  0.35555
## KNN 3  classifier error:  0.32535
## KNN 5  classifier error:  0.3101
## KNN 7  classifier error:  0.30295
## KNN 9  classifier error:  0.30075
## KNN 11  classifier error:  0.2954
## KNN 13  classifier error:  0.293
## KNN 15  classifier error:  0.2896
## 9 noise variables
## Bayes error:  0.23755
## Linear classifier error:  0.2738
## KNN 1  classifier error:  0.3658
## KNN 3  classifier error:  0.3403
## KNN 5  classifier error:  0.3195
## KNN 7  classifier error:  0.31515
## KNN 9  classifier error:  0.3102
## KNN 11  classifier error:  0.30595
## KNN 13  classifier error:  0.29885
## KNN 15  classifier error:  0.29385
## 10 noise variables
## Bayes error:  0.23755
## Linear classifier error:  0.28085
## KNN 1  classifier error:  0.365
## KNN 3  classifier error:  0.34375
## KNN 5  classifier error:  0.32965
## KNN 7  classifier error:  0.32145
## KNN 9  classifier error:  0.3139
## KNN 11  classifier error:  0.3105
## KNN 13  classifier error:  0.3068
## KNN 15  classifier error:  0.30325
```

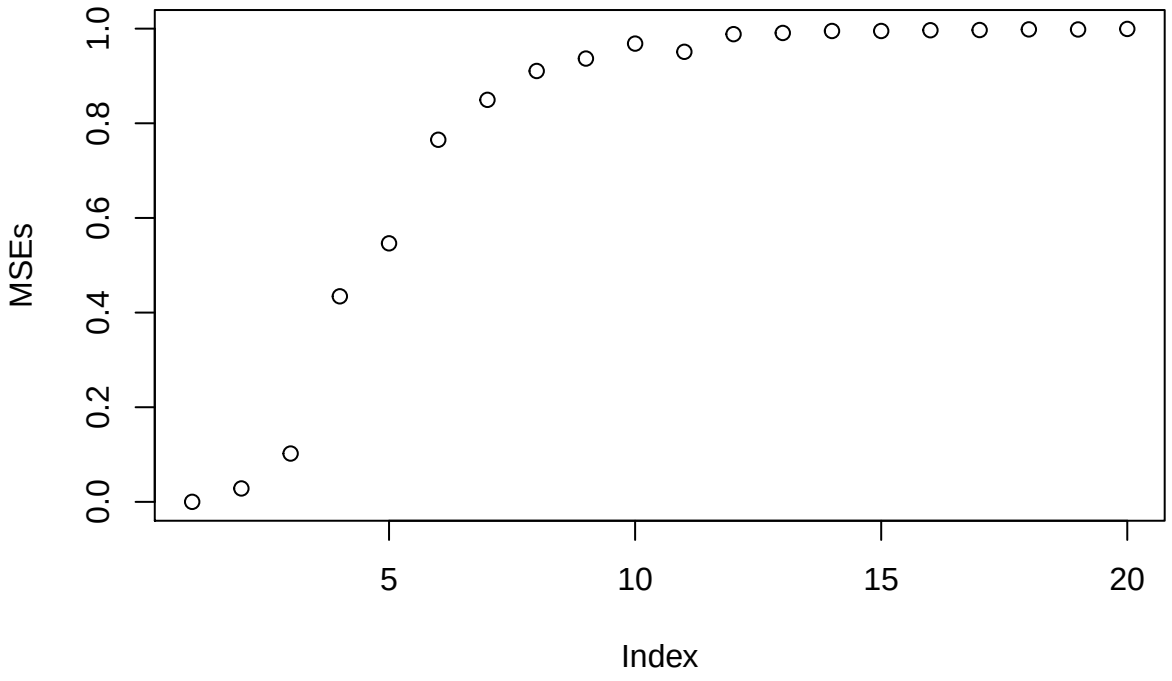
Question 2: Curse of Dimensionality

(c) Suppose we have 1000 training examples x_i generated uniformly inside the unit cube in p dimensions. The true relation between X and Y is $Y = e^{-8||X||_*^2}$ where, $||X||_*$ is the 2-norm of the first $floor(p/2)$ entries of the vector X.

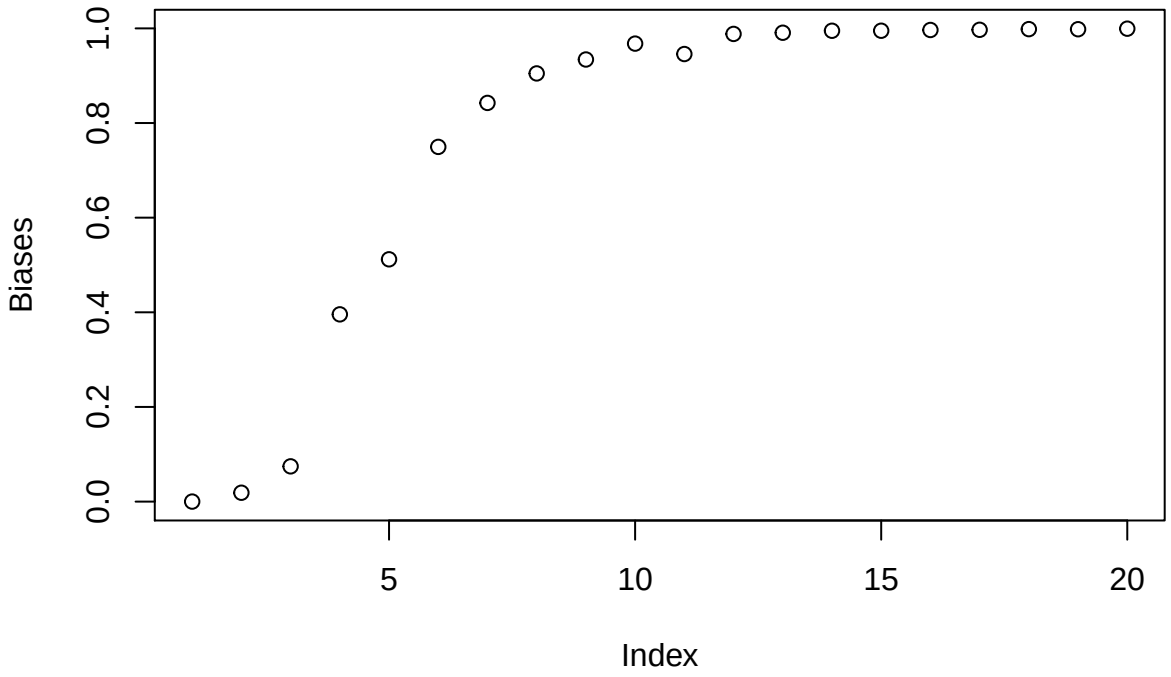
We use the 1-nearest neighbour rule to predict y_0 at the test-point $x_0 = 0$. Plot the MSE, squared-bias and variance as a function of p, for $p = 1, ..., 20$. (See Fig. 2.7 from ESL). Use at least 100 simulations if you are estimating the above quantities empirically. Explain the behavior of the variance as p increases?

```
rm(list = ls())
num_reps <- 100
n <- 1000
ps <- seq(1,20,1)

E_y_hat_0s <- c()
MSEs <- c()
Biases <- c()
```

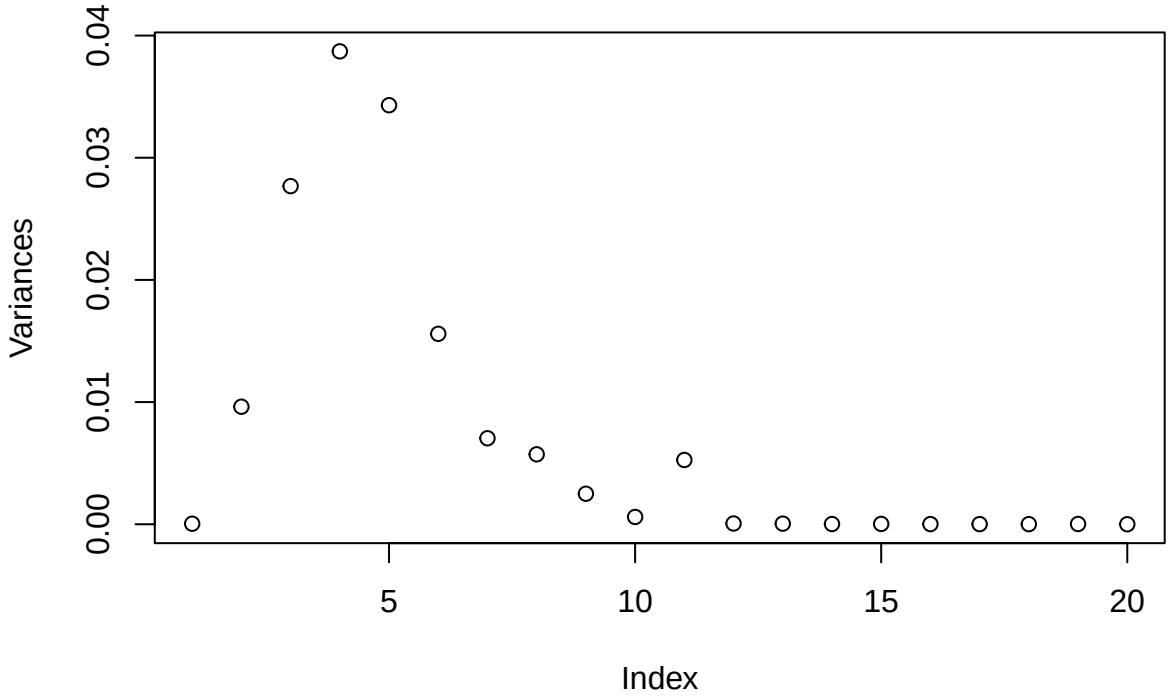
```r
Variances <- c()
for(p in ps){

  half <- floor(p/2)
  f_x0 <- exp(-8 * norm(rep(0,p)[1:half], type="2"))
  y_hat_0s <- c()

  for(rep in 1:num_reps){
    X <- matrix( runif(n*p), n, p)
    Y <- apply(X, 1, function(x) exp(-8 * norm(x[1:half], type="2")))
    y_hat_0_lvl <- knn(X, rep(0,p), Y, k=1, prob = FALSE)

    y_hat_0 <- as.numeric(levels(y_hat_0_lvl))[y_hat_0_lvl]
    y_hat_0s <- c(y_hat_0s, y_hat_0)
  }
  MSE <- sum((f_x0 - y_hat_0s)^2)/num_reps
  E_y_hat_0 <- sum(y_hat_0s)/num_reps
  Var_T <- sum((y_hat_0s - E_y_hat_0)^2)/num_reps
  Bias_sq <- (E_y_hat_0 - f_x0)^2

  E_y_hat_0s <- c(E_y_hat_0s, E_y_hat_0)
  MSEs <- c(MSEs, MSE)
  Biases <- c(Biases, Bias_sq)
  Variances <- c(Variances, Var_T)
}

plot(MSEs)
```




```r
plot(Biases)
```



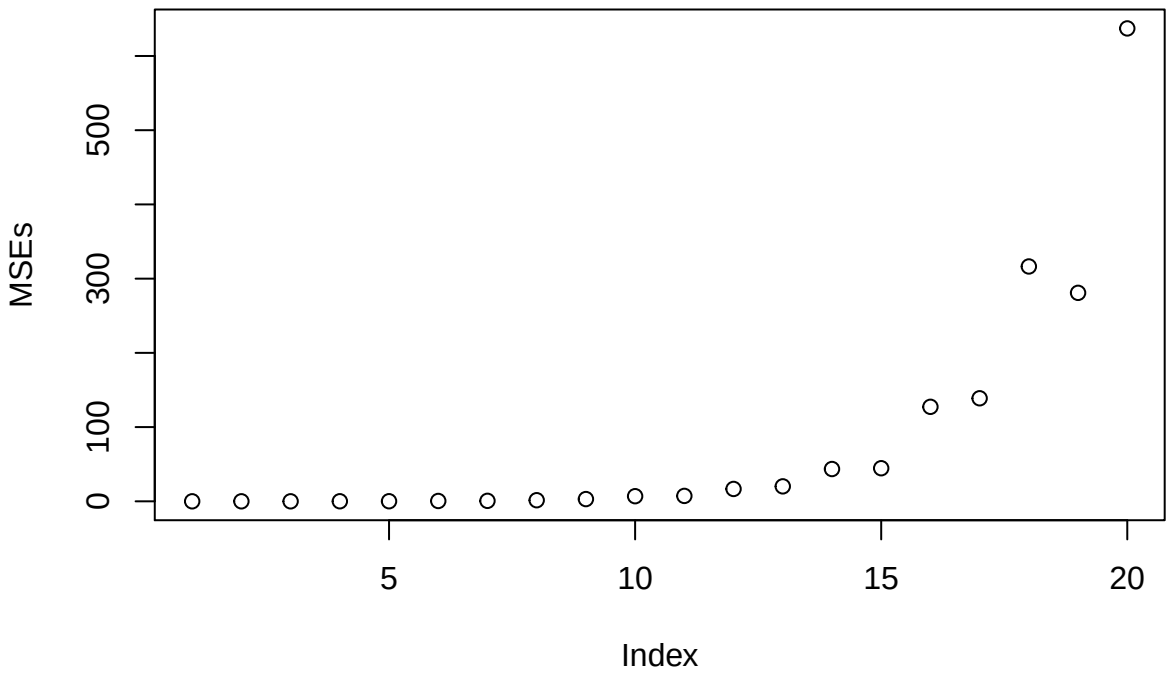

```
plot(Variances)
```



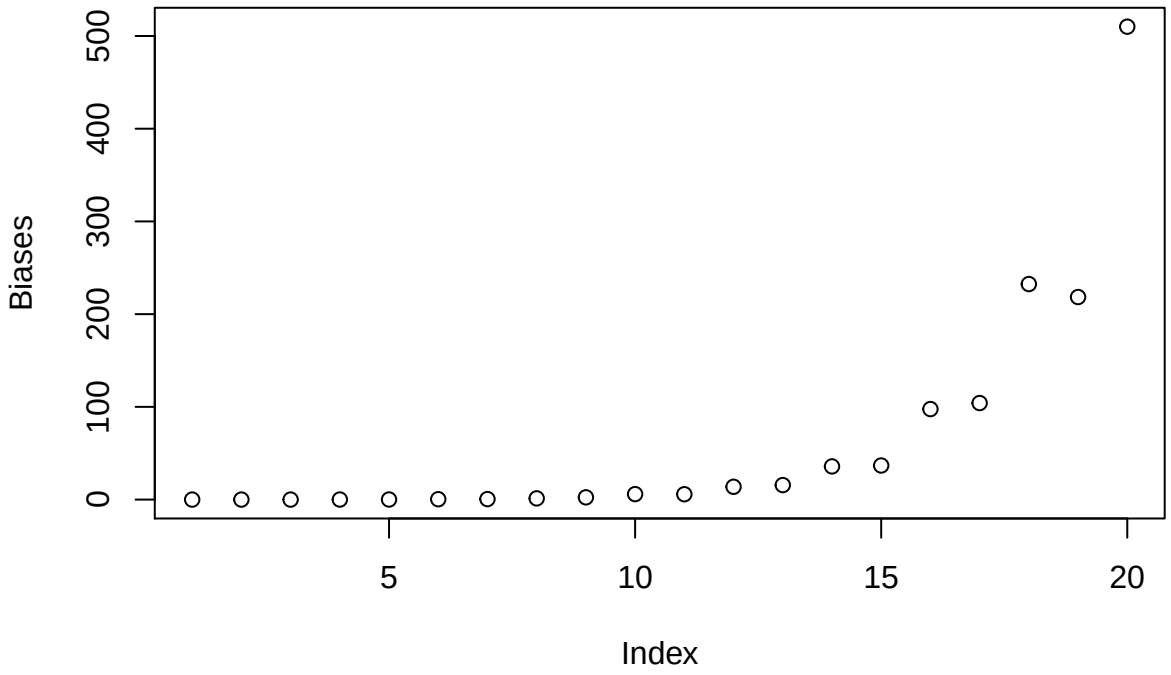
(d) Repeat part (c) with $Y = e^{f(X)}$, where $f(X)$ is the sum of the first $floor(p/2)$ entries of the vector X.

```
E_y_hat_0s <- c()
MSEs <- c()
Biases <- c()
Variances <- c()
for(p in ps){

  half <- floor(p/2)
```

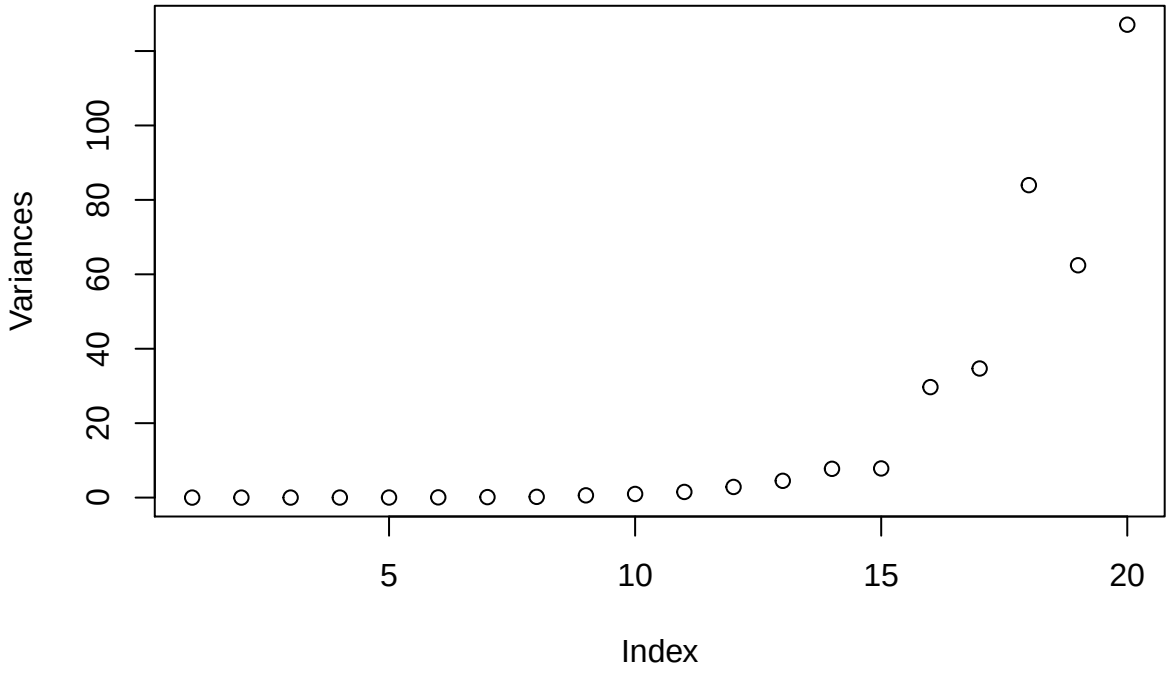
```
  f_x0 <- exp(sum(rep(0,p)[1:half]))
  y_hat_0s <- c()

  for(rep in 1:num_reps){
    X <- matrix( runif(n*p), n, p)
    Y <- apply(X, 1, function(x) exp(sum(x[1:half])))
    y_hat_0_lvl <- knn(X, rep(0,p), Y, k=1, prob = FALSE)

    y_hat_0 <- as.numeric(levels(y_hat_0_lvl))[y_hat_0_lvl]
    y_hat_0s <- c(y_hat_0s, y_hat_0)
  }
  MSE <- sum((f_x0 - y_hat_0s)^2)/num_reps
  E_y_hat_0 <- sum(y_hat_0s)/num_reps
  Var_T <- sum((y_hat_0s - E_y_hat_0)^2)/num_reps
  Bias_sq <- (E_y_hat_0 - f_x0)^2

  E_y_hat_0s <- c(E_y_hat_0s, E_y_hat_0)
  MSEs <- c(MSEs, MSE)
  Biases <- c(Biases, Bias_sq)
  Variances <- c(Variances, Var_T)
}

plot(MSEs)
```




```
plot(Biases)
```




```
plot(Variances)
```



Question 3: Bootstrap

Frequentist inference goes as follows. We have a random sample of N observations from a population distribution F, and we compute some statistic of interest (lets assume a scalar). The sampling distribution of that statistic is what we would get if we were able to repeat the process a large number of times: drawing samples of size N from F and computing the statistic each time. From this we could compute a quantity of interest, such as the standard deviation (standard error), which gives us an idea of the precision of our statistic. Many of our favorite models have convenient assumptions which allow us to derive theoretical

expressions for say, the standard error, which can then be computed for our one and only sample. Sometimes these calculations are intractable.

The *bootstrap*, invented by our own Prof Brad Efron around 1980, is a cunning device for approximating the sampling distribution. Let $\hat{F}$ be the empirical distribution of our sample (the distribution that puts mass $1/N$ on each of the observed samples). We now draw a sample of size N from $\hat{F}$. This amounts to drawing from the original N observations a sample of size N with replacement. We compute our statistic on this bootstrap sample. This process is repeated a large number B times, and we use the bootstrap distribution of our statistic to approximate its sampling distribution.

(a) Assume the model $y = f(x) + \epsilon$ with $x \sim U[0, 1]$ and $\epsilon \sim N(0, (0.2)^2)$. For us $f(x) = 1 + x^2$. We have a sample of size 50 from this model. We fit a polynomial regression of degree 2 (we don't know the true model, so we include a linear term as well as a quadratic term), and build a prediction function $f(x)$ from the fitted model. We are interested in estimating $x_0 = \hat{f}^{-1}(1.3)$. (This kind of thing is done in dose-response experiments, and we are trying to estimate the dose that achieves a desired output). Write a function for taking a sample and delivering an estimate for $\hat{x}_0$.

```
rm(list = ls())
generate_model_from_pop <- function(n){
  X <- runif(n,0,1)
  X2 <- X^2
  y_noiseless <- 1+X2

  eps <- rnorm(n, 0, sqrt(0.2^2))
  y <- y_noiseless + eps
  return(lm(y ~ X + X2))
}

generate_model_from_boostrap <- function(n, X){
  X_bs <- sample(X, n, replace = TRUE)
  X2 <- X_bs^2
  y_noiseless <- 1+X2

  eps <- rnorm(n, 0, sqrt(0.2^2))
  y <- y_noiseless + eps
  return(lm(y ~ X_bs + X2))
}

f_hat <- function(x, model){
  features <- c(1, x, x^2)
  y_hat <- features %*% model$coefficients
  return(y_hat)
}

f_hat_inverse <- function(y, model){
  a <- model$coefficients[3]
  b <- model$coefficients[2]
  intcp <- model$coefficients[1]
  c <- intcp - y

  x_hat <- (-b + sqrt(b^2 - 4*a*c)) / (2*a)

  return(x_hat)
}
```

```r
model <- generate_model_from_pop(50)
x_hat_0 <- f_hat_inverse(1.3, model)
x_hat_0
```

```
##         X
## 0.5114531
```

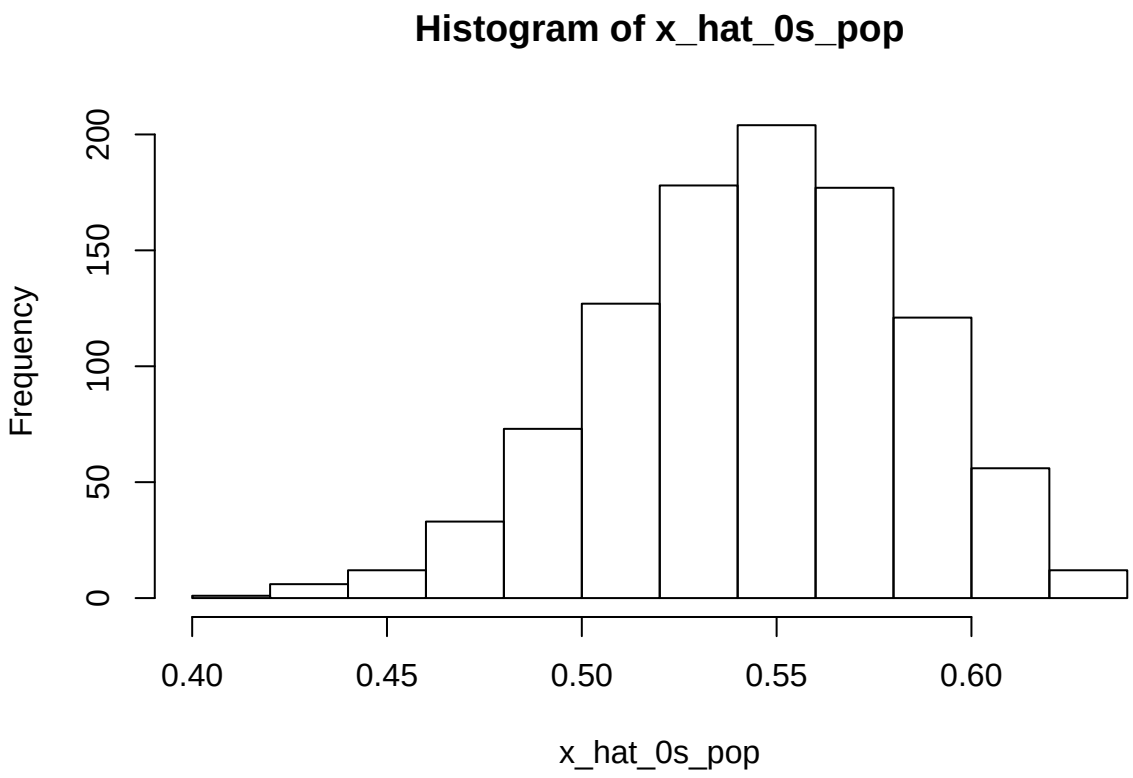
(b) Generate a sample from this model, and compute $\hat{x}_0$ for this sample.

```r
model <- generate_model_from_pop(50)
x_hat_0 <- f_hat_inverse(1.3, model)
cat("Sample value: ", x_hat_0)
```

```
## Sample value:  0.5496084
```

(c) Simulate a 1000 realizations from this model, to approximate the sampling distribution of $\hat{x}_0$.

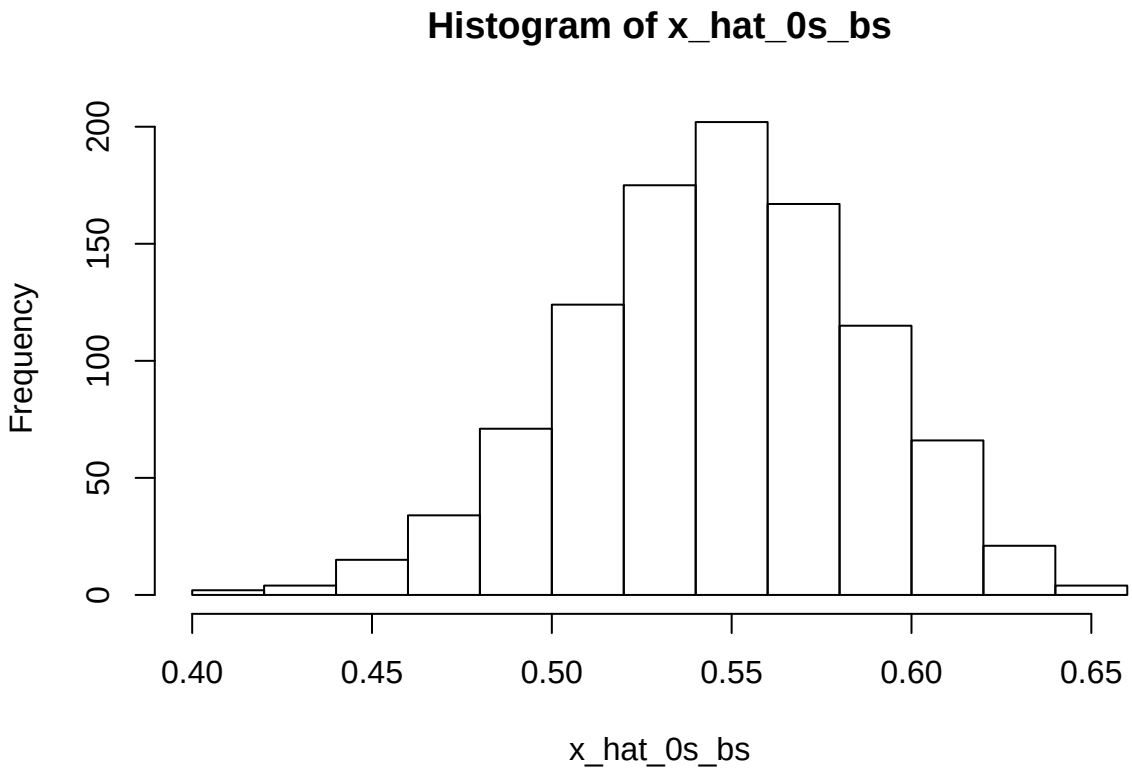
```r
realizations <- 1000
x_hat_0s_pop <- c()
for(i in 1:realizations){
  model <- generate_model_from_pop(50)
  x_hat_0s_pop <- c(x_hat_0s_pop, f_hat_inverse(1.3, model))
}

hist(x_hat_0s_pop)
```



(d) Using just your original sample, compute the bootstrap distribution of $\hat{x}_0$.

```r
X <- runif(50,0,1)
x_hat_0s_bs <- c()
for(i in 1:realizations){
  model <- generate_model_from_boostrap(50, X)
```

```
  x_hat_0s_bs <- c(x_hat_0s_bs, f_hat_inverse(1.3, model))
}

hist(x_hat_0s_bs)
```



(e) Compare these two distributions, and in particular their standard deviations.

```
cat("STD resampling from population: ", sd(x_hat_0s_pop))
```

```
## STD resampling from population:  0.03827449
```

```
cat("STD resampling from bootstrap: ", sd(x_hat_0s_bs))
```

```
## STD resampling from bootstrap:  0.04033741
```