ST340 Assignment 2

Dylan Dijk (1802183), Kum Mew Lee (1827149), Aryan Gupta (1826475)

27/02/2021

Contents

1 Expectation Maximization

A: Unique stationary point

In the M-step II we want to maximize $f(\boldsymbol{\mu_{1:k}}) = \sum_{i=1}^{n} \sum_{k=1}^{K} \gamma_{ik} \log(p(\boldsymbol{x_i}|\boldsymbol{\mu_k}))$

And we know that $p(\boldsymbol{x_i}|\boldsymbol{\mu_k}) = \prod_{j=1}^{p} \mu_{kj}^{x_{ij}} (1-\mu_{kj})^{(1-x_{ij})}$

Therefore we want to maximize:

$$f(\boldsymbol{\mu_{1:k}}) = \sum_{i=1}^{n} \sum_{k=1}^{K} \gamma_{ik} \sum_{j=1}^{p} [x_{ij} \log(\mu_{kj}) + (1-x_{ij}) \log(1-\mu_{kj})]$$

Now if we fix $k \in \{1, \ldots, K\}$, we have:

$$f(\boldsymbol{\mu_k}) = \sum_{i=1}^{n} \gamma_{ik} \sum_{j=1}^{p} [x_{ij} \log(\mu_{kj}) + (1-x_{ij}) \log(1-\mu_{kj})]$$

Now taking the partial derivative w.r.t a single μ_j:

$$\frac{\partial f}{\partial \mu_j} = \sum_{i=1}^{n} \frac{\gamma_{ik} x_{ij}}{\mu_j} - \frac{\gamma_{ik}(1-x_{ij})}{1-\mu_j}$$

Setting this equal to zero we get:

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

After rearranging we have:

$$\mu_j = \frac{\sum_{i=1}^{n} \gamma_{ik} x_{ij}}{\sum_{i=1}^{n} \gamma_{ik}} \qquad \Longrightarrow \qquad \boldsymbol{\mu_k} = \frac{\sum_{i=1}^{n} \gamma_{ik} \boldsymbol{x_i}}{\sum_{i=1}^{n} \gamma_{ik}}$$

Therefore have shown the unique stationary point is obtained by choosing for each k

$$\boldsymbol{\mu_k} = \frac{\sum_{i=1}^{n} \gamma_{ik} \boldsymbol{x_i}}{\sum_{i=1}^{n} \gamma_{ik}}$$

B: Newsgroups dataset

i

Here I have just used the function given to us for Lab 4.

```
xs = documents
out_Q1bi = em_mix_bernoulli(xs, K  = 4)
```

1	2	3	4
windows	question	team	fact
help	god	games	world
email	fact	players	case
problem	university	baseball	question
system	problem	hockey	course
software	help	season	government
computer	car	win	problem
program	course	league	state

ii

The algorithm does not tell us what each cluster represents and we have to infer this ourselves.
Above I have made a table giving the associated words of the largest 8 values of μ in $\boldsymbol{\mu}_k$ for each k.

From this table it seems pretty clear that cluster 1 represents articles related to computers and cluster 3 represents articles to sport ('rec' group).
For clusters 2 and 4 it is less clear. Could say cluster 2 represents the articles from the 'science' group as "university" has a large μ and then cluster 4 represents articles from the 'talk' group.
So in summary my guess would be 1 = `comp.*`, 2 = `sci.*`, 3 = `rec`, 4 = `talk.*`

Below I have looked at all possible permutations of the 4 clusters, and calculated the difference between the γ values of each data row and the actual label from the `newsgroups.onehot` dataset. As the γ value represents the probability that each data point belongs to a certain cluster.

```
perm = permutations(n = 4, r = 4, v = 1:4)
gamma_minus_label = vector(length = 24)
for(i in 1:24){
  gamma_minus_label[i] = sum(abs(out_Q1bi$gammas[,perm[i,]] - newsgroups.onehot))
}
```

Comparing my guess for the labeling of the clusters with the order of the `groupnames` vector,

```
groupnames
```

```
> [1] "comp.*" "rec.*"  "sci.*"  "talk.*"
```

the permutation that should give the minimum value should be `1 3 4 2`.

```
algorithm_labels = perm[which.min(gamma_minus_label),]
algorithm_labels
```

```
> [1] 1 3 4 2
```

And the average value of the difference between the actual lables and the gammas for this minimum case is:

```
min(gamma_minus_label)/nrow(documents)
```

```
> [1] 0.8210617
```

iii

So in the function given to us for Lab 4 it uses `.2 + .6*xs[sample(n,K),]` to select the starting μ_k for each k.
This randomly samples K rows from the dataset we are inputting into the algorithm and assigns μ equal to 0.2 if an element of the row is zero and 0.8 if the element is one. Important to note that we shouldn't set a

starting μ equal to zero or one. *The code for how I ran the following 3 cases is in the markdown file*

(A) I am going to rerun the algorithm but now use `.4 + .2*xs[sample(n,K),]`, this now assigns μ equal to 0.4 if an element of the row is zero and 0.6 if the element is one.

Looking again at the value we get for the minimum difference we get between the gammas and the actual labels (divided by number of rows of data)

```
> [1] 0.787949
```

The original function selects equal starting weights to each cluster written as: `rep(log(1/K),K)`. Here the function is using logs to keep the numbers stable.

(B) I will now run the algorithm with starting weights equal to the proportion of each type of article.

Using the same measure:

```
> [1] 0.7703485
```

(C) I will now run the algorithm with starting weights equal to the proportion of each type of article and starting $\mu's$ equal to the proportion of times a word appeared and adding 0.01 so that we have no zero values.

Using the same measure:

```
> [1] 0.8184101
```

Below I have made the same tables as before. The first row has the Table for case A on the left and then for B on the right. Then at the bottom is the table for case C.

1	2	3	4	1	2	3	4
windows	fact	team	question	fact	question	team	windows
email	world	games	god	world	god	games	email
help	case	players	fact	case	fact	players	help
problem	course	baseball	problem	course	problem	baseball	problem
system	question	hockey	university	government	course	hockey	system
software	government	season	course	question	university	season	software
computer	problem	win	christian	problem	christian	win	computer
program	state	league	help	state	world	league	program

1	2	3	4
question	fact	windows	team
god	world	email	games
fact	case	help	players
university	question	problem	baseball
problem	course	system	hockey
help	government	software	season
car	problem	computer	win
course	state	program	league

So in conclusion we can see from the tables that the clusters that seem to represent postings about computers and sport are very stable, but the other two less so.
And looking at the measure I used to determine 'accuracy', the output didn't seem to be too sensitive to changes in starting values. It would of been best if I had ran each case a couple of times, as we do get different output anyways due to the randomness within the functions I used e.g. the functions use `sample`.

2 Two-armed Bernoulli bandits

```
library(mvtnorm)
# Notations: ps=success_parameter; as=arm played;
# rs=reward; ns=number of plays; ss=number of success

# Set Bernoulli success parameters for each arm
ps <- c(0.6,0.4)
```

A: Implementing Thompson Sampling and $\epsilon - decreasing$ strategy

Thompson Sampling:

```
sample_arm.bernoulli <- function(ns,ss){
  alphas <- 1 + ss # success
  betas <- 1 + ns - ss # failures

  t1 <- rbeta(1, alphas[1], betas[1])
  t2 <- rbeta(1, alphas[2], betas[2])
  if(t1 > t2){
    return(1)
  } else {
   return(2)
  }
}

# Bernoulli => alpha=beta=1

thompson.bernoulli <- function(ps,n){
  as <- rep(0,n)
  rs <- rep(0,n)

  # initial number of plays and number of successes is 0 for each arm
  ns <- rep(0,2); ss <- rep(0,2)

  for (i in 1:n){
    a <- sample_arm.bernoulli(ns,ss)
    r <- ifelse(runif(1) < ps[a], 1, 0)
    ns[a] <- ns[a] + 1
    ss[a] <- ss[a] + r
    as[i] <- a
    rs[i] <- r
  }
  return(list(as=as,rs=rs))
}
```

$\epsilon - decreasing$:

```
epsilon.decreasing <- function(ps, C, n, power){
  as <- rep(0,n)
  rs <- rep(0,n)

  # initial number of plays and number of successes is 0 for each arm
  ns <- rep(0,2); ss <- rep(0,2)
```

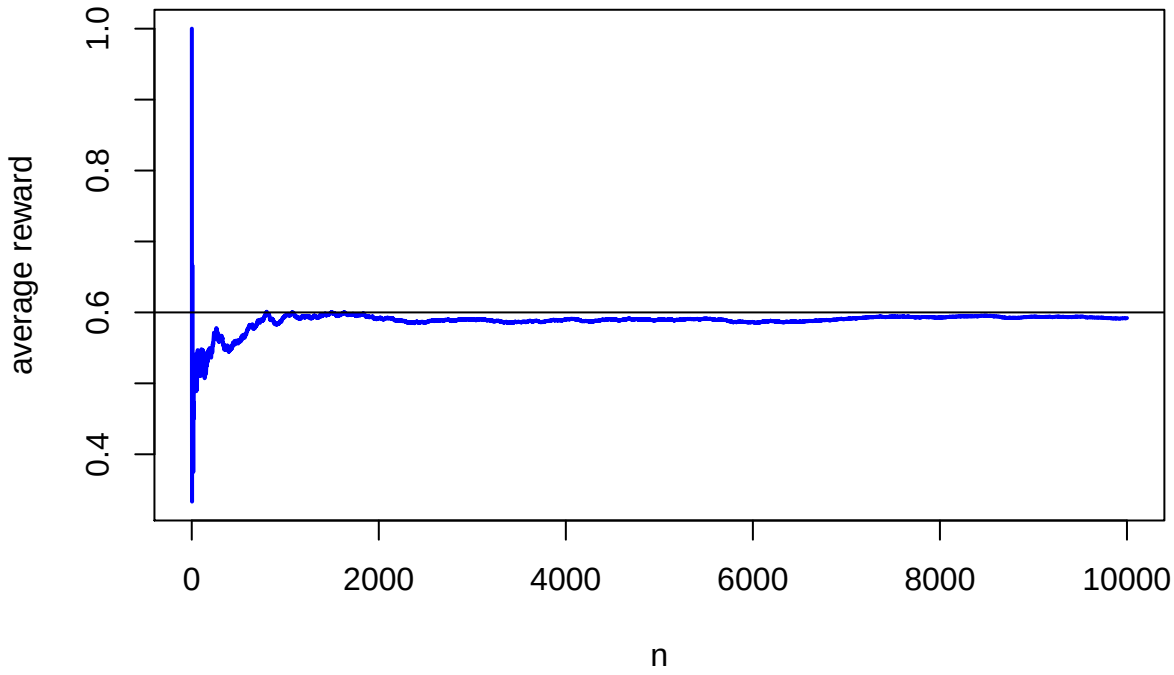
```
  # at first, play each arm once
  for (i in 1:2) {
    a <- i
    r <- runif(1) < ps[a]
    ns[a] <- ns[a] + 1
    ss[a] <- ss[a] + r
    as[i] <- a
    rs[i] <- r
  }

  # now follow the epsilon decreasing strategy
  for (i in 3:n) {
    # with probability epsilon, pick an arm uniformly at random
    epsilon <- min(1, C/i^power)
    if (runif(1) < epsilon) {
      a <- sample(2,1)
    } else { # otherwise, choose the "best arm so far".
        a <- which.max(ss/ns)
    }
    # simulate the reward
    r <- ifelse(runif(1) < ps[a], 1, 0)

    # update the number of plays, successes
    ns[a] <- ns[a] + 1
    ss[a] <- ss[a] + r

    # record the arm played and the reward received
    as[i] <- a
    rs[i] <- r
  }
  return(list(as=as,rs=rs))
}
```

B: Behavior of $\epsilon - decreasing$ for $\epsilon_n = min(1, Cn^{-1})$

At time n, we play an arm at random with probability ϵ_n, and we play the one with the best rate of success so far with probability $1 - \epsilon_n$. For $\epsilon_n = min(1, Cn^{-1})$, as n increases, ϵ_n decreases and we will play the one with the best rate of success with probability approaching 1.

To check this we compute:

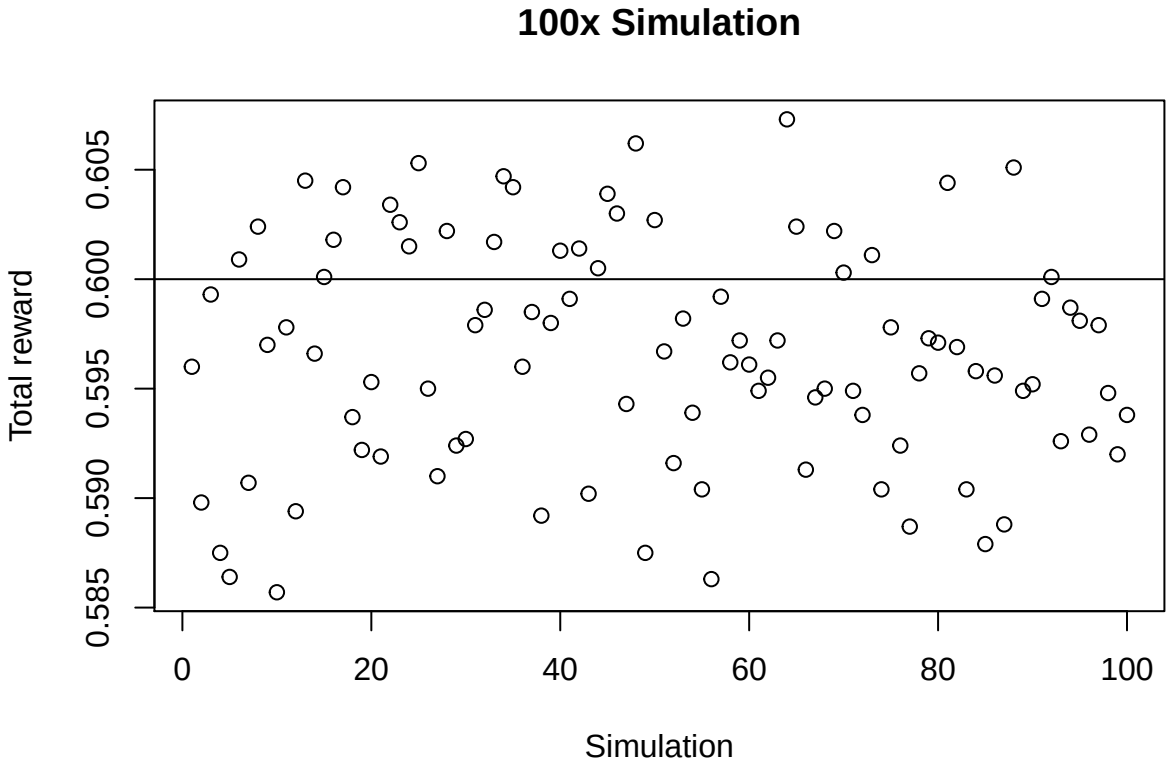
```
eg.check <- epsilon.decreasing(ps=ps,C=50,n=1e4,1)
avg <- eg.check$rs[1]
sumb <- eg.check$rs[1]
for(i in 2:length(eg.check$rs)){
    sumb[i] <- sum(sumb[i-1]+eg.check$rs[i])
    avg[i] <- sumb[i]/i
}
n <- c(1:10000)
plot(n, avg, type="l", lwd=2, col="blue",
     xlab="n", ylab="average reward")
abline(h=0.6)
```



As shown in the graph above, as n increases, the average reward converges to 0.6, which is the best rate of success so far. Hence this result is consistent with our implementation.

Here we simulate the $\epsilon - decreasing$ strategy for $\epsilon_n = min(1, Cn^{-1})$ for a hundred times:

```
Simulation <- c(1:100)
eg.out_b <- 0; eg.out_brun <- 0
for(i in 1:length(Simulation)){
  eg.out_brun <- epsilon.decreasing(ps=ps,C=50,n=1e4,1)
  eg.out_b[i] <- sum(eg.out_brun$rs)/length(eg.out_brun$rs)
}
plot(Simulation, eg.out_b,
     main="100x Simulation", xlab="Simulation", ylab="Total reward")
abline(h=0.6)
```



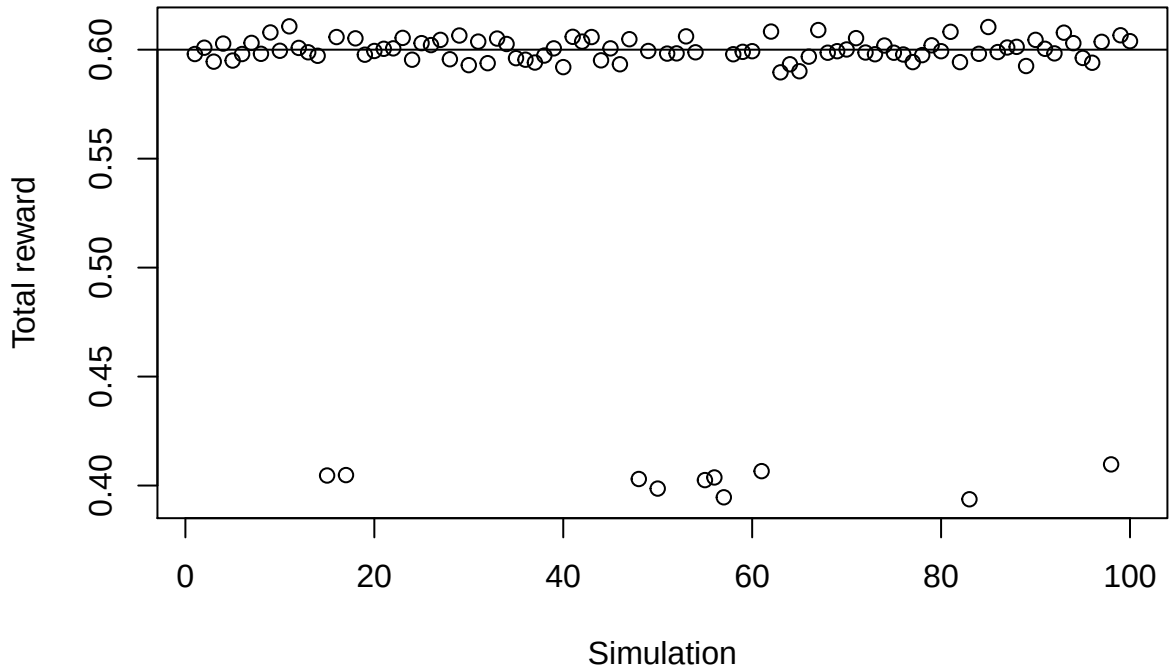
The Total reward is consistent (+-0.02) to the maximum reward of 0.6

C: Behavior of $\epsilon - decreasing$ for $\epsilon_n = min(1, Cn^{-2})$.

For $\epsilon_n = min(1, Cn^{-2})$, it acts similar to (b), but the ϵ_n will decrease faster than in (b) and hence we will play the one with the best rate of success with probability approaching 1 earlier. However, this will cause the decrease in samples played at random and this might lead to an inaccuracy of the best rate of success.

To check this we simulate the $\epsilon - decreasing$ strategy for $\epsilon_n = min(1, Cn^{-2})$ for a hundred times::

```
eg.out_c <- 0; eg.out_crun <- 0
for(i in 1:length(Simulation)){
  eg.out_crun <- epsilon.decreasing(ps=ps,C=50,n=1e4,2)
  eg.out_c[i] <- sum(eg.out_crun$rs)/length(eg.out_crun$rs)
}
plot(Simulation, eg.out_c,
     main="100x Simulation", xlab="Simulation", ylab="Total reward")
abline(h=0.6)
```



As shown in the simulation above, there exist some results with probability 0.40 which is inaccurate. Hence this is consistent with our implementation.

D: Compare and contrast the implementations of $\epsilon - decreasing$ and Thompson sampling.

To compare both implementations, first we check their performance for one simulation:

```
eg.out <- epsilon.decreasing(ps=ps,C=50,n=1e4,1)
cat("Epsilon-decreasing = ", sum(eg.out$rs)/length(eg.out$rs))
```

```
> Epsilon-decreasing =  0.6025
```

```
thompson.bernoulli.out <- thompson.bernoulli(ps=ps,n=1e4)
cat("Thompson Bernoulli = ", sum(thompson.bernoulli.out$rs)/length(thompson.bernoulli.out$rs))
```

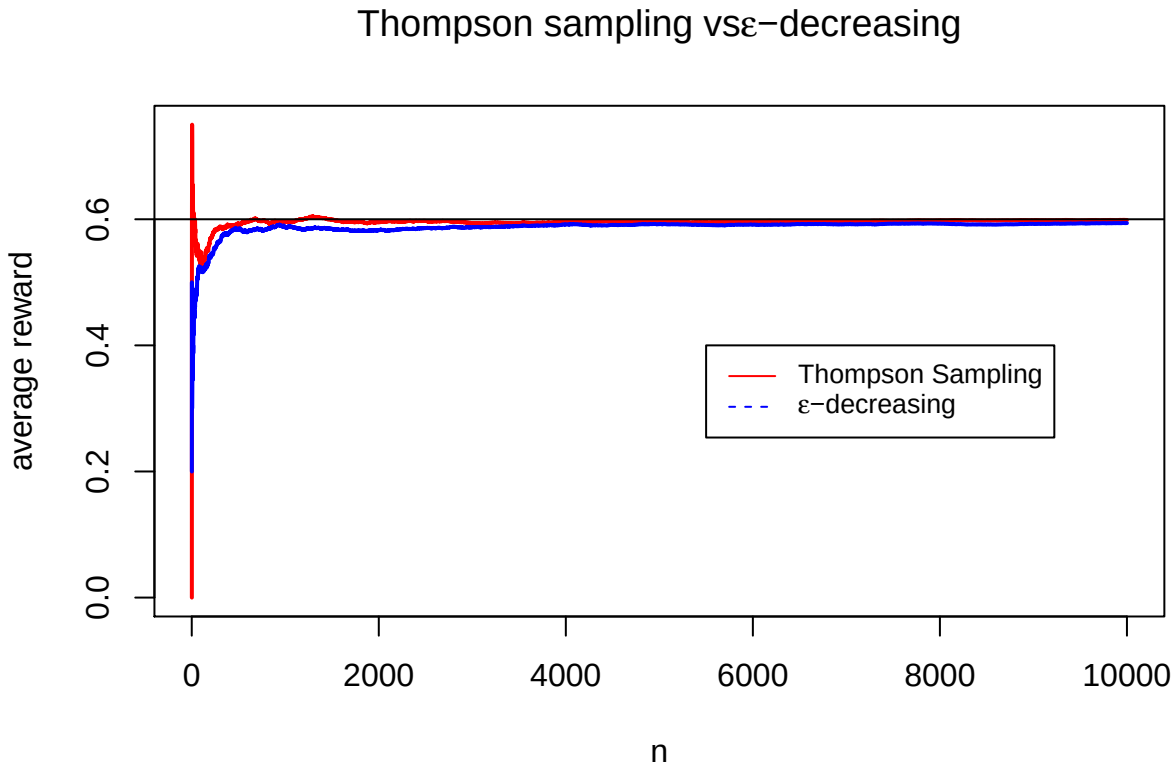
```
> Thompson Bernoulli =  0.5974
```

Obviously, the result above doesn't tell us much about the difference between the algorithms. Now, we try to check the performance of the algorithm with an increasing n:

Note: Since the output of the strategies will be necessarily random, the experiment is repeated 5 times and the mean of each n is calculated and presented.

```
# For Thompson Sampling
# i=5 for 5 repeated experiment
avg_r <- matrix(0,5,10000)
# Constructing a matrix with row for number of experiment,
# column for the output of each n
for(i in 1:5){
  thompson.bernoulli.out <- thompson.bernoulli(ps=ps,n=1e4)
  avg_r[1,1] <- thompson.bernoulli.out$rs[1]
  sum_r <- thompson.bernoulli.out$rs[1]
```

```
  for(j in 2:10000){
    sum_r[j] <- sum(sum_r[j-1]+thompson.bernoulli.out$rs[j])
    avg_r[i,j] <- sum_r[j]/j
  }
}
# Taking the average of each n of all 5 experiments
average_random <- 0
for(i in 1:10000){
  average_random[i] <- sum(avg_r[1:5,i])/5
}



# For Epsilon Decreasing
avg_r2 <- matrix(0,5,10000)
for(i in 1:5){
  eg.out <- epsilon.decreasing(ps=ps,C=50,n=1e4,1)
  avg_r2[1,1] <- eg.out$rs[1]
  sum_r2 <- eg.out$rs[1]
  for(j in 2:10000){
    sum_r2[j] <- sum(sum_r2[j-1]+eg.out$rs[j])
    avg_r2[i,j] <- sum_r2[j]/j
  }
}
# Taking the average of each n of all 5 experiments
average_random2 <- 0
for(i in 1:10000){
  average_random2[i] <- sum(avg_r2[1:5,i])/5
}


n = c(1:10000)
plot(n, average_random, type="l", lwd=2, col="red",
     main= expression(paste("Thompson sampling vs", epsilon,"-decreasing")),
     xlab="n", ylab="average reward")
lines(n,average_random2, lwd=2, col="blue")
abline(h=0.6)

legend(5500,0.4, legend=c("Thompson Sampling",
                          expression(paste(epsilon,"-decreasing"))),
       col=c("red","blue"),lty=1:2, cex=0.8)
```



As shown in the graph above, it can be seen that the average reward of Thompson Sampling strategy converges to 0.6 faster than $\epsilon - decreasing$ strategy. Hence the general performance of Thompson Sampling is better than $\epsilon - decreasing$.

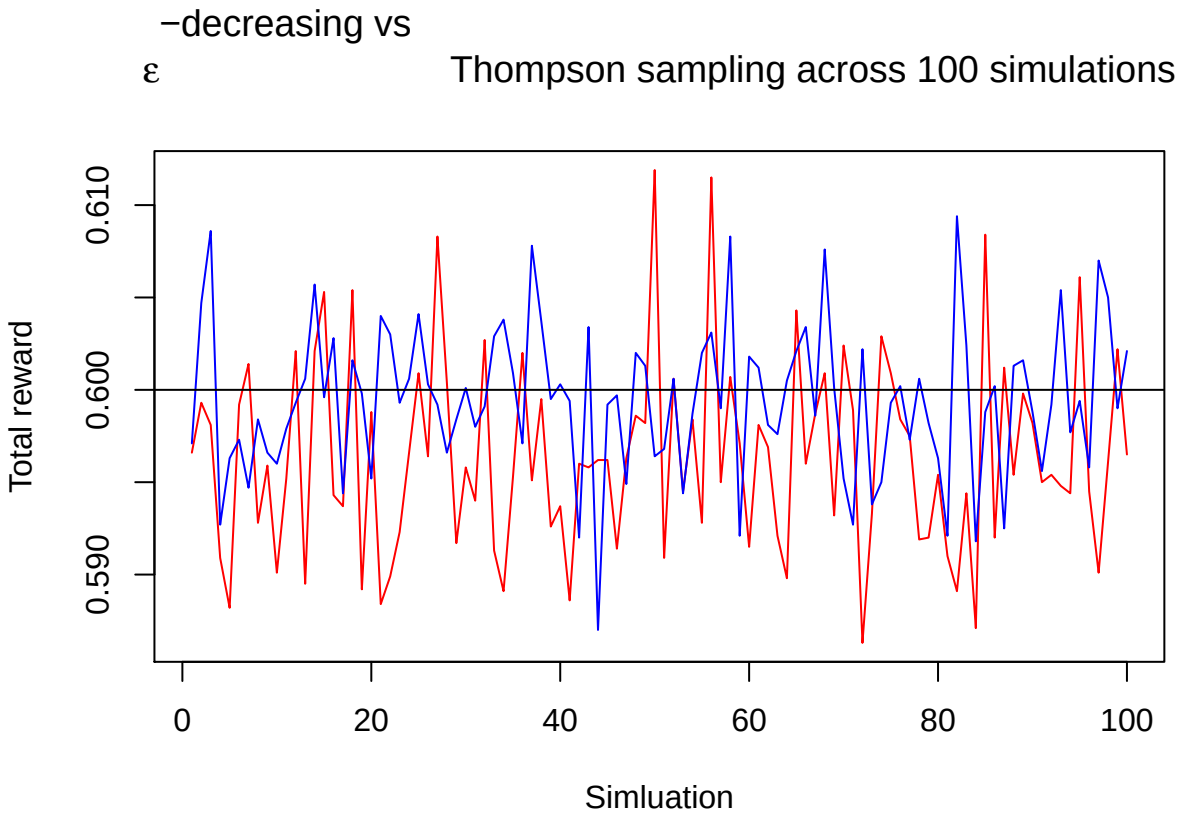
Now we compare each on their total reward across each simulation:

```
egrunout <- 0; egrun <- 0; thomrunout <- 0; thomrun <- 0
for(i in 1:length(Simulation)){
  egrunout <- epsilon.decreasing(ps=ps,C=50,n=1e4,1)
  thomrunout <- thompson.bernoulli(ps=ps,n=1e4)
  egrun[i] = sum(egrunout$rs)/length(egrunout$rs)
  thomrun[i] = sum(thomrunout$rs)/length(thomrunout$rs)
}
```

```
plot(Simulation, egrun, type="l", lwd=1, col="red",
     main= expression(paste(epsilon,"-decreasing vs
                            Thompson sampling across 100 simulations")),
     xlab="Simluation", ylab="Total reward")
```

```
> Warning in title(...): font metrics unknown for character 0xa

> Warning in title(...): font metrics unknown for character 0xa
```

```
lines(Simulation, thomrun, lwd=1, col="blue")
abline(h=0.6)
```



From the graph above, it can be seen that both strategies are quite comparable in terms of their total reward performance. However, $\epsilon - decreasing$ (blue) has a higher risk compared to Thompson Sampling(red) as it is not as consistent as Thompson Sampling strategy.

3: k nearest neighbours

A: kNN regression

```
knn.regression.test <- function(k,train.X,train.Y,test.X,test.Y,distances) {
  estimates = c() # creating a vector to store estimates
  for (i in 1:nrow(test.X)){
    d = c() # creating a vector to store the distances between test.X and train.X points

    # a loop to row wise calculate the user-defined distances and store them
    for (j in 1:nrow(train.X)){
      d = c(d, distances(train.X[j,],test.X[i,]))
    }

    ref = c(1:nrow(train.X)) # vector to store initial positions of points
    df = data.frame(ref, d) # a data frame with distances and initial positions
    df = df[order(df$d),] # ascending order of distances
    df = df[1:k,] # selecting the k closest points

    # dummy variables
    est = 0
    s = 0
```

```
    # use Inverse-distance weighting (IDW) to calculate estimates
    for (i in 1:k){
      # IDW states - if the distance is 0 we simply use the train.Y value as estimate
      if (df$d[i] == 0){
        est = train.Y[df$ref[i]]
        s = 1
        break
      } else {
        # Otherwise, IDW uses a formula such that closer points are weighed higher
        est = est + (train.Y[df$ref[i]]/df$d[i])
        s = s + 1/df$d[i]
      }
    }

    estimates = c(estimates, est/s) # vector of estimates for each test.X point
  }
  # Calculating the squared difference to real values - used as measure of model accuracy
  print(sum((test.Y-estimates)^2))
}
```

B: Tests

The code below describes the ℓ_1 distances between pairs of row vectors in two matrices as described in Lab6:

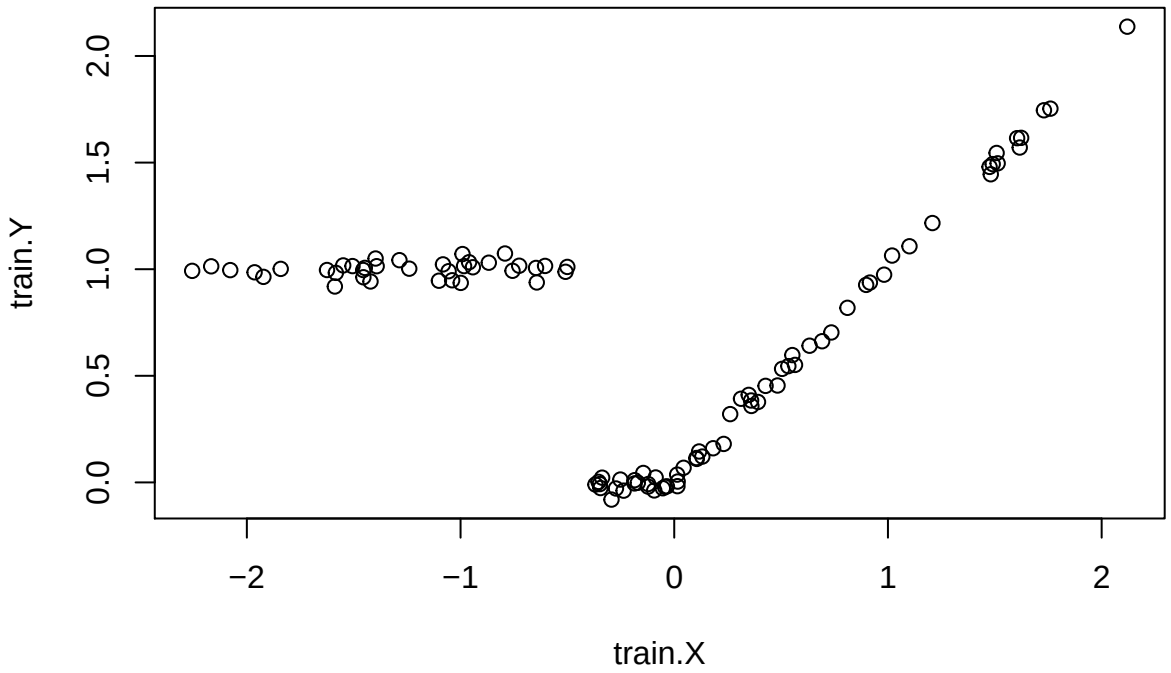
```
distances.l1 <- function(X,W) {
  sum(abs(X-W))
}
```

The upper bound for k values evaluated is the number of rows in the training X value data set as that defines the total number of individual data points that can be considered to calculate an estimate. The lower bound, meanwhile, will be a single data point, i.e. the "closest data point".

Keeping this mind we can evaluate the toy datasets for the k range from 1 to 100.

Toy dataset 1 with k = 2:

```
n <- 100
set.seed(2021)
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 <- 2
knn.regression.test(k,train.X,train.Y,test.X,test.Y,distances.l1)
```

```
> [1] 3.91539
```

Other k values for the Toy dataset 1:

```
Table = matrix(Table,ncol=1,byrow=TRUE)
rownames(Table) = c("k = 1","k = 10","k = 20","k = 30","k = 40",
                    "k = 50","k = 60","k = 70","k = 80","k = 90","k = 100")
colnames(Table)<-c("Squared Difference")
Table <- as.table(Table)
Table
```

```
>         Squared Difference
> k = 1             3.587658
> k = 10            2.805828
> k = 20            3.561326
> k = 30            4.606498
> k = 40            5.967978
> k = 50            7.484759
> k = 60            8.886167
> k = 70            9.464738
> k = 80            9.665040
> k = 90            9.802249
> k = 100           9.912574
```

Closer estimates to real values suggests a more accurate model therefore, a smaller squared difference is preferable between estimates and real test Y values for the kNN regression.

The table suggests a general trend where the squared difference gradually increases k = 10 onward. The absolute difference when k = 10 is lower than when k = 1 or k = 2 as initially set. Therefore, k = 10 is preferred amongst considered k values in estimating for the toy dataset 1.

Toy dataset 2 with k = 3:

```
set.seed(100)
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 <- 3
knn.regression.test(k,train.X,train.Y,test.X,test.Y,distances.l1)
```

```
> [1] 3.232214
```

Other k values for the Toy dataset 2:

```
Table = matrix(Table,ncol=1,byrow=TRUE)
rownames(Table) = c("k = 1","k = 10","k = 20","k = 30","k = 40","k = 50",
                    "k = 60","k = 70","k = 80","k = 90","k = 100")
colnames(Table)<-c("Squared Difference")
Table <- as.table(Table)
Table
```

```
>         Squared Difference
> k = 1             3.072546
> k = 10            5.009417
> k = 20            8.545205
> k = 30           12.306883
> k = 40           16.378280
> k = 50           19.300316
> k = 60           22.845359
> k = 70           25.848833
> k = 80           29.465413
> k = 90           33.717301
> k = 100          38.468815
```

The table again suggests a general trend where the squared difference gradually increases k = 10 onward. The absolute difference when k = 1 is lower than when k = 10 and k = 3 as initially set. Therefore, k = 1 is preferred among considered k values in estimating for the toy dataset 1.

C: Predicting on the `lasso` dataset

We will try to predict the yield in the years 1931, 1933, . . . based on the data from 1930, 1932, . . . using the kNN regression function.

The code below describes the a function to calculate a matrix of pairwise ℓ_2 distances as described in Lab6:

```
distances.l2 <- function(X,W) {
  sqrt(sum((X-W)^2))
}
```

```
#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)
```

```
> [1] 1530.799
```

Exploring with different k values on the `Iowa` dataset:

Once again the upper bound for k values evaluated is the number of rows in the training X value data set and the lower bound is one.

Keeping this mind we can evaluate the toy datasets for the k range from 1 to 17.

```
Table = matrix(Table,ncol=1,byrow=TRUE)
rownames(Table) = c("k = 1","k = 3","k = 5","k = 10","k = 12","k = 17")
colnames(Table)<-c("Squared Difference")
Table <- as.table(Table)
Table
```

```
>        Squared Difference
> k = 1            1892.310
> k = 3            1942.714
> k = 5            1530.799
> k = 10           1845.360
> k = 12           1741.733
> k = 17           1714.341
```

Closer estimates to real values suggests a more accurate model therefore, a smaller squared difference is preferable between estimates and real test Y values for the kNN regression.

Looking at the table above we notice setting the k values to 5 produces the smallest squared difference. Hence, k = 5 is the best k value among considered k values as it produces estimates closest to the real value. Beyond this there is no discernible general trend.

D: Ordinary least squares regression and Ridge regression

Ordinary least squares regression (OLS):

```
train.X = data.frame(train.X)

# fitting a linear model for all exploratory variables
fit_OLS = lm(train.Y ~ Year + Rain0+ Temp1+ Rain1+  Temp2+ Rain2+ Temp3+ Rain3+ Temp4,data=train.X)

# using predict function to calculate the estimates
estimates = predict(fit_OLS, data.frame(test.X))

# Calculating the squared difference to real values - used to compare to other models
sum((test.Y-estimates)^2)
```

```
> [1] 1891.556
```

OLS produces estimates that are slightly preferable to the k = 1 kNN regression in terms of squared difference from real test Y values. However, the squared difference is much greater than the k = 5 squared difference and hence, k = 5 kNN regression is preferred in estimating for the `Iowa` dataset.

Ridge regression:

```
library(glmnet)
train.X = as.matrix(train.X)

# possible lambda values
lambdas = 10^seq(2, -3, by = -.1)
# fitting model
ridge_reg = glmnet(train.X, train.Y, nlambda = 25, alpha = 0, family = 'gaussian', lambda = lambdas)
```

```r
# finding an optimal lambda
cv_ridge = cv.glmnet(train.X, train.Y, alpha = 0, lambda = lambdas, grouped=FALSE)
optimal_lambda = cv_ridge$lambda.min

# using predict function to calculate the estimates
estimates = predict(ridge_reg, s = optimal_lambda, newx = test.X)

# Calculating the squared difference to real values - used to compare to other models
sum((test.Y-estimates)^2)
```

```
> [1] 1671.125
```

Ridge regression produces estimates that are slightly preferable to the k = 12 kNN regression in terms of squared difference from real test Y values. In comparison to OLS, ridge regression produces a lower absolute difference and hence is the better method with regards to estimating for the `Iowa` dataset. However, once again the squared difference is much greater than the k = 5 squared difference and hence, k = 5 kNN regression is preferred in estimating for the `Iowa` dataset.