Summary:

Q1	40 / 40
Q2	29 / 30
Q3	30 / 30
Total	99 / 100

ST340 Assignment 2

Anojan Mariathas (2244070), Jude Thomas (2210732)

2025-02-24

Contributions

We worked closely on all aspects of this assignment discussing and contributing to each part as a team . Jude (2210732) took the lead with Question 1, and Anojan (2244070) took the lead on Question 2 and we jointly collaborated on Question 3.

Question 1

Part (a)

Question 1a) 10/10

We want to obtain the stationary point of the function given by

$$f(\boldsymbol{\mu}_{1:K}) = f(\boldsymbol{\mu}_1, \ldots, \boldsymbol{\mu}_K) = \sum_{i=1}^{n} \sum_{k=1}^{K} \gamma_{ik} \log p(x_i|\boldsymbol{\mu}_k),$$

where for each observation $i \in \{1, \ldots, n\}$, $\boldsymbol{x}_i \in \{0,1\}^p$ and for each $k \in \{1, \ldots, K\}$ we have $\boldsymbol{\mu}_k = (\mu_{k1}, \mu_{k2}, \ldots, \mu_{kp}) \in [0,1]^p$.

The Bernoulli likelihood for each component k is given by:

$$p(\boldsymbol{x}_i|\boldsymbol{\mu_k}) = \prod_{j=1}^{p} \mu_{kj}^{x_{ij}} (1-\mu_{kj})^{1-x_{ij}}$$

and thus the log-likelihood is given by

$$\log p(\boldsymbol{x}_i|\boldsymbol{\mu_k}) = \sum_{j=1}^{p} [x_{ij} \log \mu_{kj} + (1-x_{ij}) \log(1-\mu_{kj})]$$

We are now ready to compute the derivative (note that we have directly replaced $\log p(\boldsymbol{x}_i|\boldsymbol{\mu_k})$ with the expression derived above).

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

For a fixed k the function can be written as

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

This decomposition means that we can optimise each coordinate μ_{kj} separately.

For fixed k and j, define

$$f_{k,j}(\mu_{kj}) = \sum_{i=1}^{n} \gamma_{ik} \left[x_{ij} \log \mu_{kj} + (1 - x_{ij}) \log(1 - \mu_{kj}) \right]$$

to denote the j^{th} coordinate of μ_k. To compute the derivative, we take the following steps:

Step 1:

First we differentiate $f_{k,j}(\mu_{kj})$ with respect to μ_{kj}:

$$\frac{\partial f_{k,j}(\mu_{kj})}{\partial \mu_{kj}} = \sum_{i=1}^{n} \gamma_{ik} \left(\frac{x_{ij}}{\mu_{kj}} - \frac{1 - x_{ij}}{1 - \mu_{kj}} \right).$$

Step 2:

To find the stationary point we set the derivative equal to zero and solve as follows:

$$\begin{aligned}
&\sum_{i=1}^{n} \gamma_{ik} \left(\frac{x_{ij}}{\mu_{kj}} - \frac{1 - x_{ij}}{1 - \mu_{kj}} \right) = 0 \\
&\sum_{i=1}^{n} \gamma_{ik} \left[x_{ij}(1 - \mu_{kj}) - (1 - x_{ij})\mu_{kj} \right] = 0 \\
&\sum_{i=1}^{n} \gamma_{ik} \left[x_{ij} - \mu_{kj} \right] = 0 \\
&\sum_{i=1}^{n} \gamma_{ik} x_{ij} - \mu_{kj} \sum_{i=1}^{n} \gamma_{ik} = 0.
\end{aligned}$$

Solving for μ_{kj}, we get

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

Since this derivation holds for every coordinate $j = 1, \ldots, p$, the vector form for the maximum point for cluster k is

$$\boldsymbol{\mu}_k = \frac{\sum_{i=1}^{n} \gamma_{ik} \boldsymbol{x}_i}{\sum_{i=1}^{n} \gamma_{ik}}, \quad \forall\, k \in \{1, \ldots, K\}$$

which is the unique stationary point for the function $f(\boldsymbol{\mu}_{1:K})$.

Part (b)

```
load("20newsgroups.rdata")
```

(i) Question bi) 6/6

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

# Compute the log-likelihood directly for the Bernoulli mixture model
compute_ll_direct <- function(xs, mus, lws) {
  ll <- 0
  n <- dim(xs)[1]
  K <- dim(mus)[1]
  for (i in 1:n) {
    s <- 0
    for (k in 1:K) {
      # Compute log-probability for observation i under component k
      logp <- sum(xs[i,] * log(mus[k,]) + (1 - xs[i,]) * log(1 - mus[k,]))
      s <- s + exp(lws[k]) * exp(logp)
    }
    ll <- ll + log(s)
  }
  return(ll)
}

# Main function
em_mix_bernoulli <- function(xs, K, start = NULL, max.numit = Inf, verbose=TRUE) {
  n <- dim(xs)[1]   # number of observations
  p <- dim(xs)[2]   # number of features

  # Initialise log mixing weights (lws) equally in log-domain.
  lws <- rep(log(1/K), K)

  # Initialise cluster means (Bernoulli parameters) safely.

  if (is.null(start)) {
    mus <- 0.3 + 0.5 * xs[sample(n, K), ]
  } else {
    mus <- start
  }
  # Clip mus to ensure values are strictly between 0 and 1
  mus[which(mus >= 1, arr.ind = TRUE)] <- 1 - 1e-15
  mus[which(mus <= 0, arr.ind = TRUE)] <- 1e-15

  # Initialise responsibilities matrix.
  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 responsibilities for each observation and component.
    for (i in 1:n) {
      lprs <- rep(0, K)
      for (k in 1:K) {
        # Compute log probability for observation i under component k inline:
        logp <- sum(xs[i,] * log(mus[k,]) + (1 - xs[i,]) * log(1 - mus[k,]))
        lprs[k] <- lws[k] + logp
      }
      gammas[i,] <- exp(lprs - logsumexp(lprs))
    }

    # Update log-likelihood (using the direct method)
    ll <- compute_ll_direct(xs, mus, lws)

    # M-step: Update parameters using weighted averages.
    Ns <- rep(0, K)
    for (k in 1:K) {
      Ns[k] <- sum(gammas[, k])
      lws[k] <- log(Ns[k]) - log(n)
      # Update mus for component k by accumulating contributions from all observations.
      mus[k,] <- rep(0, p)
      for (i in 1:n) {
        mus[k,] <- mus[k,] + (gammas[i, k] / Ns[k]) * xs[i,]
      }
    }

    # Clip mus again to ensure they stay strictly within (0,1)
    mus[which(mus >= 1, arr.ind = TRUE)] <- 1 - 1e-15
    mus[which(mus <= 0, arr.ind = TRUE)] <- 1e-15



    # Check for convergence: Only check if ll and ll.old are finite.
    if (is.finite(ll) && is.finite(ll.old) && abs(ll - ll.old) < 1e-5) {
      converged <- TRUE
    }
  }

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

We now run the EM algorithm on the given dataset, and only present the final iteration and the value that it converges to

```
set.seed(123)
result <- em_mix_bernoulli(documents, K = 4, verbose = FALSE)
```

```
## [1] "iteration : log-likelihood"
```

```
tibble_result <- tibble(iterations = result$numit, converged = result$ll)
kable(tibble_result)
```

iterations	converged
297	-235933.7

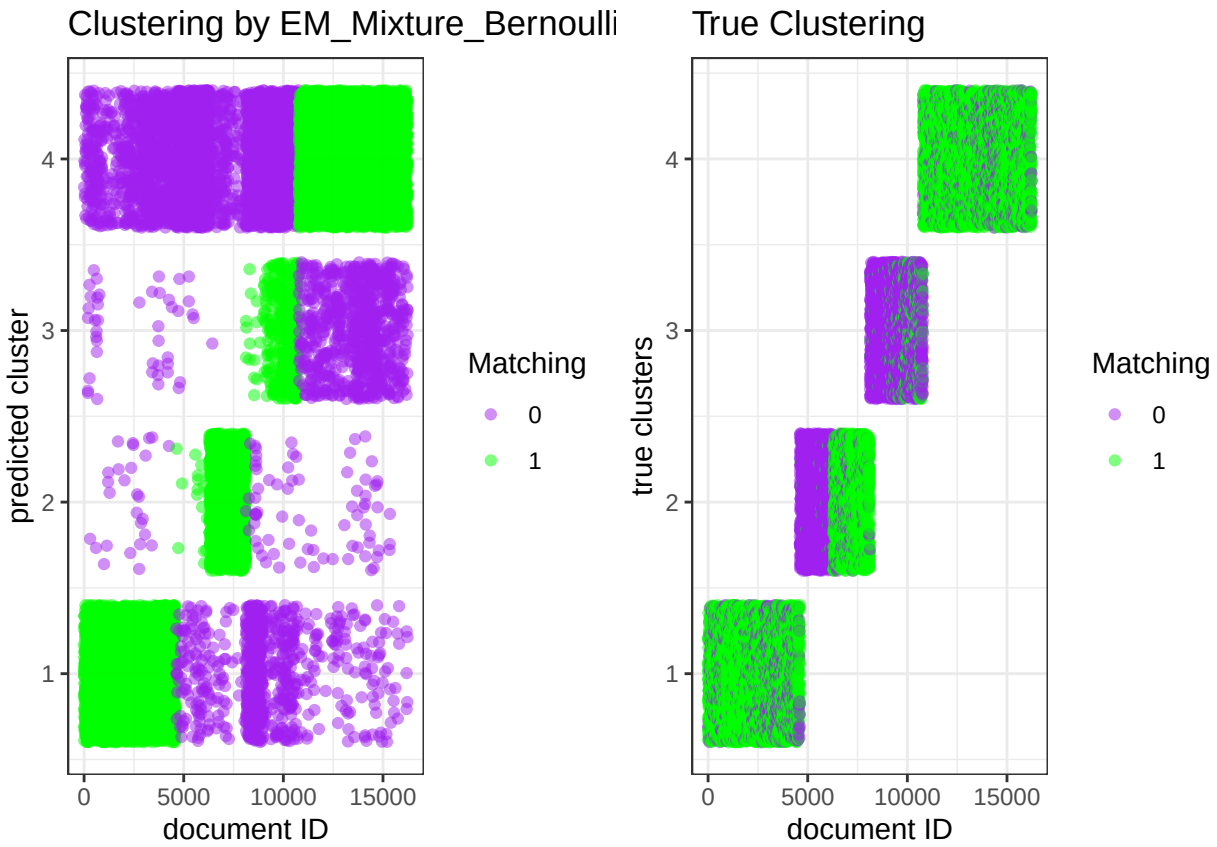
(ii) Question bii) 11/11

```
tibble_weighting <- tibble(Cluster = 1:length(result$lws),
       Weighting = round(exp(result$lws), 2)) %>%
  arrange(desc(Weighting))

kable(tibble_weighting)
```

Cluster	Weighting
4	0.53
1	0.29
2	0.11
3	0.08

This table presents the mixing coefficients (or weightings), denoted by w_k, for each cluster in the fitted mixture model. Each row represents one of the four Bernoulli distributions learned by the algorithm, and the corresponding weighting indicates the overall proportion of observations assigned to that cluster. For example, a weighting of $w_4 = 0.53$ for cluster 4 means that roughly 53% of the documents are most likely explained by the Bernoulli parameters associated with that cluster, while clusters 1, 2, and 3 account for approximately 29%, 11%, and 8% of the observations, respectively.

```
output <- tibble(
  max_gammas = apply(result$gammas, 1 , which.max),
  newsgroups = newsgroups,
  id = 1:length(newsgroups)
) %>%
  mutate(correct = if_else(max_gammas == newsgroups, 1, 0))

# plot predicted clustering
em_result_plot <- output %>%
  ggplot(aes(x = id, y = max_gammas, colour = factor(correct))) +
  geom_jitter(alpha = 0.5) +
  labs(x = "document ID", y = "predicted cluster",
       title = "Clustering by EM_Mixture_Bernoulli", colour = "Matching") +
  scale_colour_manual(values = c("0" = "purple", "1" = "green")) +
  theme_bw()

#plot true clustering

true_plot <- output %>%
  ggplot(aes(x = id, y = newsgroups, colour = factor(correct))) +
  geom_jitter(alpha = 0.5) +
  labs(x = "document ID", y = "true clusters",
       title = "True Clustering", colour = "Matching") +
  scale_colour_manual(values = c("0" = "purple", "1" = "green")) +
  theme_bw()

grid.arrange(em_result_plot, true_plot, ncol = 2)
```



The left panel shows each document's predicted cluster on the y-axis, coloured by whether it matches the true label (green for correct, purple for incorrect). The right panel displays the actual clusters (i.e., newsgroups) for comparison. Visually, some clusters are predominantly correct (mostly green), whereas others contain more purple points, indicating incorrect classification. One cluster dominates a large portion of the data, consistent with the highest mixing weight. Overall, the algorithm achieves reasonable separation but still misclassifies certain documents.

Well done!

The accuracy of the clustering algorithm can be measured if the true labels are available (e.g. the newsgroups' topic tags). However, clusters are assigned arbitrary indices (e.g., 1-4) that do not inherently align with semantic categories like `comp.*` or `sci.*`. Directly comparing cluster indices to true labels risks misrepresenting accuracy due to mismatched labelling (e.g., Cluster 1 might correspond to `sci.*` instead of `comp.*`). To address this, predicted clusters must first be optimally realigned to true labels using methods like the Hungarian algorithm, which maps clusters to their best-matching topics to maximise agreement. This realignment enables meaningful accuracy calculations to avoid biases from label ordering and ensures robust evaluation of clustering quality.

(iii) Question b(iii) 13/13

The experiments reveal that the EM algorithm is highly sensitive to its initial parameter settings. As shown in the table, different mean initialisations yield final log-likelihoods in a similar range (approximately -235926 to -240277), yet the number of iterations required for convergence varies (from around 266 to 294 iterations). Moreover, varying the `set.seed` produced similar log-likelihoods, but the iteration counts varied dramatically—from as few as 200 to as many as 600. This variability highlights the importance of robust initialisation methods and multiple random restarts to achieve consistent clustering performance.

```
my_table <- data.frame(
  # Different mu initialisation
  # Changing the mu in em_mix_bernoulli
```

```
  # eg. (0.2,0.8) denotes mus <- 0.2 + 0.8 * xs[sample(n, K), ]
  "Mu Initial" = c("(0.2,0.8)", "(0.1,0.7)", "(0.2,0.5)", "(0.3,0.3)", "(0.3,0.7)"),
  "Iterations" = c("284", "294", "292", "287", "266"),
  "log likelihood" = c(-240151, -235926, -235926, -235926, -240277)
)

# Print the table using knitr
knitr::kable(my_table, align = 'c', caption = "Results from experimenting with mu")
```

Table 3: Results from experimenting with mu

Mu.Initial	Iterations	log.likelihood
(0.2,0.8)	284	-240151
(0.1,0.7)	294	-235926
(0.2,0.5)	292	-235926
(0.3,0.3)	287	-235926
(0.3,0.7)	266	-240277

Question 2

Part (a)

We implement the Thompson sampling strategy for a two-armed Bernoulli bandit. This algorithm works by maintaining a Beta posterior for each arm, samples a probability from the updated distribution at each step and selects the arm with the highest sampled value, while continuously updating beliefs about the success probabilities. (Note that certain parts of the following code are sourced from Lab 5).

```
# Select an arm based on Thompson sampling
sample_arm.bernoulli <- function(ns, ss) {

  # ns: number of times each arm has been played
  # ss: number of successes for each arm

  # Calculate alpha and beta for each arm
  alphas <- 1 + ss
  betas <- 1 + ns - ss

  # Sample from the posterior Beta distribution for each arm
  m1 <- rbeta(1, alphas[1], betas[1])
  m2 <- rbeta(1, alphas[2], betas[2])

  # Choose the arm with the higher sampled value
  if (m1 > m2) {
    return(1)
  } else {
    return(2)
  }
}


# Thompson sampling algorithm for Bernoulli bandits
thompson.bernoulli <- function(ps, n){
  # as: Vector to store actions (arms)
```

```
  # rs: Vector to store rewards
  as <- rep(0,n)
  rs <- rep(0,n)

  # ns: Number of times each arm has been played
  # ss: Number of successes for each arm
  ns <- rep(0,2)
  ss <- rep(0,2)

  for(i in 1:n){
  a <- sample_arm.bernoulli(ns,ss)
  r <- rbinom(1,1,ps[a]) #A binomial distribution with 1 trial is a Bernoulli distribution

  # Update counts
  ns[a] <- ns[a] + 1
  ss[a] <- ss[a] + r

  # Store results
  as[i] <- a
  rs[i] <- r
  }
  return(list(as=as,rs=rs))
}
```

Now we implement the ϵ-decreasing strategy as follows. This strategy works by initially playing each arm once, then gradually shifts from exploration to exploitation.

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

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

  # First we play each arm once
  for (i in 1:2){
    a <- i
    r <-  rbinom(1,1,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) {

    epsilon_i <- min(1, C * (i^(-k)))

    if (runif(1) < epsilon_i) {
      a <- sample(1:2, 1)
    } else {
      a <- which.max(ss / ns)
    }
```

```
    r <- rbinom(1, 1, ps[a])

    ns[a] <- ns[a] + 1
    ss[a] <- ss[a] + r

    as[i] <- a
    rs[i] <- r
  }

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

}
```

```
# Define success probabilities
ps <- c(0.6, 0.4)

# Number of trials
n <- 1000

thompson_results <- thompson.bernoulli(ps, n)

epsilon_results <- epsilon.decreasing(ps, C = 1, k = 1, n)

# Compute average reward for both strategies
thompson_avg_reward <- mean(thompson_results$rs)
epsilon_avg_reward <- mean(epsilon_results$rs)


cat("Thompson Sampling - Average Reward:", thompson_avg_reward, "\n")
```

```
## Thompson Sampling - Average Reward: 0.585
```

```
cat("Epsilon-Decreasing - Average Reward:", epsilon_avg_reward, "\n")
```

```
## Epsilon-Decreasing - Average Reward: 0.601
```

Since we are dealing with a two-armed Bernoulli bandit problem, with success probabilities $p_1 = 0.6$ and $p_2 = 0.4$ for arms 1 and 2 respectively, the optimal strategy would be to always pull the best arm, that is arm 1, and achieve an average reward of 0.6. Testing our implementation using the provided probabilities, we see that both strategies converge to an average reward of approximately 0.6. Since the output of these strategies are random, we run the experiment 100 times to obtain a more reliable estimate of their average performance and reduce the impact of randomness in any single run as follows.

```
# Repeat experiments multiple times for stability
num_experiments <- 100
thompson_rewards <- numeric(num_experiments)
epsilon_rewards <- numeric(num_experiments)

for (i in 1:num_experiments) {
  thompson_rewards[i] <- mean(thompson.bernoulli(ps, n)$rs)
  epsilon_rewards[i] <- mean(epsilon.decreasing(ps, C = 1, k = 1, n)$rs)
}

cat("Thompson Sampling - Average Reward over", num_experiments, "runs:", mean(thompson_rewards), "\n")
```

```
## Thompson Sampling - Average Reward over 100 runs: 0.59552
```

```r
cat("Epsilon-Decreasing - Average Reward over", num_experiments, "runs:", mean(epsilon_rewards), "\n")
```

```
## Epsilon-Decreasing - Average Reward over 100 runs: 0.57807
```

Running the experiment repeatedly for 100 runs, we see that the convergence to 0.6 is consistent, confirming the fact that both strategies learn to favour the optimal arm over time.

8 | 8

Part (b)

To describe the behaviour of the ϵ-decreasing strategy, we observe how the average reward evolves as the strategy is applied over multiple runs. In the following code block, we calculate, at each step i where $i \in \{1, 2, \dots n\}$ for each simulation (`num_simulation`), the cumulative total of all rewards up to that step and and divide that by the number of steps taken. This gives us the cumulative average up to that specific step. That is,

$$\bar{r}_i^{(m)} = \frac{1}{i} \sum_{j=1}^{i} r_j^{(m)} \quad ,$$

where $r_j^{(m)}$ denotes the reward obtained at step j in the m^{th} run and $\bar{r}_i^{(m)}$ denotes the cumulative average reward up to step i. Once we have the cumulative averages for all M runs, we compute the overall average reward at each step i as

$$\bar{r}_i = \frac{1}{M} \sum_{m=1}^{M} \bar{r}_i^{(m)}$$

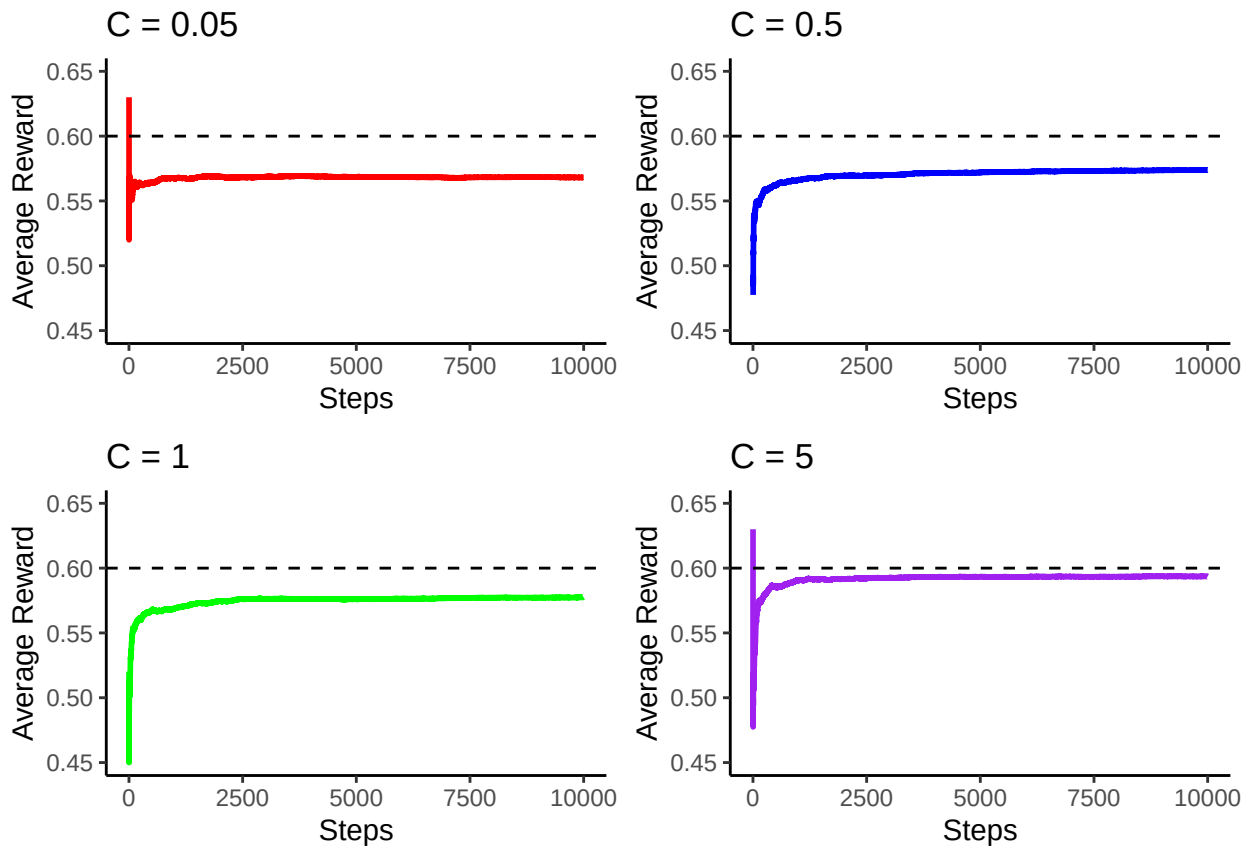
The code below computes this average for different values of C while keeping $t = 1$ which defines the ϵ sequence as $\epsilon_n = \{1, Cn^{-1}\}$. The results are then plotted to show how the different values of C affect the learning process.

```r
# Set parameters
ps <- c(0.6, 0.4)  # True success probabilities
n <- 10000         # Number of steps
C_values <- c(0.05, 0.5, 1, 5)
num_simulations <- 100  # Number of simulations for stability

# Function to compute average rewards over multiple simulations
run_simulations <- function(ps, C, k, n, num_simulations) {
  avg_rewards <- matrix(0, nrow = n, ncol = num_simulations)

  for (sim in 1:num_simulations) {
    results <- epsilon.decreasing(ps, C, k, n)
    avg_rewards[, sim] <- cumsum(results$rs) / (1:n)  # Compute cumulative average reward
  }

  # Compute the mean average reward across simulations
  mean_avg_rewards <- rowMeans(avg_rewards)
  return(mean_avg_rewards)
}


mean_avg_rewards_C1_k1 <- run_simulations(ps, C = 0.05, k = 1, n, num_simulations)
mean_avg_rewards_C2_k1 <- run_simulations(ps, C = 0.5, k = 1, n, num_simulations)
mean_avg_rewards_C3_k1 <- run_simulations(ps, C = 1, k = 1, n, num_simulations)
mean_avg_rewards_C4_k1 <- run_simulations(ps, C = 5, k = 1, n, num_simulations)
```

```
plot_rewards <- function(mean_rewards, C_value, color) {
  ggplot(data = data.frame(Steps = 1:n, Reward = mean_rewards), aes(x = Steps, y = Reward)) +
    geom_line(color = color, size = 1) +
    geom_hline(yintercept = max(ps), linetype = "dashed", color = "black") +
    labs(title = paste("C =", C_value), x = "Steps", y = "Average Reward") +
    ylim(0.45, 0.65) +
    theme_classic()
}

# Create the four plots
plot1 <- plot_rewards(mean_avg_rewards_C1_k1, 0.05, "red")
```

```
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use `linewidth` instead.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where this warning was
## generated.
```

```
plot2 <- plot_rewards(mean_avg_rewards_C2_k1, 0.5, "blue")
plot3 <- plot_rewards(mean_avg_rewards_C3_k1, 1, "green")
plot4 <- plot_rewards(mean_avg_rewards_C4_k1, 5, "purple")

# Arrange plots in a 2x2 grid
grid.arrange(plot1, plot2, plot3, plot4, ncol = 2, nrow = 2)
```



The plot shows the average reward over time for different values C in the ϵ-decreasing strategy where $\epsilon_n = \min\{1, Cn^{-1}\}$. The horizontal line at $\max\{\mu_1, \mu_2\} = 0.6$, where $\mu_1 = 0.6$ and $\mu_2 = 0.4$ represents the optimal reward threshold that we expect the algorithm to approach as n increases. First note that the spike at $n = 1$ that we can see for all values of C in the plot, is due the the fact that the arm is only played once. Since the reward is either 1 (success) or 0 (failure), some simulations randomly obtain an average reward of 1 as in our case above. However, this is not relevant and we focus on the more general trend for $n > 1$.

For small C such as $C = 0.05$ denoted by the red line in the above figure, it is clear to see that the there is very little exploration, because $\epsilon_n = \min(1, 0.05n^{-1})$ decreases very quickly and instead the algorithm begins to exploit before it has gathered enough information leading to a quick stabilisation at a suboptimal reward. For moderate C value such as 0.5 and 1 (blue and green lines respectively), we see that these values strike a balance between exploration and exploitation, leading to a steady convergence towards the optimal reward. In contrast, a large value of C such as 5 (purple line) will encourage more exploration leading to slow but eventual convergence. However, this eventually reaches a higher reward at the cost of stabilising later than the other values of C.

This aligns well with the theoretical result in lectures which states that if $\{\epsilon_i\}$ is decreasing slowly enough, then

$$\mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n} R_i\right] \to \mu_1.$$

In summary, the ϵ-decreasing strategy with $\epsilon_n = \min\{1, Cn^{-1}\}$ exhibits a trade-off between exploration and exploitation which is determined by the values of C. Small C values lead to rapid exploration but risk premature convergence to a suboptimal arm, while large C values encourage more exploration ensuring eventually convergence to the optimal arm at the cost of slower stabilisation.

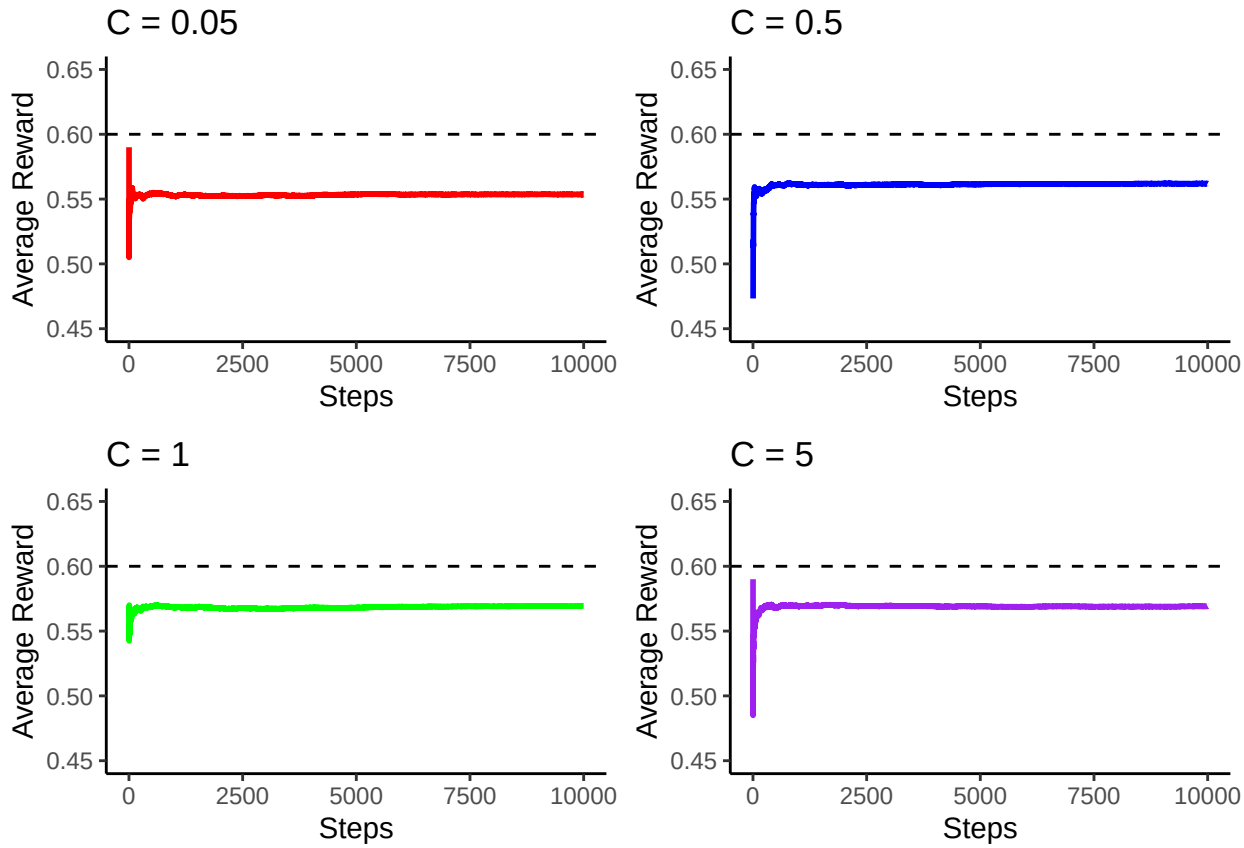
8/9

Part (c) Why is c) worse than b)? Since b) is asymptotically optimal.

```
# Set parameters
ps <- c(0.6, 0.4)  # True success probabilities
n <- 10000         # Number of steps
C_values <- c(0.05, 0.5, 1, 5, 100)  # New C values (add C=100 - to show extreme)
num_simulations <- 100  # Number of simulations for stability

# Function to compute average rewards over multiple simulations
run_simulations <- function(ps, C, k, n, num_simulations) {
  avg_rewards <- matrix(0, nrow = n, ncol = num_simulations)

  for (sim in 1:num_simulations) {
    results <- epsilon.decreasing(ps, C, k, n)
    avg_rewards[, sim] <- cumsum(results$rs) / (1:n)  # Compute cumulative average reward
  }

  # Compute the mean average reward across simulations
  mean_avg_rewards <- rowMeans(avg_rewards)
  return(mean_avg_rewards)
}


mean_avg_rewards_C1_k2 <- run_simulations(ps, C = 0.05, k = 2, n, num_simulations)
mean_avg_rewards_C2_k2 <- run_simulations(ps, C = 0.5, k = 2, n, num_simulations)
mean_avg_rewards_C3_k2 <- run_simulations(ps, C = 1, k = 2, n, num_simulations)
mean_avg_rewards_C4_k2 <- run_simulations(ps, C = 5, k = 2, n, num_simulations)
mean_avg_rewards_C5_k2 <- run_simulations(ps, C = 100, k = 2, n, num_simulations)
```

```r
plot_rewards <- function(mean_rewards, C_value, color) {
  ggplot(data = data.frame(Steps = 1:n, Reward = mean_rewards), aes(x = Steps, y = Reward)) +
    geom_line(color = color, size = 1) +
    geom_hline(yintercept = max(ps), linetype = "dashed", color = "black") +
    labs(title = paste("C =", C_value), x = "Steps", y = "Average Reward") +
    ylim(0.45, 0.65) +
    theme_classic()
}

# Create the four plots
plot1 <- plot_rewards(mean_avg_rewards_C1_k2, 0.05, "red")
plot2 <- plot_rewards(mean_avg_rewards_C2_k2, 0.5, "blue")
plot3 <- plot_rewards(mean_avg_rewards_C3_k2, 1, "green")
plot4 <- plot_rewards(mean_avg_rewards_C4_k2, 5, "purple")
grid.arrange(plot1, plot2, plot3, plot4, ncol = 2, nrow = 2)
```

```
## Warning: Removed 1 row containing missing values or values outside the scale range
## (`geom_line()`).
```

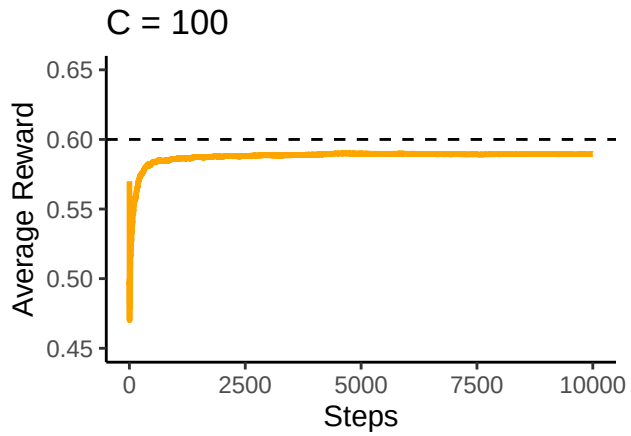


Using the fact that we have allowed the rate parameter variable, it allows us to set `k=2` in `epsilon.decreasing` to describe the behaviour of ϵ-decreasing when the sequence is ϵ_n is defined by $\epsilon_n = min\{1, Cn^{-2}\}$. Implementing this change yields the above plots four plots which show the average reward for varying values of C. In comparison to when ϵ_n was defined by $\epsilon_n = min\{1, Cn^{-1}\}$, the probability of exploration decreases at a much faster rate leading to rapid exploitation but at a cost of higher risk of premature convergence to a suboptimal arm. This is why, in the above plots, we can noticeably see that even for the case where $C = 5$,

we have not converged close to the optimal reward line. To compensate for the exploration probability decreasing rapidly, we require a much larger value of C to achieve the convergence to the optimal reward of 0.6. On the other hand, for small values of C, we can see that, as before, the algorithm explores less and exploits more so is stuck at a suboptimal reward and this effect is more pronounced in figure compared to the previous one.

We illustrate below the effect of using a much larger value of C when $\epsilon_n = min\{1, Cn^{-2}\}$.

```
plot5 <- plot_rewards(mean_avg_rewards_C5_k2, 100, "orange")
plot5
```

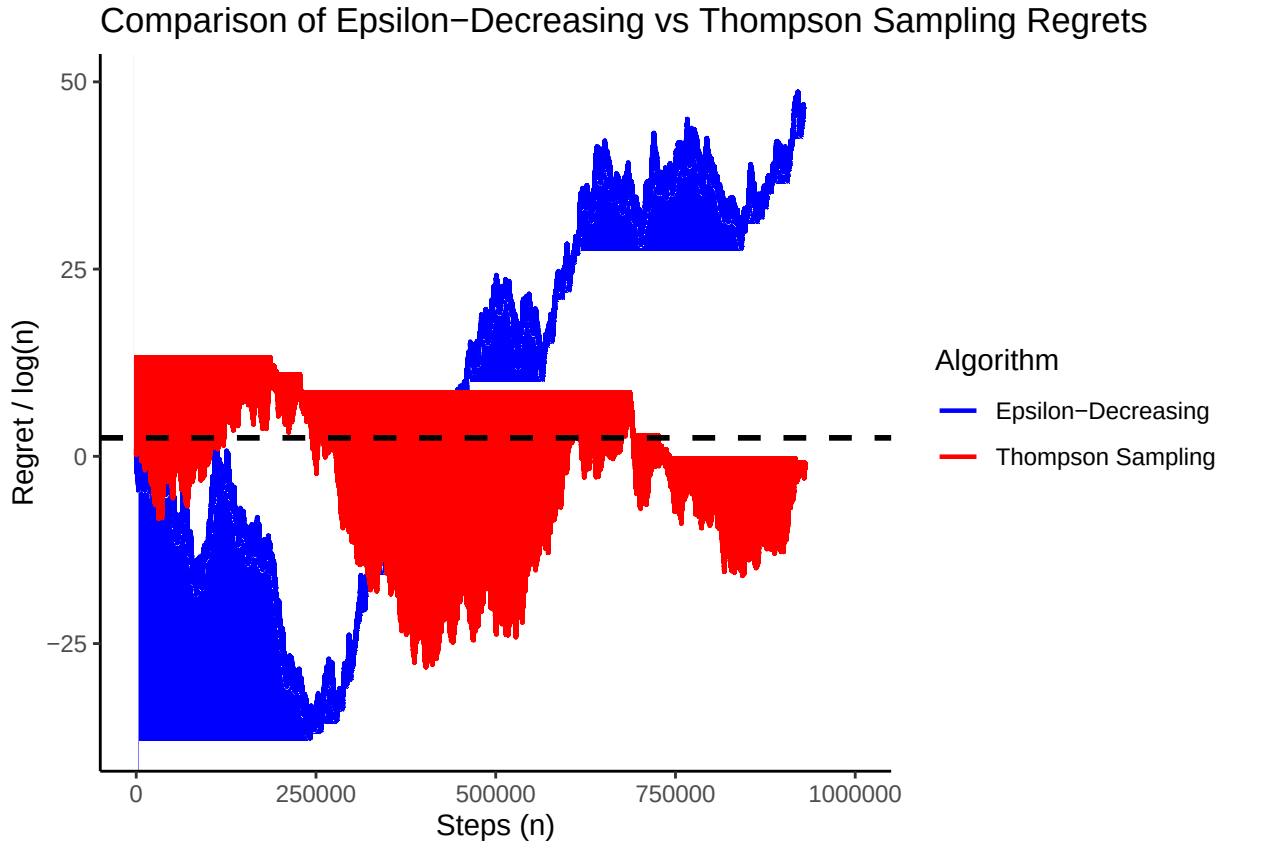


As shown above, a much larger value of C such as $C = 100$ was necessary for us to get a convergence close to 0.6 contrasting the case where ϵ_n is defined by $\epsilon_n = min\{1, Cn^{-1}\}$ where we only need a value of $C = 5$ to achieve convergence.

Part (d)

```
set.seed(99)
# Set parameters
ps <- c(0.6, 0.4)  # True success probabilities
n <- 1000000       # Number of steps
mu1 <- max(ps)     # Best arm's success probability

# Simulate regret for epsilon-decreasing
epsilon_rewards <- epsilon.decreasing(ps, n, C = 1, k = 1)$rs
epsilon_regret <- (1:n) * mu1 - cumsum(epsilon_rewards)
epsilon_normalised <- epsilon_regret / log(1:n)

# Simulate regret for Thompson sampling
thompson_rewards <- thompson.bernoulli(ps, n)$rs
thompson_regret <- (1:n) * mu1 - cumsum(thompson_rewards)
thompson_normalised <- thompson_regret / log(1:n)

# Create a data frame for plotting
df <- data.frame(
  Steps = 1:n,
  Epsilon = epsilon_normalised,
  Thompson = thompson_normalised
)
```

```r
# Plot the results
ggplot(df, aes(x = Steps)) +
  geom_line(aes(y = Epsilon, color = "Epsilon-Decreasing"), size = 0.8) +
  geom_line(aes(y = Thompson, color = "Thompson Sampling"), size = 0.8) +
  geom_hline(yintercept = (mu1 - min(ps))
             / (ps[2] * log(ps[2] / ps[1]) + (1 - ps[2]) * log((1 - ps[2]) / (1 - ps[1]))),
             linetype = "dashed", color = "black", size = 1) +
  labs(
    title = "Comparison of Epsilon-Decreasing vs Thompson Sampling Regrets",
    x = "Steps (n)",
    y = "Regret / log(n)",
    color = "Algorithm"
  ) +
  theme_classic() +
  scale_color_manual(values = c("Epsilon-Decreasing" = "blue", "Thompson Sampling" = "red"))
```



To compare both methods, we plot a graph of realised regret / $\log(n)$ against n to gain an understanding of the losses. The regret is defined as the expected lost reward and is defined as

$$\mathcal{R}(n) = \mathbb{E}[\sum_{i=1}^{n}(\mu_1 - R_i)]$$

where μ_1 is denotes the optimal arm.

The dashed black line denotes the constant $\frac{\mu_1 - \mu_2}{D_{KL}(\mu_2||\mu_1)}$ which denotes the best possible regret bound. We can see from the plot that while both algorithms fluctuate, significantly, Thompson sampling often hovers closer to this line and compared to ϵ-decreasing reflecting its near optimal balance between exploitation and exploration

while ϵ-decreasing shows a more volatile pattern which agrees with the theory seen in lectures. Overall, Thompson sampling is more consistent at achieving the theoretical lower bound compared to ϵ-decreasing.

Question 3

Part (a)

9/9

```
knn.regression.test <- function(k, train.X, train.Y, test.X, test.Y, distances) {

  # Initialise a vector to store the estimates
  estimates <- numeric(nrow(test.X))

  # Loop through each test point
  for (i in 1:nrow(test.X)) {
    # Calculate distances between the current test point and all training points
    # drop = FALSE ensures that subsetting a matrix preserves its dimensions
    dists <- apply(train.X, 1, function(train.point) {
      distances(matrix(train.point, nrow = 1), test.X[i, , drop = FALSE])
    })


    nearest.indices <- order(dists)[1:k]
    nearest.dists <- dists[nearest.indices]
    nearest.targets <- train.Y[nearest.indices]

    weights <- 1 / nearest.dists
    estimates[i] <- weighted.mean(nearest.targets, weights)
  }

  # Print the sum of squared errors
  return(sum((test.Y - estimates)^2))
}
```

Part (b)

6/6

We'll test our function to do kNN regression on the following toy datasets using the `distances.l1` function from Lab 6.

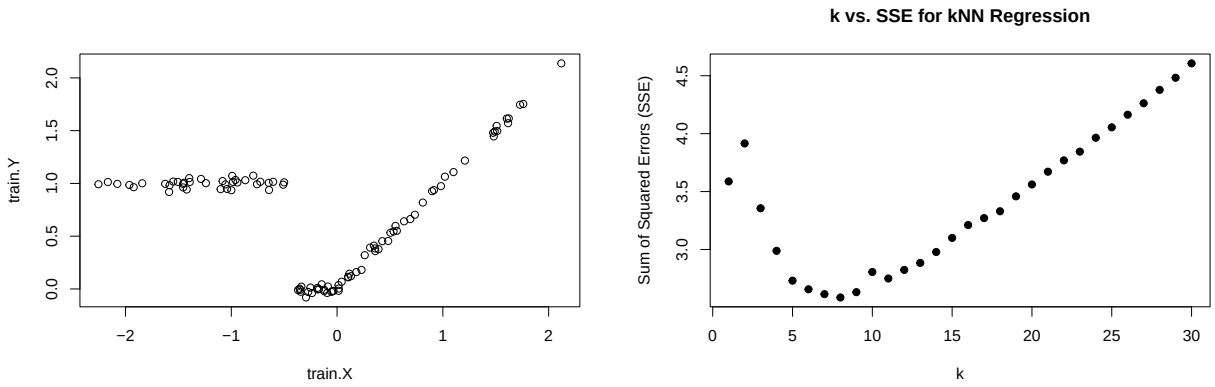
Toy Dataset 1:

```
# distances.l1 function from Lab 6
distances.l1 <- function(X,W) {
  apply(W, 1, function(p) apply(X,1,function(q) sum(abs(p-q))))
}


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
```

```r
knn.regression.test(k,train.X,train.Y,test.X,test.Y,distances.l1)
```

```
## [1] 3.91539
```

```r
# Define a range of k values to test
k_values <- 1:30

# Create a vector to store SSE values
sse_values <- numeric(length(k_values))

# Test the function for each k value
for (i in 1:length(k_values)) {
  k <- k_values[i]
  sse_values[i] <- knn.regression.test(k, train.X, train.Y, test.X, test.Y, distances.l1)
}

# Plot k vs. SSE
plot(k_values, sse_values, pch = 19,
     xlab = "k", ylab = "Sum of Squared Errors (SSE)",
     main = "k vs. SSE for kNN Regression")
```



The figure on the left is a scatter plot that displays the training dataset (`train.X` vs. `train.Y`). The dataset is non-linear with a nearly constant region for `train.X < -0.5` where `train.Y` is around 1. Then there is a sharp transition from around `train.X = 0`, where `train.Y` suddenly drops to 0 and then follows a more linear increasing trend for `train.X > 0`.

The figure on the right is a plot that shows the effect on the sum of squared errors of varying k in the kNN algorithm. It is clear to see that the curve initially decreases, reaches a minimum and then starts increasing again. For very small k, i.e. $k = 1, \ldots, 4$, the model is highly sensitive to the noise in the training data, which leads to overfitting. As a result, the model performs well on the training data but predictions on the test data set are poor and this is evident from the high total squared error. For values of k in the range of $k = 5, \ldots, 10$ the SSE is minimised because the model is able to balance capturing the true relationship in the data and the sensitivity to the noise, leading to a lower SSE. For large k, such as $k = 25$, the model averages 25 nearest neighbours. This leads to the model being too simplistic and thus, fails to capture the true relationship in the dataset, leading to underfitting.

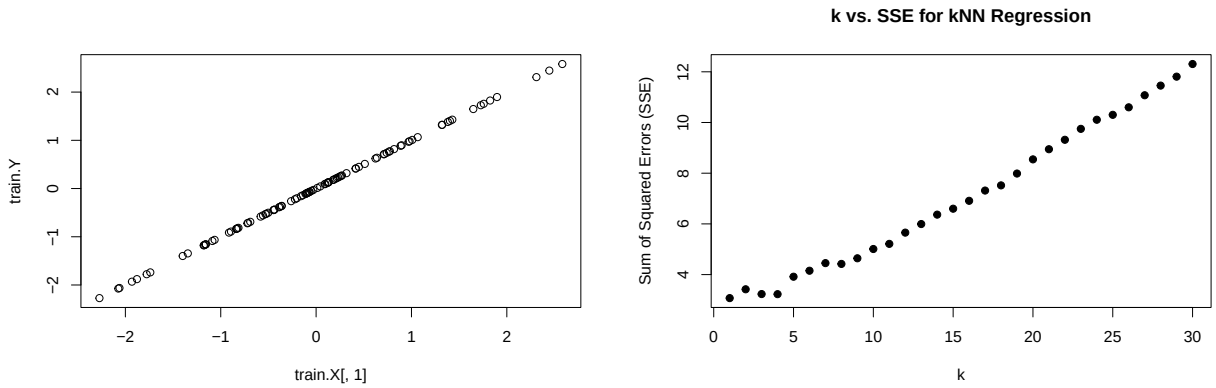
Toy Dataset 2:

```r
set.seed(100)
train.X <- matrix(rnorm(200),100,2)
train.Y <- train.X[,1]
plot(train.X[,1], train.Y)
```

```r
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
```

```r
# Define a range of k values to test
k_values <- 1:30

# Create a vector to store SSE values
sse_values <- numeric(length(k_values))

# Test the function for each k value
for (i in 1:length(k_values)) {
  k <- k_values[i]
  sse_values[i] <- knn.regression.test(k, train.X, train.Y, test.X, test.Y, distances.l1)
}

# Plot k vs. SSE
plot(k_values, sse_values, pch = 19,
     xlab = "k", ylab = "Sum of Squared Errors (SSE)",
     main = "k vs. SSE for kNN Regression")
```



Similar to the previous toy dataset, the figure on the left displays the training dataset (`train.X[,1]` vs `train.Y`). Although, `train.X` has two features, the reason we are only plotting `train.X[,1]` is because `train.Y` is directly derived from `train.X[,1]`. As a result, there is no influence from `train.X[,2]` on `train.Y` and it is evident from the plot that the dataset exhibits a perfect linear relationship.

The figure on the right is a plot that shows the effect on the sum of squared errors of varying k in the kNN algorithm. For this dataset, the relationship between Y and X_1 is linear, so a small k value captures the trend reasonably well. This is why we can see that the total sum of squared errors is minimised for the lowest k values and then continues to increase in a linear fashion as k increases.

Part (c)

```r
distances.l2 <- function(X, W) {
  apply(W, 1, function(p) apply(X, 1, function(q) sqrt(sum((p - q)^2))))
}
```

```r
# Dataset Iowa downloaded from Moodle
Iowa <- read.table("Iowa.txt", header = TRUE, sep = "\t")
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
```

We define a new `knn.regression.predict` function to return the predicted estimates instead of the total sum of squared errors. **Note**: The only change to the `knn.regression.test` function in part **(a)** is to the final line of the code, where `return(sum((test.Y - estimates)^2))` has been replaced with `return(estimates)`.

```r
knn.regression.predict <- function(k, train.X, train.Y, test.X, test.Y, distances) {

  # Initialise a vector to store the estimates
  estimates <- numeric(nrow(test.X))

  # Loop through each test point
  for (i in 1:nrow(test.X)) {
    # Calculate distances between the current test point and all training points
    # drop = FALSE ensures that subsetting a matrix preserves its dimensions
    dists <- apply(train.X, 1, function(train.point) {
      distances(matrix(train.point, nrow = 1), test.X[i, , drop = FALSE])
    })


    nearest.indices <- order(dists)[1:k]
    nearest.dists <- dists[nearest.indices]
    nearest.targets <- train.Y[nearest.indices]

    weights <- 1 / nearest.dists
    estimates[i] <- weighted.mean(nearest.targets, weights)
  }

  # Print the sum of squared errors
  return(estimates)
}
```

```r
# Generate predictions
predictions <- knn.regression.predict(k, train.X, train.Y, test.X, test.Y, distances.l2)

# Extract years from test.X
years <- test.X[, 1]

# Create a data frame of years and predictions
predictions_df <- data.frame(
  Year = years,
  Predicted_Yield = round(predictions,1)
)

# Print the data frame
kable(predictions_df, caption = "Predicted Yields for Test Years", align = "c")
```

Table 4: Predicted Yields for Test Years

	Year	Predicted_Yield
2	1931	36.4
4	1933	39.6
6	1935	47.2
8	1937	41.7
10	1939	47.4
12	1941	46.7
14	1943	53.5
16	1945	55.0
18	1947	55.5
20	1949	56.6
22	1951	55.5
24	1953	56.2
26	1955	59.2
28	1957	58.8
30	1959	64.4
32	1961	60.2

The above table shows the predicted yield in the years $1931, 1933, \ldots$ based on the data from $1932, 1934, \ldots$.

8 | 8

Part (d)

```
library(glmnet)
```

```
## Loading required package: Matrix
```

```
##
## Attaching package: 'Matrix'
```

```
## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
```

```
## Loaded glmnet 4.1-8
```

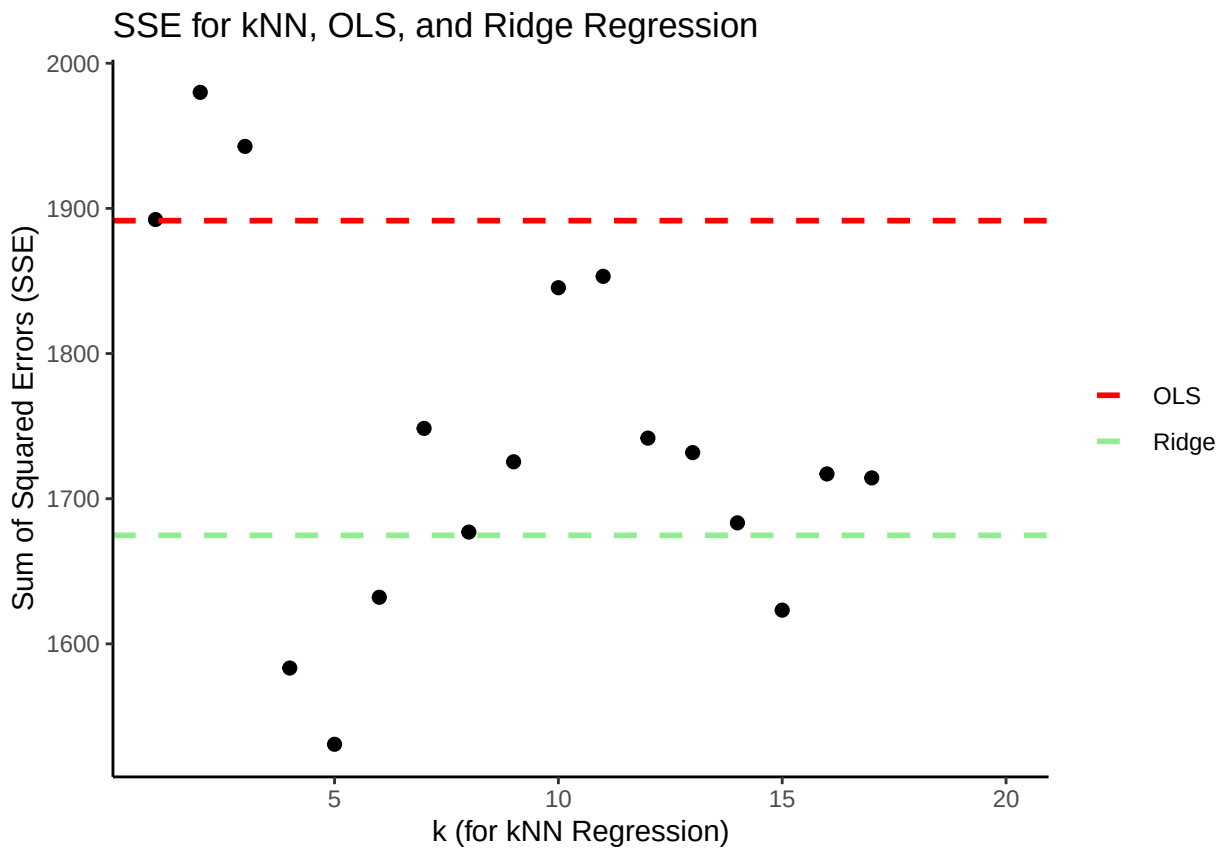
```
library(Matrix)

# Test kNN regression for different k values
k_values <- 1:20
sse_knn <- numeric(length(k_values))

for (i in 1:length(k_values)) {
  k <- k_values[i]
  sse_knn[i] <- knn.regression.test(k, train.X, train.Y, test.X, test.Y, distances.l2)
}

# Fit OLS regression
ols_model <- lm(train.Y ~ ., data = data.frame(train.X))
ols_predictions <- predict(ols_model, newdata = data.frame(test.X))
sse_ols <- sum((test.Y - ols_predictions)^2)

# Fit Ridge regression
```

```r
# Set alpha = 0 for ridge regression and use `cv.glmnet` to find the optimal lambda
ridge_model <- cv.glmnet(train.X, train.Y, alpha = 0)
```

```
## Warning: Option grouped=FALSE enforced in cv.glmnet, since < 3 observations per
## fold
```

```r
optimal_lambda <- ridge_model$lambda.min
ridge_predictions <- predict(ridge_model, newx = test.X, s = optimal_lambda)
sse_ridge <- sum((test.Y - ridge_predictions)^2)



knn_results <- data.frame(k = k_values, SSE = sse_knn)



ggplot() +
  geom_point(data = knn_results, aes(x = k, y = SSE), color = "black", size = 2) +
  geom_hline(aes(yintercept = sse_ols, color = "OLS"), linetype = "dashed", size = 1) +
  geom_hline(aes(yintercept = sse_ridge, color = "Ridge"), linetype = "dashed", size = 1) +
  labs(title = "SSE for kNN, OLS, and Ridge Regression",
       x = "k (for kNN Regression)",
       y = "Sum of Squared Errors (SSE)") +
  scale_color_manual(values = c("OLS" = "red", "Ridge" = "lightgreen")) +
  theme_classic() +
  theme(legend.title=element_blank())
```

```
## Warning: Removed 3 rows containing missing values or values outside the scale range
## (`geom_point()`).
```



The plot above provides a clear comparison of the performance of kNN regression, ordinary least squares (OLS) regression and ridge regression (RR). The red horizontal line representing OLS regression has a higher sum of squared errors (SSE) compared to the green line for ridge regression, indicating that ridge regression outperforms OLS. This is likely due to additional regularisation parameter λ in ridge regression to handle overfitting and thus leads to a lower SSE compared to OLS. The black points, representing the sum of squared errors for kNN regression for varying values of k, with most of the points lying between the ordinary least squares and ridge regression line, with only a handful points falling outside this band. This indicates that the performance of kNN regression is sensitive to the choice of k as for certain values of k it performs better than both OLS and RR, whilst for others it performs worse than both and thus the variability highlights the influence on the value of k to minimise the SSE in kNN regression. Overall, ridge regression is the best-performing method for this dataset, while kNN can be competitive with careful selection of the k value and OLS regression performs the worst.