Chapter 8: Performance Optimization

Brandon Morgan

1/16/2021

E9.1

We are given the following vector function in quadratic form:

$$F(X) = \frac{1}{2}X^T \begin{bmatrix} 6 & -2 \\ -2 & 6 \end{bmatrix} X + [-1 \ -1]X$$

Which can be reduced down to:

$$F(X) = 3x_1^2 - x_1x_2 + 3x_2^2 - x_1 - x_2$$

1

```r
library(purrr) # used for map2_dbl function
library(plotly) # used for 3D plotting
```

```
## Warning: package 'plotly' was built under R version 3.6.3

## Loading required package: ggplot2

## Warning: package 'ggplot2' was built under R version 3.6.3

##
## Attaching package: 'plotly'

## The following object is masked from 'package:ggplot2':
##
##     last_plot

## The following object is masked from 'package:stats':
##
##     filter

## The following object is masked from 'package:graphics':
##
##     layout
```

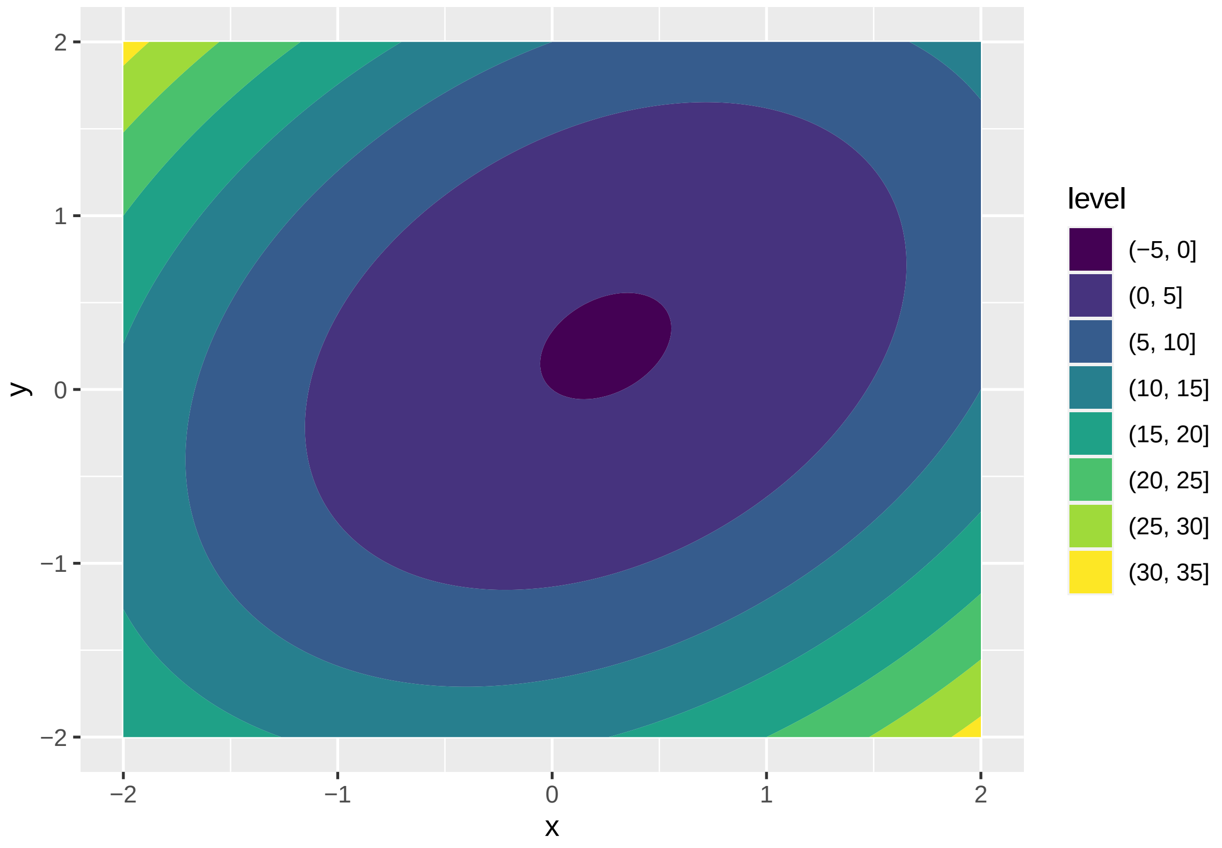
```r
library(ggplot2) # used for plotting
```

```r
fun = function(x, y) { # our original function
  A = matrix(c(6, -2, -2, 6), ncol=2)
  d = matrix(c(-1, -1), ncol=2)
  X = matrix(c(x, y), ncol=1)
  0.5*t(X)%*%A%*%X+d%*%X
}
```

Here we have the contour of our function:

```r
# BOUNDS USED FOR CONTOUR
pointsX = seq(-2, 2, length=200)# create a 200x1 vector of values for x axis
pointsY = seq(-2, 2, length=200) # create a 200x1 vector of values for y axis
myGrid = expand.grid(pointsX, pointsY) # create 200x200 grid of points
z = map2_dbl(myGrid$Var1, myGrid$Var2, ~fun(.x, .y)) # maps all possible combinations
# of the grid through the function
x = myGrid$Var1
y =  myGrid$Var2
myPlot = plot_ly(x=~x, y=~y, z=~z, type="mesh3d", intensity=~z,
        colors = colorRamp(c("black","red","yellow","chartreuse3")))
#myPlot
myGrid$z = z

v = ggplot( myGrid,aes(x, y, z=z)) +geom_contour_filled()
v
```



In figure 1 we have a three dimensional rendering of our function to add interpretation of our contour plot.

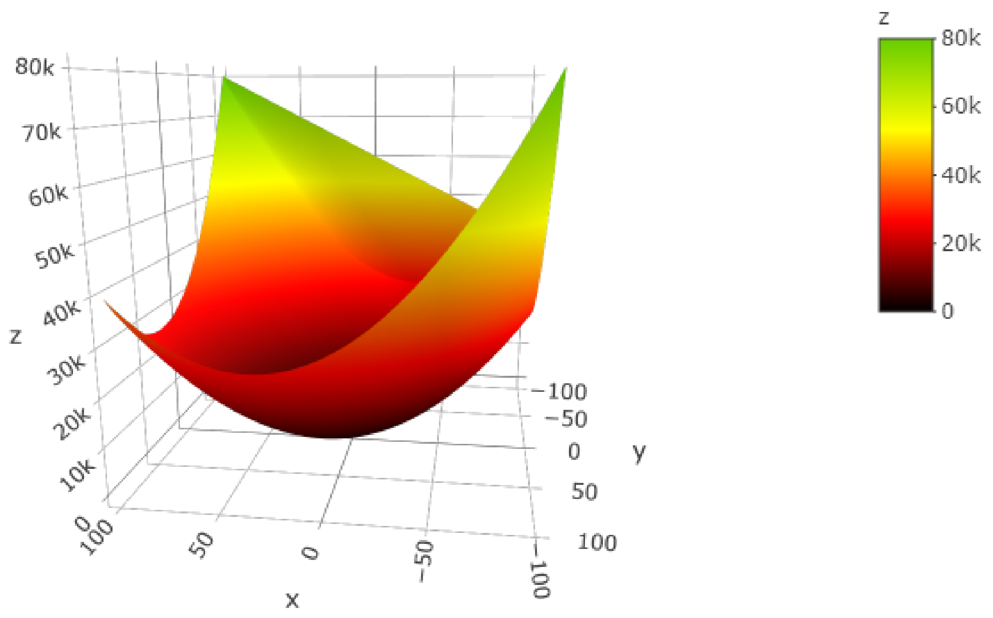


Figure 1: Function 3D graph

2

Unfortunately, I was unable to accurately create orthogonal vectors from the contour lines; so instead, I drew an arrow in the direction of the minimum. We know from Chapter 8 that the stationary points are located at $\nabla F(X) = 0$, where the gradient of a quadtratic function is given to be $\nabla F(X) = AX + d$. Therefore, any set of points that satisfy:

$$\nabla F(X) = \begin{bmatrix} 6 & -2 \\ -2 & 6 \end{bmatrix} X + [-1 \ \ -1] = 0$$

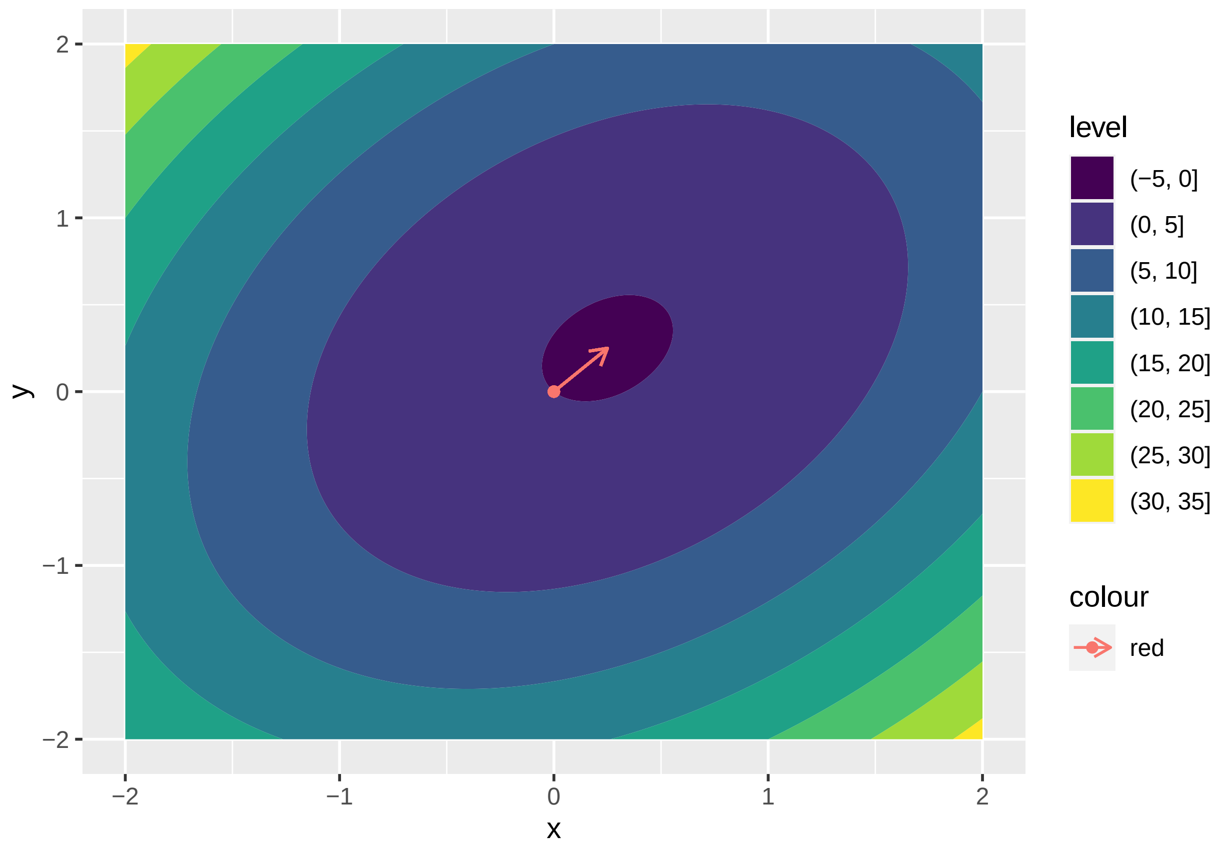
is a stationary point. Our gradient can be reduced down to:

$$\nabla F(X) = \begin{bmatrix} 6x_1 - 2x_2 - 1 \\ -2x_1 + 6x_2 - 1 \end{bmatrix}$$

Like in Chapter 8's solutions, the reduced form of our quadtratic was given earlier, the partial derivaites with respect to each x_i could have been computed to get the same answer. It can be shown algebraically that the gradient only equals zero at $x = [0.25, 0.25]$.

Here we have the same contour but this time with the initial point $x^* = [0, 0]$, pointing to the stationary point $x = [0.25, 0.25]$, which can be shown is the global minimum. Thus, we have a linear trajectory to the minimum.

```
v + geom_point(aes(x=0, y=0, colour="red")) +
  geom_segment(aes(x=0, y=0, xend = 0.25, yend=0.25, colour="red"), arrow=arrow(length=unit(0.25, "cm")
```



3

Given a learning rate of $\alpha = 0.1$, perform two iterations of steepest descent.

Stated in part 2, the gradient of our function can be calculated and reduced down to be:

$$\nabla F(X) = \begin{bmatrix} 6x_1 - 2x_2 - 1 \\ -2x_1 + 6x_2 - 1 \end{bmatrix}$$

First Iteration

To start, we need to find the gradient at our initial point x_0, $g_0 = \nabla F([0, 0])$:

$$g_0 = \nabla F([0, 0]) = \begin{bmatrix} -1 \\ -1 \end{bmatrix}$$

Then, we calculate our next point x_1 by $x_1 = x_0 - \alpha g_0$:

$$x_1 = x_0 - \alpha g_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - 0.1 \begin{bmatrix} -1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix}$$

Second Iteration

To start, we need to find the gradient at our new point x_1:

$$g_1 = \nabla F([0.1, 0.1]) = \begin{bmatrix} -0.6 \\ -0.6 \end{bmatrix}$$

Then, we calculate our next point x_1 by $x_1 = x_0 - \alpha g_0$:

$$x_2 = x_1 - \alpha g_1 = \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix} - 0.1 \begin{bmatrix} -0.6 \\ -0.6 \end{bmatrix} = \begin{bmatrix} 0.16 \\ 0.16 \end{bmatrix}$$

4

The maximum stable learning rate for a quadtratic function is determined by Eq. (9.25):

$$\alpha < \frac{2}{\lambda_{max}}$$

where λ is the maximum eigenvalue of the Hessian matrix.

From chapter 8, the Hessian matrix of a quadtratic matrix is given by $\nabla^2 F(X) = A$. Thus, our Hessian matrix is:

$$\nabla^2 F(X) = A = \begin{bmatrix} 6 & -2 \\ -2 & 6 \end{bmatrix}$$

Which as the following eigenvalues:

```r
A = matrix(c(6, -2, -2, 6), ncol=2)
eigen(A)
```

```
## eigen() decomposition
## $values
## [1] 8 4
##
## $vectors
##            [,1]       [,2]
## [1,] -0.7071068 -0.7071068
## [2,]  0.7071068 -0.7071068
```

The max is calculated to be 8; therefore, the maximum stable learning rate less than

$$\alpha < \frac{2}{8} = 0.25$$

5

Currently not available at the time.

6

Instead of MATLAB code, R code will be used instead to create the steepest descent algorithm.

Here we have our steepest descent function. For this problem, the function has a built in A and d matrix, will requiring input for the initial point, learning rate, algorithm maximum iteration, and tolerance. The maximum iteration is given so as to not enter into an infinity loop and to see how long it will take to compute the stationary point. The tolerance value is given as another stopping criterion where if the gradient at the x value is less than the tolerance, then the stationary point has been found to within said tolerance. To compare the tolerance with the matrix value of the gradient, the Frobenius Norm was used to measure its length, which can be calculated by $||A||_F = \sqrt{\Sigma a_i^2}$ for vectors.

```
steepest_descent = function(init_point, learning_rate, maxIter, tol, mask) {

  A = matrix(c(6, -2, -2, 6), ncol=2) # hard coded for our example

  d = matrix(c(-1, -1), ncol=1) # hard coded for our example


  x = init_point

  # algorithm for fixed learning rate
  for(i in 1:maxIter) {

    if(!mask) { # if the user does not want print out statements
      print(sprintf("      Iteration: %d", i))
      print("x value:")
      print(x)
    }


    g = A%*%x+d # find gradient at x
    x = x - learning_rate*g # find next x

    if(!mask) { # if the user does not want print out statements
       print("gradient value:")
      print(g)
    }


    if(norm(g, "F") < tol) { # convert gradient into single value
                           # through frobenius norm
      print("Stationary Point Found at point:")
      print(x)
      print(sprintf("Iterations taken: %d", i))
      return(x)
    }
  }

  print("MAXIMUM ITERATIONS REACHED")
  print("Current point is: ")
  print(x)
  print("Error: ")
  print(abs(norm(g, "F")-tol)) # computing the error
  return(x)
}
```

Testing Part 3

Here we will call our algorithm to test part 3

```
init = matrix(c(0,0), ncol=1) # initial point
min = steepest_descent(init, 0.1, 2, 1e-12, 0)
```

```
## [1] "      Iteration: 1"
## [1] "x value:"
##      [,1]
## [1,]    0
## [2,]    0
## [1] "gradient value:"
##      [,1]
## [1,]   -1
## [2,]   -1
## [1] "      Iteration: 2"
## [1] "x value:"
##      [,1]
## [1,]  0.1
## [2,]  0.1
## [1] "gradient value:"
##      [,1]
## [1,] -0.6
## [2,] -0.6
## [1] "MAXIMUM ITERATIONS REACHED"
## [1] "Current point is: "
##      [,1]
## [1,] 0.16
## [2,] 0.16
## [1] "Error: "
## [1] 0.8485281
```

Here we will call our algorithm to test part 3; but this time, see how long it takes to converge to the stationary point:

```
min = steepest_descent(init, 0.1, 1000, 1e-12, 1)
```

```
## [1] "Stationary Point Found at point:"
##      [,1]
## [1,] 0.25
## [2,] 0.25
## [1] "Iterations taken: 56"
```

Here we will call our algorithm to test part 4's maximum stable learning rate. Remember, in part 4 it was calculated that any stable learning rate lower than 0.25 could be chosen.

Testing Part 4

```
min = steepest_descent(init, 0.2499, 1000, 1e-12, 1)
```

```
## [1] "Stationary Point Found at point:"
##      [,1]
## [1,] 0.25
## [2,] 0.25
## [1] "Iterations taken: 5"
```

Testing differnt learning rates

When $\alpha = 0.01$:

```
min = steepest_descent(init, 0.01, 1000, 1e-12, 1)
```

```
## [1] "Stationary Point Found at point:"
##      [,1]
## [1,] 0.25
## [2,] 0.25
## [1] "Iterations taken: 687"
```

When $\alpha = 0.05$:

```
min = steepest_descent(init, 0.05, 1000, 1e-12, 1)
```

```
## [1] "Stationary Point Found at point:"
##      [,1]
## [1,] 0.25
## [2,] 0.25
## [1] "Iterations taken: 127"
```

When $\alpha = 0.1$:

```
min = steepest_descent(init, 0.1, 1000, 1e-12, 1)
```

```
## [1] "Stationary Point Found at point:"
##      [,1]
## [1,] 0.25
## [2,] 0.25
## [1] "Iterations taken: 56"
```

When $\alpha = 0.15$:

```
min = steepest_descent(init, 0.15, 1000, 1e-12, 1)
```

```
## [1] "Stationary Point Found at point:"
##      [,1]
## [1,] 0.25
## [2,] 0.25
## [1] "Iterations taken: 32"
```

When $\alpha = 0.20$:

```
min = steepest_descent(init, 0.20, 1000, 1e-12, 1)
```

```
## [1] "Stationary Point Found at point:"
##      [,1]
## [1,] 0.25
## [2,] 0.25
## [1] "Iterations taken: 19"
```

When $\alpha = 0.25$:

```
min = steepest_descent(init, 0.25, 1000, 1e-12, 1)
```

```
## [1] "Stationary Point Found at point:"
##      [,1]
## [1,] 0.25
## [2,] 0.25
## [1] "Iterations taken: 2"
```

When $\alpha = 0.30$:

```
min = steepest_descent(init, 0.30, 1000, 1e-12, 1)
```

```
## [1] "Stationary Point Found at point:"
##      [,1]
## [1,] 0.25
## [2,] 0.25
## [1] "Iterations taken: 19"
```

When $\alpha = 0.35$:

```
min = steepest_descent(init, 0.35, 1000, 1e-12, 1)
```

```
## [1] "Stationary Point Found at point:"
##      [,1]
## [1,] 0.25
## [2,] 0.25
## [1] "Iterations taken: 32"
```

When $\alpha = 0.40$:

```
min = steepest_descent(init, 0.40, 1000, 1e-12, 1)
```

```
## [1] "Stationary Point Found at point:"
##      [,1]
## [1,] 0.25
## [2,] 0.25
## [1] "Iterations taken: 56"
```

When $\alpha = 0.45$:

```
min = steepest_descent(init, 0.45, 1000, 1e-12, 1)
```

```
## [1] "Stationary Point Found at point:"
##      [,1]
## [1,] 0.25
## [2,] 0.25
## [1] "Iterations taken: 127"
```

When $\alpha = 0.46$:

```
min = steepest_descent(init, 0.50, 1000, 1e-12, 1)
```

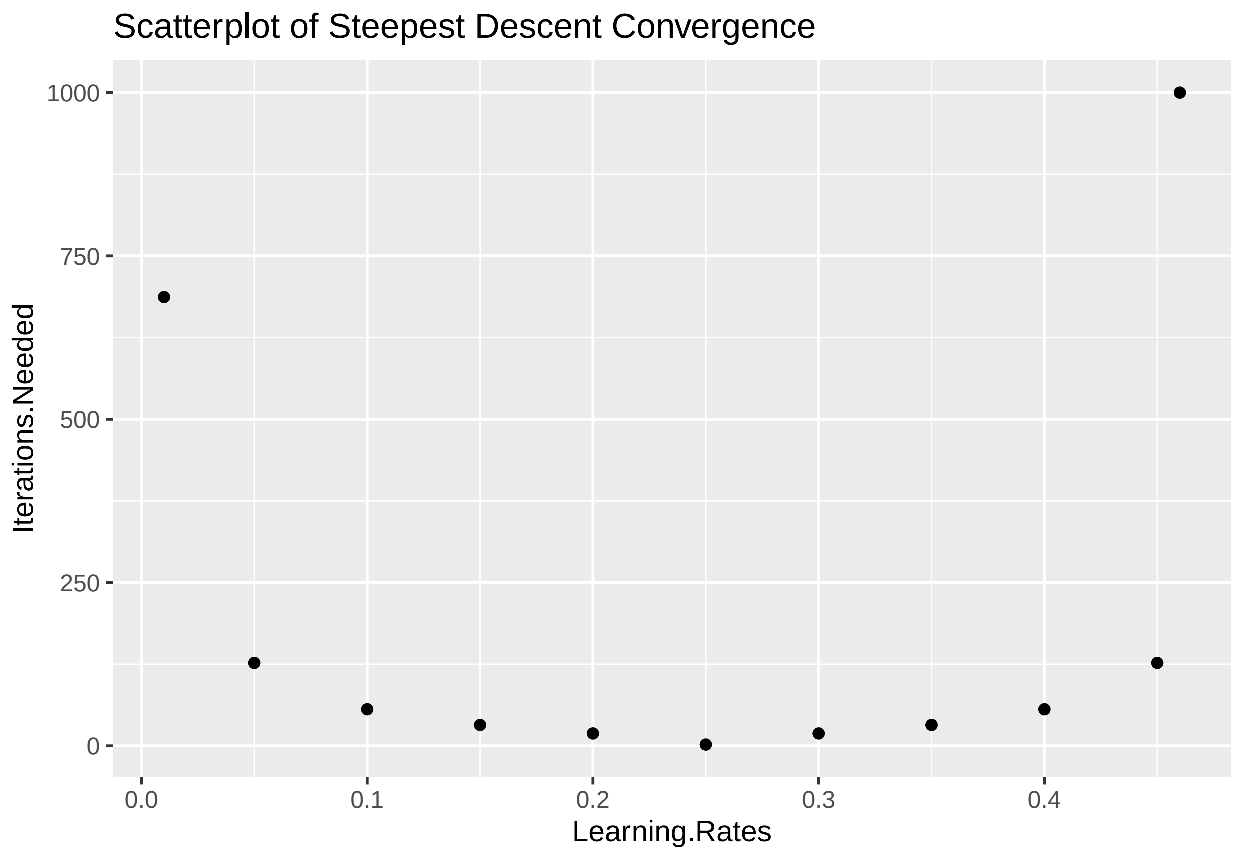
```
## [1] "MAXIMUM ITERATIONS REACHED"
## [1] "Current point is: "
##      [,1]
## [1,]    0
## [2,]    0
## [1] "Error: "
## [1] 1.414214
```

As we can see, we have a safeguard of a stable learning rate of $\alpha < 0.25$; however, it is only until a learning rate greater than 0.45 is used before the algorithm diverges.

In summary, here we have a scatterplot of the learning rates against the number of iterations needed to converg; note that value's of 1000 indicate divergence.

```
learning_rates = c(0.01, 0.05, 0.10, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.46)
iterations_needed = c(687, 127, 56, 32, 19, 2, 19, 32, 56, 127, 1000)
df = data.frame(‘Learning Rates‘ = learning_rates, ‘Iterations Needed‘ = iterations_needed)

ggplot(df, aes(x=‘Learning.Rates‘, y=‘Iterations.Needed‘))+geom_point()+
  ggtitle("Scatterplot of Steepest Descent Convergence")
```


Scatterplot of Steepest Descent Convergence
Iterations.Needed
1000
750
500
250
0
0.0
0.1
0.2
0.3
0.4
Learning.Rates