Assignment 2

Nathan Quinn 1512005, Edward Burn

1 (a) Show that the unique stationary point is obtained by choosing for each $k \in \{1, \ldots, K\}$

$$\boldsymbol{\mu}_k = \frac{\sum_{i=1}^{n} \gamma_{ik} \boldsymbol{x}_i}{\sum_{i=1}^{n} \gamma_{ik}}.$$

Maximising the function for a fixed $\boldsymbol{\mu}_{kj}$:

$$f(\boldsymbol{\mu}_{1:K}) = f(\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K) = \sum_{i=1}^{n} \sum_{k=1}^{K} \gamma_{ik} \sum_{j=1}^{p} (x_{ij} \log(\mu_{kj}) + (1 - x_{ij}) \log(1 - \mu_{kj}))$$

Need to find the partial derivative to f with respect to μ_{kj} and set to 0 to find maximum of f for fixed μ_{kj}.
Hence:

$$\frac{\partial f}{\partial \mu_{kj}} = \sum_{i=1}^{n} \gamma_{ik} \left(\frac{x_{ij}}{\mu_{kj}} - \frac{1 - x_{ij}}{1 - \mu_{kj}} \right) = 0$$

Hence:

$$\frac{\sum_{i=1}^{n} x_{ij} \gamma_{ik}}{\mu_{kj}} = \frac{\sum_{i=1}^{n} \gamma_{ik} (1 - x_{ij})}{1 - \mu_{kj}}$$

Hence:

$$(1 - \mu_{kj}) \sum_{i=1}^{n} \gamma_{ik} x_{ij} = \mu_{kj} \sum_{i=1}^{n} \gamma_{ik} (1 - x_{ij})$$

Hence:

$$\sum_{i=1}^{n} \gamma_{ik} x_{ij} - \mu_{kj} \sum_{i=1}^{n} \gamma_{ik} x_{ij} = \mu_{kj} \sum_{i=1}^{n} \gamma_{ik} - \mu_{kj} \sum_{i=1}^{n} \gamma_{ik} x_{ij}$$

Hence:

$$\mu_{kj} \sum_{i=1}^{n} \gamma_{ik} = \sum_{i=1}^{n} \gamma_{ik} x_{ij}$$

Hence:

$$\hat{\mu}_{kj} = \frac{\sum_{i=1}^{n} \gamma_{ik} x_{ij}}{\sum_{i=1}^{n} \gamma_{ik}}$$

As j was arbitrary this implies:

$$\hat{\mu}_{k} = \frac{\sum_{i=1}^{n} \gamma_{ik} x_{i}}{\sum_{i=1}^{n} \gamma_{ik}}$$

As required.

(b)

```
#Define likelihood function for bernoulli rv
library(Rlab)
```

```
## Rlab 2.15.1 attached.
```

```
##
## Attaching package: 'Rlab'
```

```
## The following objects are masked from 'package:stats':
##
##     dexp, dgamma, dweibull, pexp, pgamma, pweibull, qexp, qgamma,
##     qweibull, rexp, rgamma, rweibull
```

```
## The following object is masked from 'package:datasets':
##
##     precip
```

```
logsumexp <- function(x) return(log(sum(exp(x - max(x)))) + max(x))


compute_ll.bern <- function(xs,mus,lws,gammas) {
  ll <- 0
  n <- dim(xs)[1]
  K <- dim(mus)[1]
  for (i in 1:n) {
    for (k in 1:K) {
      if (gammas[i,k] > 0) {
        ll <- ll + gammas[i,k]*(lws[k]+sum(dbinom(size=1,xs[i,],mus[k,],log=TRUE))-log(gammas[i,k]))
      }
    }
  }
  return(ll)
}



em_mix_bern <- function(xs,K,max.numit=Inf) {
  p <- dim(xs)[2] #because each x is multivariate, it has p dimensions to it.
  n <- dim(xs)[1] #simply the number of xs in our dataset

  # lws is log(ws)
  # we work with logs to keep the numbers stable
  # start off with ws all equal
  lws <- rep(log(1/K),K)

  #We cannot sample from xs as we did in Gaussian because mus will be either 0 or 1,
  #this will cause the likelihood function to be 0 (by definition of likelihood)
  #Hence we will start with all mus samples from a uniform dist on [0.1,0.9]

  mus <- matrix(data=runif(max=0.9,min=0.1, n= K*p),nrow = K,ncol = p)

  # gammas will be set in the first iteration
  gammas <- matrix(0,n,K)

  converged <- FALSE
  numit <- 0
  ll <- -Inf
  print("iteration : log-likelihood")
  while(!converged && numit < max.numit) {
    numit <- numit + 1
    mus.old <- mus
    ll.old <- ll

    ## E step - calculate gammas
    for (i in 1:n) {
      # the elements of lprs are log(w_k * p_k(x)) for each k in {1,...K}
      lprs <- rep(0,K)
```

```
      for (k in 1:K) {
        lprs[k] <- lws[k] + sum(dbern(xs[i,], prob = mus[k,],log = TRUE))
      }
      # gammas[i,k] = w_k * p_k(x) / sum_j {w_j * p_j(x)}
      gammas[i,] <- exp(lprs - logsumexp(lprs))
    }

    ll <-compute_ll.bern(xs,mus,lws,gammas)


    # M step - update ws and mus
    Ns <- rep(0,K)

    for (k in 1:K) {
      Ns[k] <- sum(gammas[,k])
      lws[k] <- log(Ns[k])-log(n)

      mus[k,] <- rep(0,p)

      for (i in 1:n) {

        mus[k,] <- mus[k,]+gammas[i,k]/Ns[k]*xs[i,]
      }
      mus[which(mus > 1,arr.ind=TRUE)] <- 1 - 1e-15

    }
    print(paste(numit,": ",ll))
    # we stop once the increase in the log-likelihood is "small enough"
    if (abs(ll-ll.old) < 1e-5) converged <- TRUE

    }

  return(list(lws=lws,mus=mus,gammas=gammas,ll=ll))
}

#Test on small dataset
n <- 500; p <- 50
K.actual <- 2
mix <- runif(K.actual); mix <- mix / sum(mix)
mus.actual <- matrix(runif(K.actual*p),K.actual,p)
zs.actual <- rep(0,n)
xs <- matrix(0,n,p)
for (i in 1:n) {
  cl <- sample(K.actual,size=1,prob=mix)
  zs.actual[i] <- cl
  xs[i,] <- (runif(p) < mus.actual[cl,])
}


print(system.time(out <- em_mix_bern(xs,K.actual)))
```

```
## [1] "iteration : log-likelihood"
## [1] "1 :  -20938.2589678005"
## [1] "2 :  -13672.4437785488"
```

```
## [1] "3 :  -12226.9027015845"
## [1] "4 :  -12225.2003511647"
## [1] "5 :  -12225.2003455662"
##    user  system elapsed
##    0.12    0.01    0.15
```

```
v1 <- sum(abs(out$mus-mus.actual))
v2 <- sum(abs(out$mus[2:1,]-mus.actual))
vm <- min(v1,v2)/p/2
print(vm)
```

```
## [1] 0.01910418
```

```
if (vm > .3) print("probably not working") else print("might be working")
```

```
## [1] "might be working"
```

```
#Takes a ridiculously long time to load so eval=false is used for now.

load("20newsgroups.rdata")

em_mix_bern(xs = documents, 4)
```

```
sum(newsgroups.onehot[,1])/nrow(newsgroups.onehot) #0.2835242
sum(newsgroups.onehot[,2])/nrow(newsgroups.onehot) #0.2166605
sum(newsgroups.onehot[,3])/nrow(newsgroups.onehot) #0.1635882
sum(newsgroups.onehot[,4])/nrow(newsgroups.onehot) #0.3362271
```

Values obtained are:
-235926.4 = ll
[1] -2.228653 -2.551953 -0.642121 -1.244024
exp(-2.228653) #0.10767
exp(-2.551953) #0.07792
exp(-0.642121) #0.52617
exp(-1.244024) #0.28822

(b) (ii) The weights associated with each cluster are defined above. One cluster has over 50% probability associated with it, indicating that a majority of documents in the dataset are similar enough to be clustered together. The next largest is 28.8% , while the final two are only 10.7% and 7.79%. The clusters are not labelled however, so we do not know which topic of discussion belongs to which cluster.

Since the weights are fractions associated with each topic of discussion, the accuracy of the algorithm could be measured by working out the fraction of documents associated with each topic, and comparing this true fraction to the one obtained by our algorithm. However, this would rely on the assumption that the clustering the algorithm is doing is indeed the clustering you 'want' it to do, and thus will be comparable to the 'true' fractions.

For example, if the true fraction of posts containing sports was 50%, and the algorithm returned a cluster weight of 50%, then although it would look like the algorithm performed well, it may have found a different cluster you were unaware of.

Since we do not know the labels of the dataset, it is difficult if not impossible to measure the accuracy of the algorithm. If the newsgroups.onehot is the true allocations then it would appear the algorithm isn't extremely accurate for this dataset, however without labels it is impossible to know for sure.

2 (a) Implement both Thompson sampling and the ϵ-decreasing strategy in this setting with the unknown success probabilities of the arms being 0.6 and 0.4.

```
ps <- c(0.6,0.4)
epsilon.decreasing <- function(ps,epsilon,n) {
  realisedRegret=c()
  realisedRegretOverLogI=c()
  as <- rep(0,n)
  rs <- rep(0,n)


  ns <- rep(0,2); ss <- rep(0,2)

  for (i in 1:2) {
    a <- i
    r <- runif(1) < ps[a]
    ns[a] <- ns[a] + 1
    ss[a] <- ss[a] + r
    as[i] <- a
    rs[i] <- r
  }
 y=c()
  for (i in 3:n) {
    if (runif(1) < epsilon[i]) {
      a <- sample(2,1)
    } else {
      a <- which.max(ss/ns)
    }

    r <- runif(1) < ps[a]


    ns[a] <- ns[a] + 1
    ss[a] <- ss[a] + r
    as[i] <- a
    rs[i] <- r
    realisedRegret[i] = ps[1]*i - sum(rs)
    realisedRegretOverLogI[i] = realisedRegret[i]/log(i)
    y[i]=sum(rs)/i
  }

  return(list(as=as,rs=rs,y=y,realisedRegretOverLogI=realisedRegretOverLogI))

}

sample_arm.bernoulli  <- function(ns,ss) {

  m1 = rbeta(1,1+ss[1], 1+(ns[1]-ss[1]))
  m2 = rbeta(1,1+ss[2], 1+(ns[2]-ss[2]))
  if(m1>m2){
    return(1)
  }
  else{
    return(2)
```

```
  }


}

thompson.bernoulli <- function(ps,n) {
  realisedRegret=c()
  realisedRegretOverLogI=c()
  as <- rep(0,n)
  rs <- rep(0,n)
  ns <- rep(0,2); ss <- rep(0,2)
  y=c()
  for (i in 1:n) {
    a <- sample_arm.bernoulli(ns,ss)
    r <- runif(1) < ps[a]
    ns[a] <- ns[a] + 1
    ss[a] <- ss[a] + r
    as[i] <- a
    rs[i] <- r
    realisedRegret[i] = ps[1]*i - sum(rs)
    realisedRegretOverLogI[i] = realisedRegret[i]/log(i)
    y[i]=sum(rs)/i

  }

  return(list(as=as,rs=rs,y=y,realisedRegretOverLogI=realisedRegretOverLogI))
}
 v=1:100000
 epsilon1=c()
 epsilon2=c()
 for (i in 1:100000){
  epsilon1[i] = min(1,(100/v[i]))
 }
 for (i in 1:100000){
  epsilon2[i] = min(1,(100/(v[i])^2))
 }
 thompson.bernoulli.out <- thompson.bernoulli(ps=ps,n=100000)
 epsilonDecreasing1 = epsilon.decreasing(ps,epsilon1,100000)
  epsilonDecreasing2 = epsilon.decreasing(ps,epsilon2,100000)
    print(sum(thompson.bernoulli.out$rs))
```

```
## [1] 59994
```

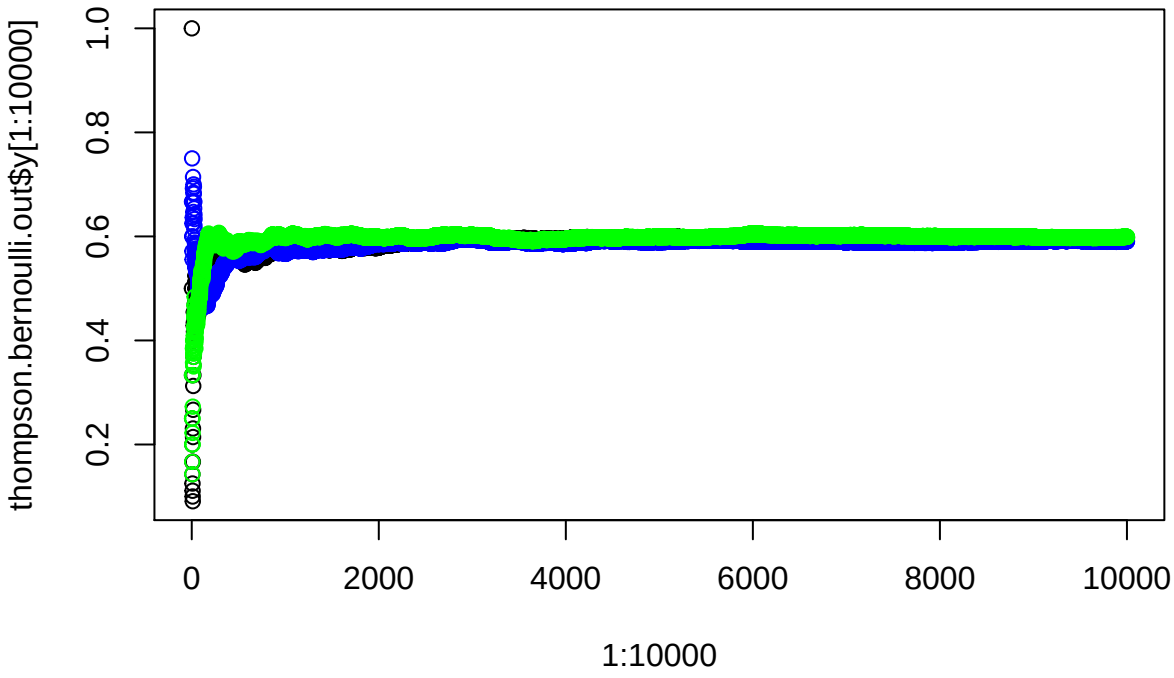
```
    print(sum(epsilonDecreasing1$rs))
```

```
## [1] 59836
```

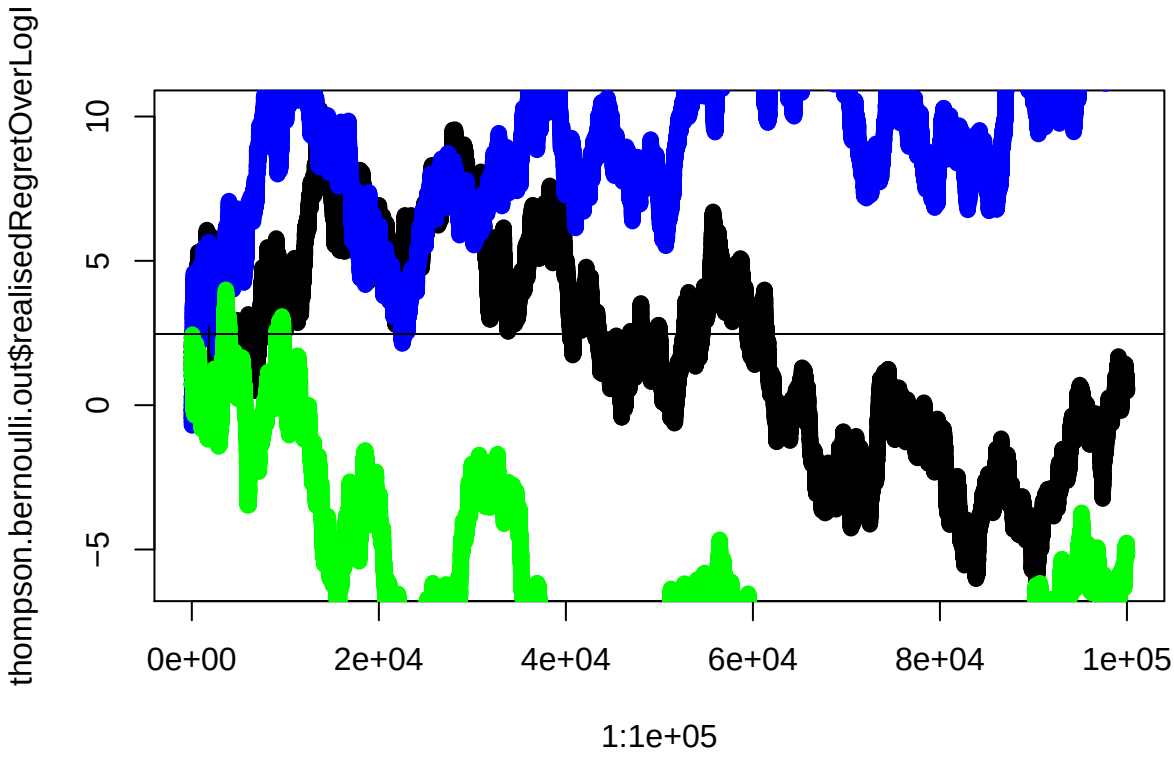
```
    print(sum(epsilonDecreasing2$rs))
```

```
## [1] 60058
```

```
    plot(1:10000,thompson.bernoulli.out$y[1:10000])
    points(1:10000,epsilonDecreasing1$y[1:10000],col= "blue")
    points(1:10000,epsilonDecreasing2$y[1:10000],col= "green")
```




```
plot(1:100000,thompson.bernoulli.out$realisedRegretOverLogI)
points(1:100000,epsilonDecreasing1$realisedRegretOverLogI,col= "blue")
points(1:100000,epsilonDecreasing2$realisedRegretOverLogI,col= "green")
abline((ps[1]-ps[2])/(ps[2]*log(ps[2]/ps[1])+(1-ps[2])*log((1-ps[2])/(1-ps[1]))),0)
```



(b) As the number of trials increase the probability you play an arm at random will go down linearly and conversly the probability you play the best arm so far goes up linealy. It turns out by the Bernouli lemmas that the probability of picking at random goes down slowly enough means your expected winnings will reach asymptotic optimality, which is your expected reward given you know the probability distributions of each arm, so you will just keep playing the best arm. The higher the constant C, the more likely you are to play at random for a given nth trial, since C can be greater than 1, when c/i, where i is trial i is greater than 1, you will play an arm at random, so it also controls for how long you choose an arm at random.

(c) The constant acts in the same way as for (b), but the probability of playing an arm at random as you go through the trials will decrease in proportion to trials squared.It turns out by the Bernouli lemmas that the probability of picking at random does not go down slowly enough for your expected winnings to reach asymptotic optimality.

(d) From the first graph you can see that all three strategies reach asymptotic optimality very quickly and at about the same rate. When looking at many different samples occasionally epsilon2 is clearly less than 0.6 the average optimal reward. This is because theoretically epsilon2 decreasing should not be asymptotically optimal, but since the rate at which epsilon2 decreases is at the boundary of where this strategy becomes sub-optimal it is still very likely this strategy ends up at the optimal expected winnings. From the second graph, the realised regret over log(n) is as expected mostly bellow the constant (µ1-µ2)/DKL(µ2||µ1) for Thopson sampling. For the two epsilon decreasing strategies the realised regret/log(n) is mainly above the graph) and from looking at many samples the epsilon2 strategy has a realised regret/log(n) much higher than the other two on average and for epsilon1 regret/log(n) ends up higher than for Thompson. This shows that Thompson sampling seems to be the best strategy on average for p=(0.6,0.4).

3 (a)

```
library(doBy) #provides us with the which.minn function to return indices of
              #n smallest numbers in a vector. It also considers two distances
              #being equal.

#we include the variable distances so we can decide each time on a distance function
knn.regression.test <- function(k,train.X,train.Y,test.X,test.Y,distances) {

  #We do this by finding the k nearest neighbours, and averaging them
  n <- dim(train.X)[1]
  p <- dim(test.X)[1]
  distance <- matrix(0, nrow = p, ncol = n)
  estimates <- c()
  for(i in 1:p){
    for(j in 1:n){
      #distance will be a matrix of distances between each testing X (the rows)
      #and each training X (the columns)
      distance[i,j] <- distances(test.X[i,], train.X[j,])

      }
  }

   distance.index <- matrix(0, nrow = n, ncol = k)

   for(i in 1:p){
     #which.minn will return the INDICES of each row, important for assigning Y values later
     #distance.index will contain which i (in 1 to n) the k closest X values correspond to
     distance.index[i,] <- which.minn(distance[i,], k)
   }

  for(i in 1:p){
    for(j in 1:k){
      #now make each element of the matrix correspond to the assignment of each element
      distance.index[i,j]<- as.vector(train.Y)[distance.index[i,j]]
    }
  }

   for(i in 1:p){
    estimates[i] <- sum(distance.index[i,])/k
  }
 print(sum((test.Y-estimates)^2))


}

#########################################################################################################

#Define l1 and l2 norms.

distances.l1 <- function(x,y){
   l1 <- 0
   for(i in 1:length(x)){
     l1 <- l1 + (abs(x[i]-y[i]))
   }
```

```
   return(l1)
}


distances.l2 <- function(x,y){
   l2 <- 0
   for(i in 1:length(x)){
     l2 <- l2 + ((abs((x[i]-y[i])))^2)
   }
   return(l2)
}
```

(b) Test your function on the two toy datasets. Try different values of k and report your results.

Toy dataset 1:

```
n=100
train.X <- matrix(sort(rnorm(n)),n,1)
train.Y <- (train.X < -0.5) + train.X*(train.X>0)+rnorm(n,sd=0.03)
plot(train.X,train.Y)
test.X <- matrix(sort(rnorm(n)),n,1)
test.Y <- (test.X < -0.5) + test.X*(test.X>0)+rnorm(n,sd=0.03)

k=3
knn.regression.test(k,train.X,train.Y,test.X,test.Y,distances.l1)


#k=2 1.161032
#k=3 1.729457
#k=4 2.193075
#k=5 2.663743
```

k=2 in this instance appears to be the most accurate value.

Toy dataset 2:

```
train.X=matrix(rnorm(200),100,2)
train.Y=train.X[,1]
test.X=matrix(rnorm(100),50,2)
test.Y=test.X[,1]

k=5
knn.regression.test(k,train.X,train.Y,test.X,test.Y,distances.l1)

#k=2 1.712251
#k=3 2.134198
#k=4 2.161132
#k=5 1.735976
```

(c) Load the Iowa dataset. Try to predict the yield in the years 1931, 1933, ... based on the data from 1930, 1932, ...

```
install.packages("lasso2")
library("lasso2")
data(Iowa)
train.X=as.matrix(Iowa[seq(1,33,2),1:9])
train.Y=c(Iowa[seq(1,33,2),10])
test.X=as.matrix(Iowa[seq(2,32,2),1:9])
test.Y=c(Iowa[seq(2,32,2),10])
k=5
knn.regression.test(k,train.X,train.Y,test.X,test.Y,distances.l2)

#Sum of squared errors is 1527.978.
```

(d) Compare the results with OLS regression and ridge regression.

```
library(leaps)
library(car)
library(MASS)

install.packages("lasso2")
library("lasso2")
data(Iowa)
attach(Iowa)




y <- as.data.frame(train.Y)
x <- as.data.frame(train.X)
xy <- cbind(x,y)
colnames(xy)[which(names(xy) == "train.Y")] <- "Yield"
#Perform best subsets regression
leap1 <- regsubsets(Yield ~ Rain0 + Temp1 + Rain1 + Temp2
                    + Rain2 + Temp3 + Rain3 + Temp4, data = xy, nbest = 1)
summary(leap1)

mod1<- lm(Yield ~  Temp3, data=xy)
summary(mod1)

mod8= lm(Yield ~ Rain0 + Temp1  + Rain2, data = xy)
summary(mod8)

#Model 1 and model 8 are the only ones
#with every predictor significant (from the best subset selection)


boxcox(mod1)      #Shows no transformation on Y is needed
boxcox(mod8)      #Again, close enough to 1 to justify not transforming



box.tidwell(xy$Yield ~ (xy$Temp3)) #No significant transformations
```

```r
box.tidwell(xy$Yield ~ (xy$Temp3+xy$Temp1+xy$Rain0)) #No significant transformations

x.test <- as.data.frame(test.X)
y.test <- as.data.frame(test.Y)
xy.test <- cbind(x.test,y.test)
colnames(xy.test)[which(names(xy.test) == "test.Y")] <- "Yield"

model1.prediction <- predict(mod1,se.fit = TRUE, newdata = xy.test)
sum(abs(model1.prediction$fit-test.Y)^2) #2501.141

model8.prediction <- predict(mod8,se.fit = TRUE, newdata = xy.test)
sum(abs(model8.prediction$fit-test.Y)^2) #3440.105


select(lm.ridge(Yield~Rain0+Temp1+Rain2,lambda = seq(0,10,0.01)))
mod8r= lm.ridge(Yield ~ Rain0+Temp1+Rain2,lambda = 1.66)
#1.66 and 1.72 make essentially the same model (to 3 decimal places) so we will only test 1.66.
mod8r
YPredictRidge = c()
for (i in 1:16){
  YPredictRidge[i] = 46.4465873 +0.5823765*test.X[i,2] -0.4370007*test.X[i,3] +5.3594071*test.X[i,6]
}

sum(abs(YPredictRidge-test.Y)^2) #1897.445
```

Both OLS and ridge regression produced worse estimates for the Yield in the odd numbered years. This is evident in the sum of squared errors in the models being 2501.141, 3440.105 and 1897.445 respectively. Compared to 1527.978 for the KNN regression. This is likely because the OLS and ridge regression models don't take into the Year variable, compared to the KNN regression. They could take into account the Year variable but this would likely to over fitting, with around 30 categorical variables included in the model. One way to fix this would be to build a dynamic linear model, which would incorporate the previous k years values into the model.