ST340 Assignment 3

Anojan Mariathas (2244070), Jude Thomas (2210732)

Contributions

- Question 1: Jude took the lead in writing most of the code for the algorithms after thorough discussion with Anojan. Meanwhile, Anojan contributed by working on the proofs and explanations to the algorithms implemented.
- Question 2: Anojan took the lead in writing the majority of the code for the grid search algorithms while Jude constructed the justifications as to why the implementations worked as expected.

Question 1 - Gradient Descent

Part (a)

In this question we are asked to find a local maxima, using the gradient descent function provided below.

```
# Provided gradient descent code
gradient.descent <- function(f, gradf, x0,iterations=1000,eta=0.2) {
x<-x0
for (i in 1:iterations) {
cat(i,"/",iterations,": ",x," ",f(x),"\n")
x<-x-eta*gradf(x)
  }
  x
}
```

```
gradient.ascent <- function(f, df, x0, iterations=1000, eta=0.2){
  # create the negated versions of the gradient
  neg_df <- function(x) { -df(x) }
  gradient.descent(f, gradf = neg_df,x0 , iterations, eta)
}

# Test the function with the provided testing code

f <-function(x) { (1+x^2)^(-1) }
gradf<-function(x) { -2*x*(1+x^2)^(-2) }
gradient.ascent(f,gradf,3,40,0.5)
```

```
## 1 / 40 :  3   0.1
## 2 / 40 :  2.97   0.1018237
## 3 / 40 :  2.939207   0.1037459
## 4 / 40 :  2.907572   0.1057756
## 5 / 40 :  2.87504   0.1079231
## 6 / 40 :  2.841554   0.1101998
## 7 / 40 :  2.807046   0.1126189
## 8 / 40 :  2.771444   0.1151954
```

```
## 9 / 40 :  2.734667   0.1179467
## 10 / 40 :  2.696624   0.120893
## 11 / 40 :  2.657212   0.1240575
## 12 / 40 :  2.616317   0.1274679
## 13 / 40 :  2.573807   0.1311564
## 14 / 40 :  2.529532   0.1351619
## 15 / 40 :  2.483321   0.1395307
## 16 / 40 :  2.434974   0.144319
## 17 / 40 :  2.384258   0.1495956
## 18 / 40 :  2.330901   0.155446
## 19 / 40 :  2.274579   0.1619772
## 20 / 40 :  2.214901   0.1693254
## 21 / 40 :  2.151398   0.1776669
## 22 / 40 :  2.083488   0.1872336
## 23 / 40 :  2.010448   0.1983379
## 24 / 40 :  1.931361   0.2114095
## 25 / 40 :  1.845041   0.2270572
## 26 / 40 :  1.74992   0.2461708
## 27 / 40 :  1.643875   0.2701006
## 28 / 40 :  1.523947   0.300986
## 29 / 40 :  1.385889   0.3423852
## 30 / 40 :  1.223424   0.400518
## 31 / 40 :  1.027169   0.4866
## 32 / 40 :  0.7839563   0.6193532
## 33 / 40 :  0.4832319   0.8106927
## 34 / 40 :  0.165641   0.9732957
## 35 / 40 :  0.008728518   0.9999238
## 36 / 40 :  1.329848e-06   1
## 37 / 40 :  4.703997e-18   1
## 38 / 40 :  0   1
## 39 / 40 :  0   1
## 40 / 40 :  0   1

## [1] 0
```

From the output above, we can see that the `gradient.ascent` function works as we expect, correctly ascending to the maximum value of $(0, 1)$.

Part (b)

(i)

We want to show that the function $f : \mathbb{R}^2 \to \mathbb{R}$ given by

```
f <- function(x) (x[1]-1)^2 + 100*(x[1]^2-x[2])^2
```

has a unique minimum. In other words, we want to find the unique minimum of the function

$$f(x_1, x_2) = (x_1 - 1)^2 + 100(x_1^2 - x_2)^2.$$

We can note that the function f is the sum of two squared terms, namely, $(x_1 - 1)^2$ and $100(x_1^2 - x_2)^2$. Hence $f(x_1, x_2) \geq 0 \; \forall (x_1, x_2) \in \mathbb{R}^2$. Hence the function is minimised exactly when each term is equal to 0, as this is the only way for the overall sum to be 0, which is the smallest value that $f(x_1, x_2)$ can take. Solving

$$(x_1 - 1) = 0 \quad \text{and} \quad (x_1^2 - x_2) = 0,$$

yields $x_1 = 1$ and $x_2 = 1$. Since this solution is the only one that satisfies both equations, we can conclude that the given function has a unique minimum at $(1, 1)$.

(ii)

The following `R` code computes the partial derivatives of the function f defined in the previous part

```
gradf <- function(x){
  # calculate each part of the gradient vector
  df_dx1 <- 2*(x[1] - 1) + 400*x[1]*(x[1]^2 - x[2])
  df_dx2 <- -200*(x[1]^2 - x[2])

  return(c(df_dx1, df_dx2))
}
```

To ensure that our implementation works as expected, we perform a sanity check as follows: since we know that the function attains its minimum at $(1, 1)$, we should see that the output from the `gradf` function is $(0, 0)$ at this point.

```
gradf(c(1,1))
```

```
## [1] 0 0
```

(iii)

We used the following 'grid search like' algorithm, to systematically test a broad range of values for η and the results, sorted by the Euclidean distance from the final output to the functions known minimum point, are shown in the table below.

```
# Systematically test a broad range of learning rates
etas <- seq(0.0001, 0.0009, by = 0.0001)
x0 <- c(3, 4)
iterations <- 1000

# Run gradient descent for each eta and store results
results <- lapply(etas, function(eta) {
  result <- gradient.descent(f, gradf, x0, iterations, eta)
  list(eta = eta, result = result, distance = sqrt(sum((result - c(1, 1))^2)))  # Fixed
})

# Convert results to a data frame
results_df <- do.call(rbind, lapply(results, function(res) {
  data.frame(
    eta = res$eta,
    x1 = res$result[1],
    x2 = res$result[2],
    distance_to_min = res$distance
  )
}))
```

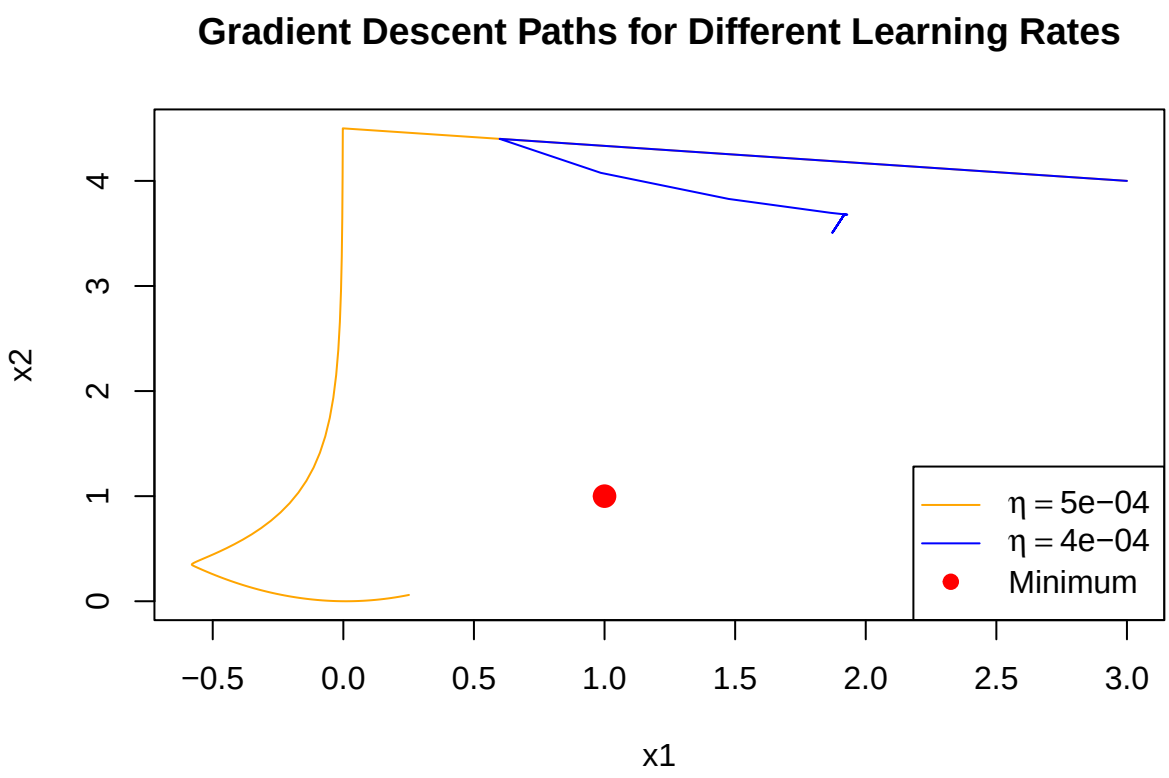
```
print(results_df[order(results_df$distance_to_min), ])
```

```
##     eta         x1         x2 distance_to_min
## 5 5e-04  0.2512386 0.06013062        1.201665
## 8 8e-04 -1.2909796 1.67434344        2.388164
## 4 4e-04  1.8718052 3.50582850        2.653153
## 7 7e-04 -1.5069512 2.27839455        2.814089
## 3 3e-04  1.9670623 3.87164322        3.030106
## 2 2e-04  2.0098808 4.04198688        3.205237
## 1 1e-04  2.0318085 4.13064016        3.296291
```

```
## 6 6e-04 -1.7072631 2.92204950        3.320173
## 9 9e-04        NaN        NaN             NaN
```

From the table above, we can see that the three best performing values of are $\eta = 0.0004, 0.0005$ and 0.0008. Based on these findings, we conducted a further refined grid search between these values and narrowed it down further to find that the best η had to lie somewhere in the in the interval $[0.0004, 0.0005]$. A further visual representation was produced to show the path taken to converge to the minimum by each of the η values and confirm that $\eta \in [0.0004, 0.0005]$.

```r
# Function to track the optimisation path
gradient.descent.track <- function(f, gradf, x0, iterations, eta) {
  x <- matrix(NA, nrow = iterations + 1, ncol = 2)
  x[1, ] <- x0
  for (i in 1:iterations) {
    x[i + 1, ] <- x[i, ] - eta * gradf(x[i, ])
  }
  x
}

# Generate paths for both learning rates
path <- gradient.descent.track(f, gradf, c(3, 4), iterations = 1000, eta = 0.0005)
path2 <- gradient.descent.track(f, gradf, c(3, 4), iterations = 1000, eta = 0.0004)

# Create a base R plot
plot(path[, 1], path[, 2], type = "l", col = "orange",
     xlab = "x1", ylab = "x2",
     main = "Gradient Descent Paths for Different Learning Rates",
     xlim = range(c(path[, 1], path2[, 1])),  # Set x-axis limits
     ylim = range(c(path[, 2], path2[, 2])))  # Set y-axis limits

# Add the second path
lines(path2[, 1], path2[, 2], col = "blue")

# Add the true minimum point
points(1, 1, col = "red", pch = 19, cex = 1.5)

# Add a legend
legend("bottomright",  legend = c(expression(eta == 0.0005), expression(eta == 0.0004), "Minimum"),
       col = c("orange", "blue", "red"), lty = c(1, 1, NA), pch = c(NA, NA, 19))
```



From the plot, we can see that the orange path, $\eta = 0.0005$ shows a more curved trajectory, initially moving away from the minimum before somewhat correcting itself. The blue path $\eta = 0.0004$ has a more controlled descent and looks on track to move towards the minimum, if we were to increase the number of iterations. Hence, we can safely conclude that η lies between these two values.

Using the information deduced up to this point, further manual testing was done to a greater precision and we found that to a suitable degree an optimal η value is 0.0004992. So we use this η to run the gradient descent algorithm for 1000 iterations and assess its performance.

```
x0 = c(3,4)
result <- gradient.descent(f, gradf, x0, iterations = 5000, eta = 0.0004992)
# Create a data frame with the final iteration details
final_result <- data.frame(
  Iteration = 5000,
  x1 = result[1],
  x2 = result[2],
  Function_Value = f(result)
)
```

```
# Print the final line of the algorithm
kable(final_result, caption = "Final Iteration Results of Gradient Descent")
```

Table 1: Final Iteration Results of Gradient Descent

Iteration	x1	x2	Function_Value
5000	1.007532	1.015151	5.68e-05

Part (c)

We implement a gradient descent with momentum algorithm below using the formulas and update rule seen in lectures.

```
gradient.descent.momentum <- function(f, gradf, x0, iterations = 1000, eta = 0.001, alpha = 0.9) {
  x <- x0
  x_prev <- x0  # Initialize previous x

  for (i in 1:iterations) {
    #This line is commented out (for now) to simplify the compilation process
    # cat(i,"/",iterations,": ",x," ",f(x),"\n")
    g <- gradf(x)
    x_new <- x - eta * g + alpha * (x - x_prev)
    x_prev <- x
    x <- x_new
  }
  return(x)
}
```

We fix our η to the value found in part **(b) (iii)** and systematically evaluate a range of momentum parameters, α, in a grid search type fashion. The α values cover a broad range ranging from 0.01 to 0.99 in increments of 0.00001. As before, we measure performance by measuring the Euclidean distance to the known minimum. All results, including the fixed η, the tested α, and the corresponding distance, are compiled into a data frame. The optimal hyperparameter is then selected as the one with the smallest distance, effectively fine-tuning the momentum parameter while leveraging the previously determined optimal learning rate.

```
eta_fixed <- 0.0004992
alphas <- seq(0.01, 0.99, by = 0.00001)
# Create the grid using the fixed eta
grid <- expand.grid(eta = eta_fixed, alpha = alphas)

# Run grid search over the grid
results.m <- lapply(1:nrow(grid), function(i) {
  params <- grid[i, ]
  result.m <- gradient.descent.momentum(
    f, gradf, x0 = c(3, 4),
    eta = params$eta,
    alpha = params$alpha,
    iterations = 100
  )
  # Calculate the Euclidean distance from the target (1, 1)
  distance <- sqrt(sum((result.m - c(1, 1))^2))
  # Return a named vector of parameters and the distance
  c(eta = params$eta, alpha = params$alpha, distance = distance)
})

# Convert results to a data frame directly (each element is a named vector)
results_df <- as.data.frame(do.call(rbind, results.m))

# Convert columns to numeric (they might be factors or characters)
results_df$eta <- as.numeric(as.character(results_df$eta))
results_df$alpha <- as.numeric(as.character(results_df$alpha))
```

```
results_df$distance <- as.numeric(as.character(results_df$distance))

# Find and print the best hyper-parameters (i.e., the one with minimum distance)
best_params <- results_df[which.min(results_df$distance), ]
print(best_params)
```

```
##             eta   alpha    distance
## 52266 0.0004992 0.53265 0.001503442
```

This process yielded an optimal α value of 0.53265. We now run the gradient descent with momentum algorithm using our optimised values of η and α. (Note, we have redefined the original `gradient.descent.momentum` function to now include the line to print out the iterations. This is the only change between the two functions and was done to ensure that we could visualise the convergence.

```
gradient.descent.momentum.2 <- function(f, gradf, x0, iterations = 1000, eta = 0.001, alpha = 0.9) {
  x <- x0
  x_prev <- x0  # Initialize previous x

  for (i in 1:iterations) {
    cat(i,"/",iterations,": ",x," ",f(x),"\n")
    g <- gradf(x)
    x_new <- x - eta * g + alpha * (x - x_prev)
    x_prev <- x
    x <- x_new
  }
  return(x)
}
```

```
x0 = c(3,4)
result.m <- gradient.descent.momentum.2(f, gradf, x0, iterations = 50, eta = 0.0004992, alpha = 0.53265)
final_result.m <- data.frame(
  Iteration = 50,
  x1 = result.m[1],
  x2 = result.m[2],
  Function_Value = f(result.m)
)
```

```
kable(final_result.m, caption = "Final Iteration Results of Gradient Descent with Momentum")
```

Table 2: Final Iteration Results of Gradient Descent with Momentum

Iteration	x1	x2	Function_Value
50	1.000686	1.001374	5e-07

From the output above, we can see that the algorithm with momentum has noticeably converged in 50 iterations compared to the 5000 iterations that were required in the previous implementation without momentum, to get to a similar convergence value. Thus, it highlights the effectiveness of the gradient descent with momentum.

Question 2 - Support Vector Machines

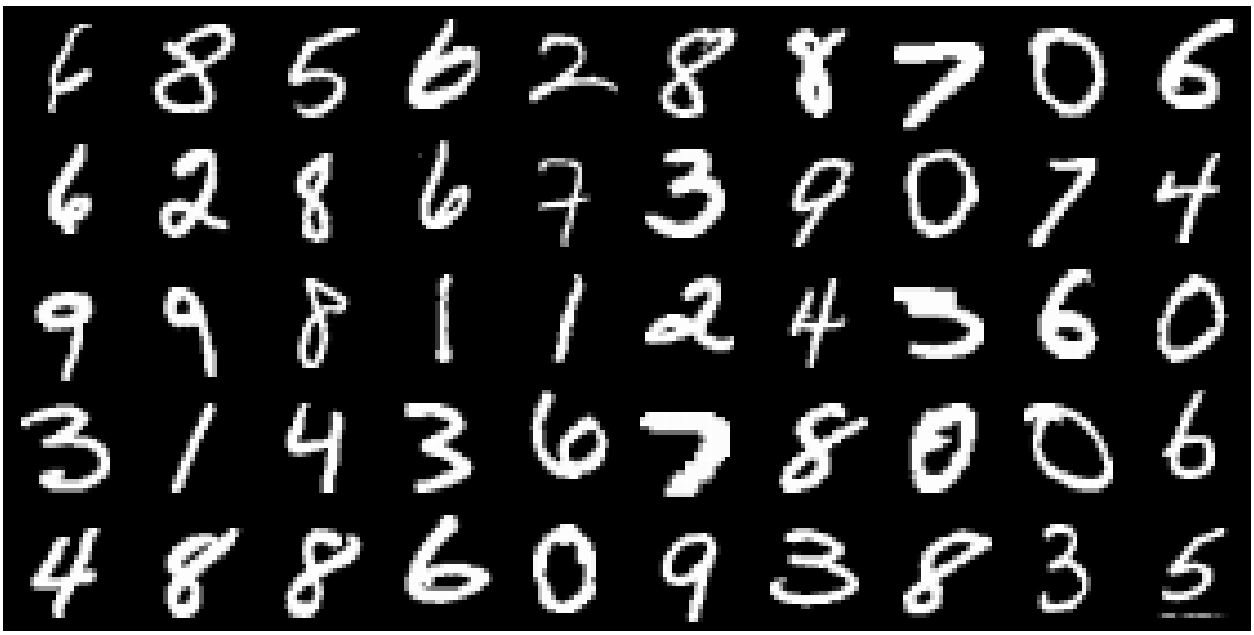
Part (a)

We load the MNIST dataset using the code provided

```
load("mnist.tiny.RData")
train.X=train.X/255
test.X=test.X/255
```

and display some digits below

```
library(grid)
grid.raster(array(aperm(array(train.X[1:50,],c(5,10,28,28)),c(4,1,3,2)),c(140,280)), interpolate=FALSE)
```



```
library(e1071)
```

Linear Kernel

The linear kernel is defined by

$$K(\boldsymbol{x}, \boldsymbol{x'}) = \boldsymbol{x} \cdot \boldsymbol{x'}$$

Since this kernel has no additional hyperparameters, the only parameter to optimise for assessing its accuracy is the cost parameter C, for the SVM optimisation process. To determine the optimal value of C, we perform a grid search using 3-fold cross validation over a wide range of C values from very small e.g. 0.001 to very large e.g. 500.

```
set.seed(12345)
cost_values <- c(0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 50, 100, 500)
```

```r
# Store cross-validation results
linear_accuracy <- data.frame(C = cost_values, Accuracy = NA)

# Perform grid search with 3-fold cross-validation
for (i in 1:length(cost_values)) {
  model <- svm(train.X, train.labels,
               type = "C-classification",
               kernel = "linear",
               cost = cost_values[i],
               cross = 3)

  linear_accuracy$Accuracy[i] <- model$tot.accuracy
}

linear_accuracy
```

```
##        C Accuracy
## 1  1e-03     64.2
## 2  5e-03     84.9
## 3  1e-02     87.5
## 4  5e-02     86.4
## 5  1e-01     86.8
## 6  5e-01     88.0
## 7  1e+00     86.0
## 8  5e+00     85.4
## 9  1e+01     86.5
## 10 5e+01     88.2
## 11 1e+02     86.6
## 12 5e+02     86.1
```

```r
# Find the best cost value
optimum_cost <- linear_accuracy$C[which.max(linear_accuracy$Accuracy)]
optimum_cost
```

```
## [1] 50
```

```r
best_linear_accuracy <- max(linear_accuracy$Accuracy)
best_linear_accuracy
```

```
## [1] 88.2
```

From the table (and output) above, we can see that the highest accuracy (88.2%) is achieved at $c = 50$. The accuracy for the linear kernel remains fairly consistent (between approximately 85% to 88%) across a broad range of values for the cost hyperparameter suggesting that the linear kernel is stable and performs well without extreme tuning. Next, we'll evaluate the polynomial and RBF kernels to see if they offer any significant improvement over the linear kernel.

Polynomial Kernel

The polynomial kernel is defined by

$$K(\boldsymbol{x}, \boldsymbol{x'}) = (c + \gamma \boldsymbol{x} \cdot \boldsymbol{x'})^p \quad \text{where } \gamma \geq 0,\ c \geq 0 \text{ and } p = 2, 3, \ldots$$

Note that in our implementation, we fix the degree to $p = 2$ and set $c = 1$, meaning that the primary hyperparameters to optimise are the cost parameter C as before and γ which controls the scaling. We tune these parameters using a grid search with 3-fold cross validation to identify the best combination of these

parameters. To ensure that we test a broad range of values for C and γ, the values of these parameters have been chosen such that they cover both extremes.

```
gamma_values <- c(0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 50, 100, 500)
cost_values <- c(0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 50, 100, 500)

# Create a grid of all combinations of gamma and cost values
polynomial_accuracy <- expand.grid(gamma = gamma_values, cost = cost_values)

# Add a column to store the accuracy results
polynomial_accuracy$accuracy <- NA

# Perform grid search with 3-fold cross-validation using 2 for loops
for (i in 1:length(gamma_values)) {
  for (j in 1:length(cost_values)) {
    # Train the SVM model
    model <- svm(train.X, train.labels,
                 type = "C-classification",
                 kernel = "poly",
                 degree = 2,
                 coef0 = 1,
                 cost = cost_values[j],
                 gamma = gamma_values[i],
                 cross = 3)

    # Find the corresponding row in the parameter_grid
    row_index <- which(polynomial_accuracy$gamma == gamma_values[i] & polynomial_accuracy$cost == cost_

    # Store the accuracy in the parameter_grid
    polynomial_accuracy$accuracy[row_index] <- model$tot.accuracy
  }
}

polynomial_accuracy[which.max(polynomial_accuracy$accuracy), ]
```

```
##    gamma cost accuracy
## 59   100  0.1       90
```

```
head(polynomial_accuracy)
```

```
##   gamma  cost accuracy
## 1 0.001 0.001     10.6
## 2 0.005 0.001     12.2
## 3 0.010 0.001     14.9
## 4 0.050 0.001     20.9
## 5 0.100 0.001     60.1
## 6 0.500 0.001     88.8
```

```
tail(polynomial_accuracy)
```

```
##     gamma cost accuracy
## 139     1  500     88.8
## 140     5  500     88.0
## 141    10  500     88.7
## 142    50  500     88.0
## 143   100  500     88.1
```

```
## 144   500  500     89.2
```

From the grid search results above, we can see that for small values of γ and small values of c, the accuracy is very low, suggesting that the model could be underfitting when these parameters are too small, but in general for $\gamma \geq 0.1$ and $C \geq 0.1$, the accuracy is somewhat consistent (between approximately 86% to 89%). Furthermore, from the output above, it is clear that, for the polynomial kernel, the highest accuracy of 90% is achieved at $\gamma = 100$ and $C = 0.1$, and offers a marginal improvement to the accuracy of the linear kernel.

RBF Kernel

The RBF (Radial Basis Function) kernel is defined by $K(\boldsymbol{x}, \boldsymbol{x'}) = \exp(-\gamma||\boldsymbol{x} - \boldsymbol{x'}||^2)$, where $\gamma \geq 0$. The only hyperparameters to optimise are the cost parameter C and γ, which we tune using a grid search function with 3-fold cross validation to identify the best combination of these parameters. As before, to ensure a comprehensive search, a broad range of values for C and γ have been chosen.

```
gamma_values <- c(0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 50, 100, 500)
cost_values <- c(0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1, 5, 10, 50, 100, 500)

# Store cross-validation results in a matrix
# Rows correspond to gamma values, columns correspond to cost values
rbf_accuracy <- expand.grid(gamma = gamma_values, cost = cost_values)

rbf_accuracy$accuracy <- NA

for (i in 1:length(gamma_values)) {
  for (j in 1:length(cost_values)) {
    # Train the SVM model
    model <- svm(train.X, train.labels,
                 type = "C-classification",
                 kernel = "radial",  # RBF kernel
                 cost = cost_values[j],
                 gamma = gamma_values[i],
                 cross = 3)

    # Find the corresponding row in the parameter_grid
    row_index <- which(rbf_accuracy$gamma == gamma_values[i] & rbf_accuracy$cost == cost_values[j])

    # Store the accuracy in the parameter_grid
    rbf_accuracy$accuracy[row_index] <- model$tot.accuracy
  }
}

rbf_accuracy[which.max(rbf_accuracy$accuracy), ]
```

```
##    gamma cost accuracy
## 87  0.01    5       91
```

```
head(rbf_accuracy)
```

```
##   gamma  cost accuracy
## 1 0.001 0.001     12.2
## 2 0.005 0.001     12.2
## 3 0.010 0.001     10.7
## 4 0.050 0.001     12.1
## 5 0.100 0.001     10.9
## 6 0.500 0.001     12.2
```

```
print(rbf_accuracy[73:78, ])
```

```
##    gamma cost accuracy
## 73 0.001    1     76.6
## 74 0.005    1     87.0
## 75 0.010    1     89.4
## 76 0.050    1     88.0
## 77 0.100    1     50.6
## 78 0.500    1     10.9
```

```
tail(rbf_accuracy)
```

```
##     gamma cost accuracy
## 139     1  500     12.3
## 140     5  500     10.2
## 141    10  500     12.2
## 142    50  500     12.2
## 143   100  500     10.4
## 144   500  500     10.7
```

From the grid search results above, we can see that for small values of γ and small values of C, the accuracy is very low (below 20%), suggesting that the model could be underfitting when these parameters are small. The RBF kernel, achieves its highest accuracy of 91% when $\gamma = 0.01$ and $C = 5$, and offers a marginal improvement to the accuracy of the linear kernel and the polynomial kernel. Beyond this, the accuracy varies significantly which could be indicative of some potential overfitting. Hence, it is clear to see that the RBF kernel's performance is highly sensitive to the choice of γ and C and depends on finding the right balance between these two parameter values.

Comparing the three kernels, we observe that the linear kernel performs the worst, the polynomial kernel offers an improvement over the linear kernel, and the RBF kernel achieves the best performance. However, the difference between the polynomial and RBF kernels is marginal, making the polynomial kernel a strong alternative candidate. It must also be noted that the computational cost to tune parameters of the RBF and polynomial kernel is higher than that of the linear kernel and, as witnessed above, the RBF kernel is very sensitive to changes in these parameters requiring careful selection of the γ and C values. Therefore, while the RBF and polynomial kernel provide almost equally good results, their sensitivity and computational demands mean that we must proceed with caution when using them.

NB: Warning Messages

When running the above code, we have set `warning = FALSE` to prevent a long warning message. The message states: `Warning: Variable(s) '1' and '2' and '3' and ... Cannot scale data`. This warning indicates that some features in the dataset are constant across all observations, making the scaling ineffective. It informs us that these features do not provide any additional/useful information and can be removed to improve the efficiency of our code.

Part (b)

In order to write a grid search function that takes two lists, `log.C.range` and `log.gamma.range`, and for each pair of entries `(lc, lg)` attempts cross-validation with parameters `cost = exp(lc)` and `gamma = exp(lg)`, we define two `for` loops to find the best combination of the parameters C and γ. One of the loops will range over values of C and the other for the values of γ. Within the for loop we will compute and store the accuracy for each combination of parameters in a grid (matrix) form and then select which combination maximises the accuracy. Below is the construction of the RBF grid search function.

```
rbf_grid_search <- function(train.X, train.labels, log.C.range, log.gamma.range) {

  accuracy_matrix <- matrix(NA, nrow = length(log.C.range), ncol = length(log.gamma.range))

  for (i in 1:length(log.C.range)) {
    for (j in 1:length(log.gamma.range)) {
      cost <- exp(log.C.range[i])
      gamma <- exp(log.gamma.range[j])

      model <- svm(train.X, train.labels,
                   type = "C-classification",
                   kernel = "radial",
                   cost = cost,
                   gamma = gamma,
                   cross = 3)

      # Store the cross-validation accuracy in the matrix
      accuracy_matrix[i, j] <- model$tot.accuracy
    }
  }
  return(accuracy_matrix)
}
```

We take a coarse-to-fine search approach to try and find the best parameters for C and γ. This means that we initially start out with a wide range of parameters to test from and depending on the output we either narrow down or broaden the search. To begin with, we will set `log.C.range` and `log.gamma.range` to take values in the interval $[-3, 3]$. If the results show that the optimal values are within this initial range, we hone it on them by narrowing the search a new interval closer to these optimal values. If however we get that the best hyperparameters lie on the end points of this interval, it is indicative that our starting point was too narrow and thus we would need to conduct the test again but with a broader starting range. This approach is illustrated below:

```
set.seed(12345)
# Define the log ranges for cost and gamma
log.C.range <- seq(-5,5, by = 1)
log.gamma.range <- seq(-5, 5, by = 1)


coarse_accuracy_matrix <- rbf_grid_search(train.X, train.labels, log.C.range, log.gamma.range)

# Display the accuracy matrix
print(coarse_accuracy_matrix )
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
##  [1,] 10.6 10.8 12.2 10.2 10.7  8.9 10.7 12.2 10.8  10.2  12.2
##  [2,] 10.8 10.6 10.5 12.2 10.0 12.2  9.4 10.9  8.6  12.2  12.2
##  [3,] 19.7 22.0 12.3 12.2 12.2 12.2 11.0 10.7 10.6  12.2  10.5
##  [4,] 65.9 68.0 21.3 11.4 12.2 10.4 10.2 12.2 12.2  10.1  12.2
##  [5,] 83.0 87.7 61.5 16.8 12.2 12.2 10.4 17.8 12.2  12.2  10.1
##  [6,] 88.6 89.9 88.2 27.7 13.5 10.4 10.8 11.0 10.1  12.2  10.7
##  [7,] 89.5 91.4 88.2 29.6 17.1 10.0 12.2 10.2 10.9  12.2  12.2
##  [8,] 88.5 92.0 88.4 32.6 14.3 12.2 12.2 15.3 12.2  10.8  10.7
##  [9,] 89.6 90.6 88.7 30.3 14.1 10.6 12.2 10.5 12.2  10.6  10.1
## [10,] 89.9 91.2 87.9 32.4 15.0 10.9 10.9 12.2 10.4  10.1  10.9
```

```
## [11,] 89.0 90.7 88.3 32.4 16.3 10.2 12.2 10.6 12.2  10.1   9.8
```

```
# Extract the values for the parameters
best_coarse_index <- which(coarse_accuracy_matrix == max(coarse_accuracy_matrix), arr.ind = TRUE)
best_coarse_index
```

```
##      row col
## [1,]   8   2
```

```
best_coarse_log_C <- log.C.range[best_coarse_index[1]]
best_coarse_log_C
```

```
## [1] 2
```

```
best_coarse_log_gamma <- log.gamma.range[best_coarse_index[2]]
best_coarse_log_gamma
```

```
## [1] -4
```

From the first (coarse) run of the function on the dataset, we see that the best values of `log.C` is 2 and the best `log.gamma` value is -4, yielding an accuracy of 92%. Since these are within the range that we defined for these values, we fine tune the result by redefining the lists `log.C.range` and `log.gamma.range` to be closer to these values as below:

```
set.seed(12345)
fine_log.C.range <- seq(1.5, 2.5, by = 0.1)
fine_log.gamma.range <- seq(-4.5, -3.5, by = 0.1)

fine_accuracy_matrix <- rbf_grid_search(train.X, train.labels, fine_log.C.range, fine_log.gamma.range)

# Display the fine accuracy matrix
print(fine_accuracy_matrix)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
##  [1,] 89.9 91.8 90.4 90.6 90.5 92.0 89.9 90.8 91.3  89.8  90.6
##  [2,] 90.1 90.4 90.4 89.2 91.3 91.8 91.5 89.5 90.7  91.4  90.5
##  [3,] 91.1 90.3 90.3 90.6 91.1 90.2 91.0 91.5 91.3  91.5  90.8
##  [4,] 90.9 91.2 90.8 90.6 90.8 90.3 89.1 91.1 90.9  90.7  90.6
##  [5,] 92.5 91.3 90.8 91.8 91.8 90.9 89.7 90.6 90.5  91.3  90.4
##  [6,] 90.5 90.7 90.4 90.4 90.3 90.5 91.2 90.6 91.3  90.3  90.0
##  [7,] 89.8 90.8 90.2 91.2 90.5 91.0 90.5 90.2 90.7  91.8  90.3
##  [8,] 89.8 91.6 90.0 90.4 91.4 91.6 90.5 91.1 90.2  91.2  90.4
##  [9,] 90.3 90.6 90.9 91.4 90.3 90.7 91.7 91.1 90.8  90.2  91.3
## [10,] 90.3 90.6 90.1 90.8 91.6 90.7 90.5 91.9 90.3  91.0  90.5
## [11,] 90.5 90.6 90.5 90.1 91.3 90.4 90.4 90.7 90.7  90.4  90.9
```

```
# Extract the values for the fine tuned parameters
best_fine_index <- which(fine_accuracy_matrix == max(fine_accuracy_matrix), arr.ind = TRUE)
best_fine_index
```

```
##      row col
## [1,]   5   1
```

```
best_fine_log_C <- fine_log.C.range[best_fine_index [1]]
best_fine_log_C
```

```
## [1] 1.9
```

```
best_fine_log_gamma <- fine_log.gamma.range[best_fine_index [2]]
best_fine_log_gamma
```

```
## [1] -4.5
```

We have now found the optimal values for `log.C` and `log.gamma` through fine tuning the parameters. Now we are ready to compute the accuracy by running it on the provided testing data.

```
# Compute the actual cost and gamma values
best_fine_cost <- exp(best_fine_log_C)
best_fine_gamma <- exp(best_fine_log_gamma)

final_model <- svm(train.X, train.labels,
                   type = "C-classification",
                   kernel = "radial",
                   cost = best_fine_cost,
                   gamma = best_fine_gamma)

predictions <- predict(final_model, test.X)
test_accuracy <- mean(predictions == test.labels) * 100
test_accuracy
```

```
## [1] 91.4
```

We obtain a strong overall accuracy of 91.4% on the MNIST test dataset. However, it must be noted that the careful tuning of the hyperparameters played a crucial role in achieving this performance. While there is always scope for improvement, e.g. by fine tuning more than once, this is a strong starting point for this dataset.