ST340 Programming for Data Science Assignment 3

Stephen Brownsey: U1619685

Q1 Gradient descent

```
library(tidyverse)
set.seed(666)
```

The function, gradient.descent, runs gradient descent with a fixed number of iterations to find the local minima:

```
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
}
```

Example as given in the assignment:

```
f <-function(x) { sum(x^2) }
gradf<-function(x) { 2*x }
gradient.descent(f,gradf,c(10,20),10,0.2)
```

(a)

It is given that gradient descent can be assumed to be a *black box* algorithm, which means it can be used in the gradient ascent algorithm. For a given function, f, gradient descent moves in the direction which decreases most rapidly. To change this algorithm to gradient ascent: it is required to define a new function, gradient.ascent, which moves in the direction which f increases most rapidly. This can be achieved by reversing the sign of the *eta* parameter in the gradient.descent function:

```
gradient.ascent <- function(f, df, x0, iterations=1000, eta=0.2) {
  #Switching the sign used in gradient descent
  gradient.descent(f, df, x0, iterations, -eta)
}
```

Testing gradient ascent works using the supplied 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
```

The gradient.ascent function works as expected and the maximum of the function $\frac{1}{1+x^2}$ is achieved when $x = 0$ and $f(x) = 1$. As seen by the output of the code.

(b) Consider the function $f : \mathbb{R}^2 \to \mathbb{R}$ given by:

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

i) Give a short mathematical proof that f has a unique minimum.

The given R code shows $f : \mathbb{R}^2 \to \mathbb{R}$, $f(x_1, x_2) = (x_1 - 1)^2 + 100(x_1^2 - x_2)^2$. It can be noted that f is a sum of squares, which means that that f can be denoted as $f(x_1, x_2) \geq 0$. In order to find the minimum, it is required to find the point such that both terms are equal to zero, this can be formulated below in equations [1] and [2]:

$$(x_1 - 1) = 0 \quad [1]$$
$$(x_1^2 - x_2) = 0 \quad [2]$$

From the equations, with a bit of algebraic manipulation, it can be concluded that $x_1 = 1$ and $x_2 = x_1^2 = 1^2 = 1$. Since both equations only have one unique result, the values x_1 and x_2 must be unique and as such $f(x_1, x_2)$

obtains its unique minimum at $f(1,1) = 0$

ii)

As stated in the question, the function *gradf* is required to take in in one parameter containing two elements referring to each of the x_1 and x_2 used in the function f in the previous part. To calculate gradf, it is required to take partial derivatives of the function f. Using the fact $x_2 = x_1^2$, this can be denoted by:

$$\begin{aligned}\nabla f(x_1, x_2) &= (\frac{\partial f}{\partial x_1}(x_1, x_2), \frac{\partial f}{\partial x_2}(x_1, x_2)) \\ &= ((400x_1^3 + 2x_1 - 400x_1x_2 - 2), (200x_2 - 200x_1^2))\end{aligned}$$

```
gradf <- function(x){
  #defining partial differentials as per method above
  diff_x1 <- 400 *x[1]^3 + 2 * x[1] - 400 * x[1] * x[2] - 2
  diff_x2 <- 200 * x[2] - 200 * x[1] ^ 2
  #Returning the outputs of the partial differentials together
  c(diff_x1, diff_x2)
}
```

To check the *gradf* function is working as expected, it can be run on $f(1,1)$, since this is a minimum, it should return $(0,0)$

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

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

iii)

The gradient descent given prints out a lot of *surplus* information, so the function has been altered below to print out the output in a more usable format. When running thousands of iterations, it is not desirable to have text printed to console as this takes up a lot of time to write out. Instead a tibble is returned containing the number of iterations, eta value used, x and $f(x)$ of just the final iteration.

```
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)
  }
  tibble(iterations = iterations, eta = eta, x = x, f_x = f(x))
}
```

Based on the literature the value of eta makes a difference as to whether convergence will be achieved, so various values of eta will be looped through to find some which could lead to convergence. If the outcome of the iterations is NaN then it is clear that convergence will not be achieved with this value of eta.

```
test_eta <- tibble()
for(i in c(0.00001, 0.0001, 0.001, 0.01 , 0.1 ,0.5 ,1 ,5)){
 test_eta <-  bind_rows(test_eta, gradient.descent(f, gradf, c(3,4), iterations = 50, eta = i))
}
#Showing the output of the function call, using head() to
#Restrict the number of rows displayed
#It can be seen here how for each eta, there are two rows
#The first refers to x1 and the second: x2, both in the x column
```

```
test_eta %>%
  head()
```

```
## # A tibble: 6 x 4
##   iterations     eta      x    f_x
##        <dbl>   <dbl>  <dbl>  <dbl>
## 1         50 0.00001   2.14  19.1
## 2         50 0.00001   4.17  19.1
## 3         50 0.0001    2.04   1.09
## 4         50 0.0001    4.18   1.09
## 5         50 0.001   NaN    NaN
## 6         50 0.001   NaN    NaN
```

```
#Subsetting the function as required, to find range for
#Further exploration
test_eta %>%
  filter(!is.nan(x)) %>%
  select(eta) %>%
  distinct
```

```
## # A tibble: 2 x 1
##       eta
##     <dbl>
## 1 0.00001
## 2 0.0001
```

Now that the range from 0.00001 to 0.001 has been identified as giving good possibilities of convergence results. Intervals of 0.00001 between these two values will be considered. It is known that the value of $f(x)$ must be positive. Therefore, the closest value to zero, (lowest), will lead to the best estimate of convergence from the range explored. It is then necessary to consider odd terms for the convergence value of x_1 and even terms for the convergence value of x_2. This is due to the tibble dataframe having one row for x_1 and the next row for x_2.

```
convergence <- tibble()
for(i in seq(from =  0.00001, to = 0.001, by = 0.00001)){
 convergence <-  bind_rows(convergence, gradient.descent(f,gradf,c(3,4),
                                                        iterations = 50000, eta = i))
}
#Removing non-convergant occasions should there be any
convergence <- convergence %>%
  filter(!is.nan(x))
#x1 coordinate is odd row number, so finding x1 convergence
x1 <- convergence %>%
  filter(row_number() %% 2 == 1) %>%
  slice(which.min(f_x))
#Showing whole row output
x1
```

```
## # A tibble: 1 x 4
##   iterations     eta     x      f_x
##        <dbl>   <dbl> <dbl>    <dbl>
## 1      50000 0.00098 1.000 3.17e-18
```

```
#extracting just x for use in text below
x1 <- x1 %>% select(x)
#x2 coordinate is the even row number, so finding x2 convergence
```

```r
x2 <- convergence %>%
  filter(row_number() %% 2 == 0) %>%
  slice(which.min(f_x))
#Showing whole row output
x2
```

```
## # A tibble: 1 x 4
##   iterations     eta     x      f_x
##        <dbl>   <dbl> <dbl>    <dbl>
## 1      50000 0.00098 1.000 3.17e-18
```

```r
#extracting just x for use in text below
x2 <- x2 %>% select(x)

#Calculating the eta value at which lowest convergence value occured.
optimum_eta <- convergence %>%
  filter(f_x == min(f_x)) %>%
  select(eta) %>%
  as_vector() %>%
  #So that the value doesn't appear twice
  unique()
optimum_eta
```

```
## [1] 0.00098
```

```r
#As a point
x <- c(x1, x2)
x
```

```
## $x
## [1] 1
##
## $x
## [1] 1
```

This demonstrates that as $x_1 = 0.999999998221768$ and $x_2 = 0.99999999643642$, both variables converge to 1. The optimal eta value of 9.8×10^{-4} has also been calculated and this will be used in the next part of the question where momentum descent is considered.

(c)

The implementation of momentum descent can be seen below, this is based on material from the lectures and takes takes an eta and alpha parameter. The output of this function is a tibble dataframe containing the number of iterations, eta and alpha used along with the x and $f(x)$ value associated with final iteration. Again the first row will refer to x_1 and the second row x_2.

```r
momentum_descent <- function(f, gradf, x0, iterations = 1000, eta = 0.2, alpha = 0.5) {
#Setting both x1 and x2 equal to starting point x0
x1 <- x0
x2 <- x0
#iterating through with alpha term and updating
#x1 and x2 at each stage
for(i in 1:iterations){
  x2 <- x1 - eta * gradf(x1) + alpha * (x1 - x0)
  x0 <- x1
  x1 <- x2
  }
  tibble(iterations = iterations, eta = eta, alpha = alpha, x = x1, f_x = f(x1))
}
```

Setting the eta value to be our optimum_eta: 9.8×10^{-4} calculated in part *biii*), it is then necessary to loop through the alpha values to come up with a suitable range for testing in more detail in order to find our optimum eta, α combination.

```
test_momentum <- tibble()
for(i in c(0.00001, 0.0001, 0.001, 0.01 , 0.1 ,0.5 ,1 ,5)){
test_momentum <-  bind_rows(test_momentum,
                            momentum_descent(f, gradf,c(3,4), iterations = 50,
                                             eta = optimum_eta, alpha = i))
}

test_momentum %>%
  filter(!is.nan(x)) %>%
  select(alpha, x) %>%
  distinct
```

```
## # A tibble: 8 x 2
##     alpha     x
##     <dbl> <dbl>
## 1 0.00001 -1.52
## 2 0.00001  2.21
## 3 0.0001  -1.52
## 4 0.0001   2.21
## 5 0.001   -1.55
## 6 0.001    2.22
## 7 0.01     1.55
## 8 0.01     2.25
```

Based on this output, it can be noted that looking at α values between 0.01 and 1 in more detail is likely to yield an optimum result. The code below will loop between 0.01 and 1 at increments of 0.0001. Again, the best estimate will occur for the lowest value of $f(x)$, and its associated values of x_1 and x_2.

```
#Looping through to find optimum value of alpha
momentum <- tibble()
for(i in seq(from = 0.01 , to = 1 , by = 0.0001)){
momentum <-  bind_rows(momentum,
                        momentum_descent(f, gradf, c(3,4), iterations = 50,
                                         eta = optimum_eta,
                                         alpha = i))
}

#Demonstrating what the momentum tibble looks like
momentum %>%
  head()
```

```
## # A tibble: 6 x 5
##   iterations     eta  alpha     x   f_x
##        <dbl>   <dbl>  <dbl> <dbl> <dbl>
## 1         50 0.00098 0.01    1.55  2.84
## 2         50 0.00098 0.01    2.25  2.84
## 3         50 0.00098 0.0101  1.57  4.56
## 4         50 0.00098 0.0101  2.25  4.56
## 5         50 0.00098 0.0102  1.56  3.58
## 6         50 0.00098 0.0102  2.25  3.58
```

```r
#x1 value
x1 <- momentum %>%
  filter(row_number() %% 2 == 1) %>%
  slice(which.min(f_x)) %>%
  select(x)

#x2 value
x2 <- momentum %>%
  filter(row_number() %% 2 == 0) %>%
  slice(which.min(f_x)) %>%
  select(x)

#point
c(x1 ,x2)
```

```
## $x
## [1] 1.001931
##
## $x
## [1] 1.003873
```

This demonstrates that since $x_1 = 1.00193070406454$ and $x_2 = 1.00387285260996$, both converge to 1 as in part *biii*). This time however, only 50 iterations are used rather than 50,000. Overall, this demonstrates how gradient decent with momentum is much quicker to converge than gradient decent without momentum ($\alpha = 0$). If the number of iterations were increased, this value would get closer and closer to 1, the reason 50 was chosen was just to emphasise how quickly it does converge.

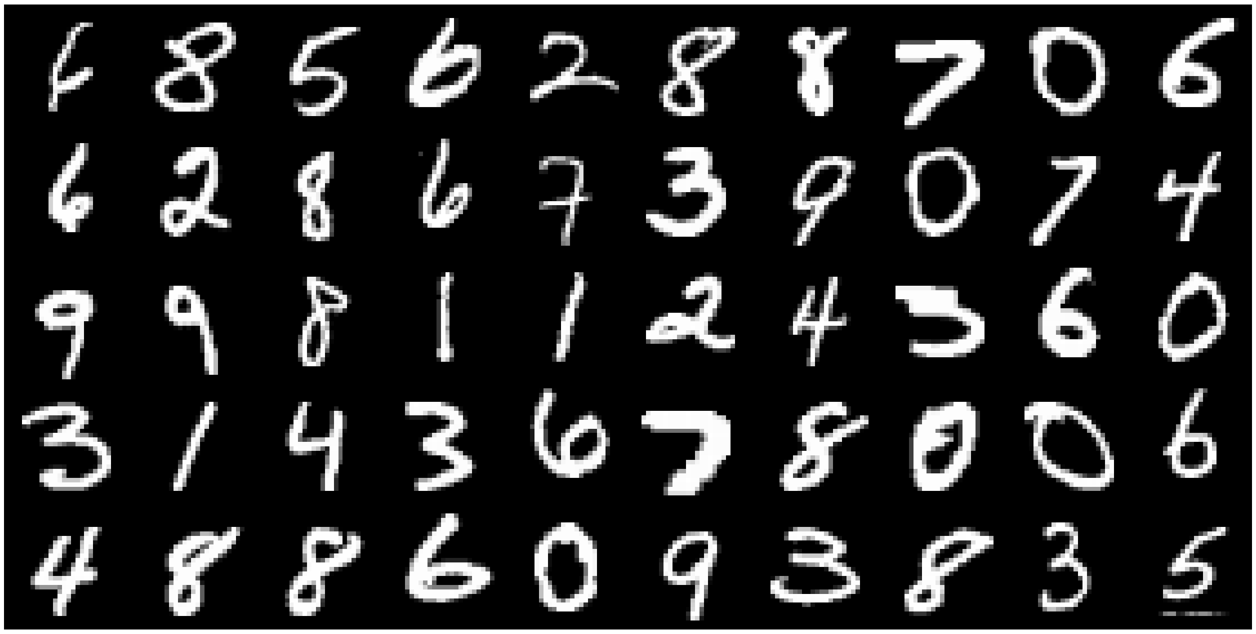
Q2 Support vector machines

Below is some code which was given in the question to set it up. Run the following code to load the tiny MNIST dataset:

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

and then show some digits:

```r
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)
```



Example code given in question sheet:

```r
library(e1071)
```

(a)

The aim is to use three-fold cross validation on the training set to compare SVMs with the following kernals; linear [1], polynomial [2] and RBF [3] denoted below:

$$Linear : K(x, x') = x \cdot x' \qquad [1]$$
$$Polynomial : K(x, x') = (c + \gamma x \cdot x')^p, \quad c \geq 0, \;\; p \in [2, 3, ...) \qquad [2]$$
$$RBF : K(x, x') = exp(-\gamma ||x - x'||^2), \quad \gamma \geq 0 \qquad [3]$$

In each scenario, different inputs will be looped through and the optimal one for each kernal used for the purposes of comparison.

Linear Kernels

To assess the accuracy of the linear model, the only tuning parameter to be considered is the cost:

```r
tuning <- c(0.001, 0.01, 0.1, 1, 10, 50, 100, 250)
n <- length(tuning)
linear_accuracy = vector(mode ="numeric", length = length(tuning))
for(i in 1:n){
  linear_accuracy[i] <- svm(train.X, train.labels, type = "C-classification",
             kernel="linear", cross = 3, cost = tuning[i])$tot.accuracy
}
linear_accuracy
```

```
## [1] 63.5 86.5 86.2 86.1 85.2 86.8 86.5 86.4
```

```
#Calculating the maximum linear svm accuracy
max_lin <- max(linear_accuracy)
max_lin
```

```
## [1] 86.8
```

```
positions <- which(linear_accuracy == max(linear_accuracy), arr.ind = TRUE)
best_cost <- tuning[positions[1]]
best_cost
```

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

The linear kernel svm tends to have a good general performance for all cost parameters ≥ 0.01. With an optimum accuracy of 86.8 occuring at cost = 50.

Polynomial Kernels

As well as the cost tuning parameter, polynomial kernals also have a γ tuning parameter, so this will be defined below as well and then used to create the models. These models will be stored in a matrix and the the optimum accuracy score taken for comparison.

```
#length(gamma_values) also equals n so no need to redifine a new n variable
gamma_values <- c(0.001, 0.01, 0.1, 1, 10, 50, 100, 250)

#Defining the matrix with abitrary values which will all be overwritten
#Chose the values to be > 100 so if there is an error in the code
#Then it will be apparent as max will return a number > 100
polynomial_accuracy <- matrix(666, nrow = n, ncol = n)

for(i in 1:n){
  for(j in 1:n){
      polynomial_accuracy[i,j] <- svm(train.X, train.labels,
                                         type = "C-classification", kernel = "polynomial",
degree = 2, coef = 1, cross = 3, gamma = gamma_values[i], cost = tuning[j])$tot.accuracy
  }
}
polynomial_accuracy
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]  7.7 10.5 10.6 77.6 87.0 87.5 86.6 87.8
## [2,] 12.2 13.4 80.7 86.8 87.8 88.0 88.1 88.1
## [3,] 62.0 86.8 89.9 88.8 88.7 89.8 88.5 87.1
## [4,] 88.1 88.5 88.1 87.7 88.9 89.4 88.5 87.6
## [5,] 88.9 88.5 88.3 87.6 88.7 89.1 86.8 87.5
## [6,] 88.4 87.9 89.4 89.6 88.4 88.5 88.2 88.6
## [7,] 88.2 88.1 87.9 88.0 89.0 88.3 89.1 88.6
## [8,] 88.6 89.5 89.1 88.7 89.4 88.3 88.3 89.0
```

```
#Calculating the maximum polynomial svm accuracy
max_poly <- max(polynomial_accuracy)
max_poly
```

```
## [1] 89.9
```

```
#Calculating the optimum value of gamma and cost used:
#Where row is gamma and cost is column
positions <- which(polynomial_accuracy == max(polynomial_accuracy), arr.ind = TRUE)
poly_gamma <- gamma_values[positions[1]]
poly_c <- tuning[positions[2]]

positions
```

```
##      row col
## [1,]   3   3
```

```
poly_gamma
```

```
## [1] 0.1
```

```
poly_c
```

```
## [1] 0.1
```

The polynomial kernel svm gives good accuracy values in general for $\gamma \geq 1$ for all c values in the range considered. It has an optimum accuracy of 89.9 occurring at $\gamma = 0.1$ and cost $= 0.1$.

RBF

Similar to polynomial kernal, a matrix is defined which contains all the accuracy values for the different combinations of cost and α tuning parameters.

```
radial_accuracy <- matrix(666, n, n)

for(i in (1:n)){
  for(j in (1:n)){
  radial_accuracy[i, j] = svm(train.X, train.labels, type = "C-classification", kernel = "radial",
  cross = 3, gamma = gamma_values[i], cost = tuning[j])$tot.accuracy

  }
}
radial_accuracy
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,] 18.3 10.0 10.9 77.9 88.4 87.0 85.7 86.4
## [2,]  9.8 10.5 56.9 89.0 91.3 89.8 90.3 88.8
## [3,] 12.2 10.7 10.7 51.9 55.3 58.2 54.5 54.7
## [4,]  9.1 10.6 12.2 10.0 10.4 10.5 10.2 10.1
## [5,] 12.2 17.8  9.9 15.0  8.8 12.2 15.6 10.3
## [6,] 12.2 12.2 10.6 10.6 10.3 10.7 12.2 10.6
## [7,] 10.6 10.7 10.4 10.7 10.4 10.4 12.2 12.2
## [8,] 11.0 12.2 12.2 10.8 10.7  9.9  9.6 10.4
```

```
#Calculating the radial polynomial svm accuracy
max_radial <- max(radial_accuracy)
max_radial
```

```
## [1] 91.3
```

```
positions <- which(radial_accuracy == max(radial_accuracy), arr.ind = TRUE)
radial_gamma <- gamma_values[positions[1]]
radial_c <- tuning[positions[2]]
```

```
positions
```

```
##      row col
## [1,]   2   5
```

```
radial_gamma
```

```
## [1] 0.01
```

```
radial_c
```

```
## [1] 10
```

The radial kernel gives very good accuracy but over a much smaller range of c and γ values, showing just how sensitive it is to any changes in these values. It has an optimum accuracy of 91.3 occurring at $\gamma = 0.01$ and cost = 10.

Conclusion

The linear kernel gives good accuracy for all cost parameters considered ≥ 0.01, this can be improved upon by using the polynomial kernel. The polynomial kernel is more robust to changes in cost and γ in comparison to the radial kernal. The accuracy of the kernels can be summarised as: radial > polynomial > linear as shown by 91.3 > 89.9 > 86.8. From this it can be clearly seen that the radial kernal has the optimum maximum accuracy and as such is the best kernel. This does however need the cost and γ value combination to be chosen very specifically as these play a large role in the performance of the radial svm.

Warning

The Warning given is: ('X' and … 'Xn') . Cannot Scale the data. This warning message occurs as the SV matrix output from the svm function contains most values of 0. As such, this matrix is very sparse and the warning message is just ensuring the user's awareness of this.

(b)

Based on the advice in the lecture, the required function, radial_gs, needs to take in two list parameters as well as the two training X and Y variables. Thee-fold cross validation will then be used to find the optimum cross validation accuracy and associated log.C.Range and log.gamma.range values. From the analysis undertaken in part *a*), radial svm was the optimum model and as such the following code will be based on that approach. The output from the function will be a list containing four objects: the first is a matrix containing the accuracies all the svm models, where the row denotes the log.gamma value and column the , the second contains the optimum accuracy from the cross validation, thirdly the location of the optimum log.C value and lastly the location of the optimum log.gamma value. These can be extracted from the list by referencing their respective positions. To calculate the optimum log.c value and optimum log.gamma value the location is passed in as the index to the list. The reason for writing the code like this, rather than visually selecting the highest combination, is to enable theoretical future improvements where thousands of model combinations could be considered.

```r
radial_gs  <- function(log.C.range, log.gamma.range, train.X , train.Y){
#Defining best values so far
best_cv <- 0
best_c <- 0
best_gamma <- 0

#Defining length variables and matrix
m <- length(log.gamma.range)
n <- length(log.C.range)
gs_accuracy <- matrix(666, m, n)

for(i in 1:m){
  for(j in 1:n){
  gs_accuracy[i, j]  <- svm(train.X,train.Y , type = "C-classification", kernel = "radial",
  gamma = exp(log.gamma.range[i]), cost = exp(log.C.range[j]), cross = 3)$tot.accuracy

  #Updating best so far if new i,j model has better accuracy
  if(gs_accuracy[i, j] > best_cv){
    best_cv <- gs_accuracy[i, j]
    #rows relate to gammma values, so storing row number of optimum
    best_gamma <- i
    #columns relate to c values, so storing colummn number of optimum
    best_c <- j
    }

  }
}

list(gs_accuracy, best_cv, best_c, best_gamma)

}
```

The first round of exploring will consider values of log.C.range and log.gamma.range on the interval scale between $[-5, 5]$. The accuracy values will be stored in a matrix and the region with the best results will be investigated further. So long as the optimum values calculated at the end are not the extreme values of the range, then it will be a suitable optimal solution. In other words, if the optimal solution was for example (5,5), more searching would be required as it is likely there is a better solution outside of the range first considered.

```
first_round <-  radial_gs(c(-5:5), c(-5:5), train.X, train.labels)
#Extacting the matrix from the first round variable
first_round[[1]]
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
##  [1,] 12.2 10.4 20.6 64.8 83.2 88.2 89.7 89.8 89.3  90.8  90.1
##  [2,] 10.8 12.9 20.9 67.5 86.2 89.4 90.4 91.0 90.4  90.6  91.3
##  [3,] 12.2 13.1 10.7 21.8 61.9 87.1 88.0 88.5 85.9  88.7  87.5
##  [4,] 10.6  9.8 10.6 10.1 14.9 27.4 32.2 33.8 34.0  31.9  30.0
##  [5,] 10.2  9.8 10.7 12.2 10.7 14.5 14.8 15.0 15.8  16.4  15.6
##  [6,] 10.2 10.5 10.7 10.4 12.2 10.5 11.2 12.2 10.5  12.2  11.4
##  [7,] 12.2 18.1 10.2 10.6 12.2 12.2  8.6 10.7 12.2  10.7  12.2
##  [8,] 15.3  7.8 13.3 12.2 10.1 12.2 12.2 10.6  9.8  10.5  12.2
##  [9,] 12.2  8.3 12.7 10.1  9.8 12.2 12.2 12.2 12.2  12.2   8.6
## [10,] 10.7 10.5 10.7 12.2 12.2 10.5 12.2 12.2 10.7  10.9  10.4
## [11,]  9.7 12.2 10.6 10.4 12.2 10.8 12.2  9.1 12.2  12.2  10.9
```

From a visual inspection of the output, the second round of grouping will be analysis will be conducted on log.C.range $\in [1, 5]$ and log.gamma.range $\in [-5, -3]$ with interval steps of 0.5 and 0.25 respectively.

```
c_seq <- seq(1, 5, 0.5)
gamma_seq <- seq(-5, -3, 0.25)
second_round <-  radial_gs(c_seq, gamma_seq , train.X, train.labels)
second_round[[1]]
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
##  [1,] 90.0 89.6 89.2 89.6 89.0 89.3 90.2 90.0 89.5
##  [2,] 90.8 90.2 89.8 90.2 88.5 89.8 90.5 90.4 89.7
##  [3,] 90.7 91.1 89.9 89.6 90.6 89.7 90.0 89.7 89.5
##  [4,] 90.9 91.5 91.2 91.3 90.8 90.7 90.3 91.0 90.9
##  [5,] 91.3 90.5 90.1 90.8 91.1 90.1 90.9 91.8 90.5
##  [6,] 90.4 91.5 90.8 91.2 90.2 90.9 91.3 89.8 90.6
##  [7,] 90.4 89.9 90.4 89.9 90.1 90.8 91.1 90.4 91.0
##  [8,] 90.4 89.6 89.9 90.2 89.7 90.4 90.2 89.3 90.4
##  [9,] 89.0 88.5 88.6 88.5 89.4 88.1 88.5 88.2 87.9
```

```
#Storing the optimum calculation from our second scenario for use in the next part
optimum <- tibble(best_cv = second_round[[2]],
#This line calculates the optimum values by passing the location value into the list
                  best_c = c_seq[second_round[[3]]], best_gamma = gamma_seq[second_round[[4]]])
```

Finally, the final svm can be defined. It uses a radial kernel with the optimal values of c and γ associated with the optimum cross validation of 91.8 calculated from our second round of inspection. These are $c =$ 4.5 and $\gamma =$ -4. This model is trained on the whole training data. Then, to test the accuracy, it is run on the test data and the mean accuracy of the predictions is calculated as the final accuracy.

```
final_svm <- svm(train.X,train.labels , type = "C-classification", kernel = "radial",
gamma = exp(optimum$best_gamma), cost = exp(optimum$best_c), cross = 3)

final_accuracy <- mean(predict(final_svm, test.X) == test.labels)
final_accuracy
```

```
## [1] 0.916
```

From this we get a final test accuracy of over 90% as shown by: 0.916 which is pretty good. Since the values of c and γ are not the extremes of the range considered, the final accuracy is a valid choice. The accuracy could always be improved by trialing many more combinations and choosing the best of these, but this would require a much greater level of computational resources and parallelisation of code to run effectively.