Assignment 3

Jan Terlikowski: 1609428, Alexander Milanovic: 1610710, Zhaohuang Chen: 1623540

Q1

a)

The function `gradient.descent` is as follows:

```r
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)
    }
  print(x)
}
```

We now create a function `gradient.ascent` by calling only `gradient.descent`:

```r
gradient.ascent <- function(f, df, x0, iterations=1000, alpha=0.2) {
  gradient.descent(f, df, x0, iterations, -alpha)
}

# test 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
```

b) i)

We want to find minimum of:

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

Firstly, we have to find its stationary points by calculating derivative of $f(x_1, x_2)$ and setting it to 0:

$$\frac{\partial f}{\partial x_1} = 2x_1 - 2 + 400x_1^3 - 400x_1x_2$$

$$\frac{\partial f}{\partial x_2} = -200x_1^2 + 200x_2$$

So, the stationary point is the solution to the system of two equations:

$$\begin{cases} 2x_1 - 1 + 400x_1^3 + 400x_1x_2 = 0 \\ -200x_1^2 + 200x_2 = 0 \end{cases}$$

There is only one solution, i.e. there is an unique stationary point:

$$x_1 = x_2 = 1$$

Now, we must check what kind of stationary point (1,1) is. Therefore, we calculate the second derivatives:

$$\frac{\partial^2 f}{\partial x_1^2} = 2 + 1200x_1 + 400x_2$$

$$\frac{\partial^2 f}{\partial x_2^2} = 200$$

$$\frac{\partial^2 f}{\partial x_1 x_2} = -400x_1$$

And we can see that at (1,1):

$$f_{x_1x_1} = 1602 > 0$$

$$\det D^2 f = f_{x_1x_1} \cdot f_{x_2x_2} - f^2_{x_1x_2} = 1602 * 200 - 400^2 = 160400 > 0$$

Hence, $D^2 f(1,1)$ is positive definite which means that (1,1) is an unique minimum point.

b) ii)

We want a function that will take as an input $x = (x_1, x_2)$ and will return:

$$\nabla f = (2x_1 - 2 + 400x_1^3 - 400x_1x_2, -200x_1^2 + 200x_2)$$

This is accomplished using the function `gradf`:

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

# the gradient is:
gradf <- function(x) {
  c(2* x[1] - 2 + 400 * (x[1] ^ 3) - 400 * (x[2]) *
      (x[1]), -200 * (x[1] ^ 2) + 200 * x[2])
}
```

b) iii)

Now we will use `gradient.descent` to find minimum of `f` starting at $x_0 = (3, 4)$. When we set number iterations to 40000 and $alpha = 0.00089$, it becomes apparent that `gradient.descent` converges to 0 as both x_1 and x_2 go to 1, i.e. unique minimum point of the function f can be found at $(1, 1)$ and the minimum value of f is 0.

```
gradient.descent(f, gradf, c(3, 4), 40000, 0.00089)
```

b) iv)

Function performing gradient descent with momentum, called `gradient.descent.momentum` is as follows:

```
gradient.descent.momentum <- function(f, gradf, x0, iterations=1000, eta=0.2, alpha) {
  x1 <- x0
  # make the first iteration
  x2 <- x1 - eta * gradf(x1)
  cat(1, "/", iterations, ": ", x1, " ", f(x1), "\n")
  cat(2, "/", iterations, ": ", x2, " ", f(x2), "\n")
  for (i in 3:iterations) {
    # apply formula for gradient descent with momentum from the lecture notes.
    x <- x2 - eta * gradf(x2) + alpha * (x2 - x1)
    # set new values for x1 and x2
    x1 <- x2
    x2 <- x
    cat(i, "/", iterations, ": ", x, " ", f(x), "\n")
  }
  x
}
```

Now, it will be used to find minimum of the function f:

```
gradient.descent.momentum(f, gradf, c(3,4), 50, 0.000491323, 0.029999)
```

```
## 1 / 50 :  3 4   2504
## NULL
## 2 / 50 :  0.05009671 4.491323   2015.847
## NULL
## 3 / 50 :  0.006730436 4.064971   1653.348
## 4 / 50 :  0.01178231 3.652742   1335.128
## 5 / 50 :  0.02136278 3.281454   1077.452
## 6 / 50 :  0.03638682 2.94791   869.1655
## 7 / 50 :  0.05885567 2.648359   700.4327
## 8 / 50 :  0.09104765 2.379473   563.0773
## 9 / 50 :  0.1353354 2.138403   450.2249
## 10 / 50 :  0.1939023 1.922842   356.0643
## 11 / 50 :  0.2682933 1.731123   275.8104
## 12 / 50 :  0.3587263 1.562336   205.947
## 13 / 50 :  0.463142 1.416396   144.7434
## 14 / 50 :  0.5761996 1.293914   92.70631
## 15 / 50 :  0.6889341 1.195718   52.09353
## 16 / 50 :  0.7902539 1.121915   24.78603
## 17 / 50 :  0.8707518 1.070823   9.789458
## 18 / 50 :  0.9267907 1.038571   3.232056
## 19 / 50 :  0.9612619 1.019952   0.9217182
## 20 / 50 :  0.9804564 1.009967   0.2372861
## 21 / 50 :  0.9904301 1.004885   0.05737245
## 22 / 50 :  0.9953973 1.002381   0.01339641
## 23 / 50 :  0.9978132 1.001169   0.00307178
## 24 / 50 :  0.9989739 1.000589   0.0006979975
## 25 / 50 :  0.999528 1.000312   0.0001579059
## 26 / 50 :  0.9997917 1.00018   3.564718e-05
## 27 / 50 :  0.9999171 1.000118   8.039706e-06
## 28 / 50 :  0.9999766 1.000088   1.812918e-06
## 29 / 50 :  1.000005 1.000074   4.092681e-07
## 30 / 50 :  1.000018 1.000067   9.293943e-08
## 31 / 50 :  1.000025 1.000064   2.165989e-08
## 32 / 50 :  1.000028 1.000062   5.598938e-09
## 33 / 50 :  1.000029 1.000062   1.979906e-09
## 34 / 50 :  1.00003 1.000061   1.164212e-09
## 35 / 50 :  1.00003 1.000061   9.801387e-10
## 36 / 50 :  1.00003 1.000061   9.383751e-10
## 37 / 50 :  1.00003 1.000061   9.286753e-10
## 38 / 50 :  1.00003 1.000061   9.261997e-10
## 39 / 50 :  1.00003 1.000061   9.253519e-10
## 40 / 50 :  1.00003 1.000061   9.24871e-10
## 41 / 50 :  1.00003 1.000061   9.244728e-10
## 42 / 50 :  1.00003 1.000061   9.240935e-10
## 43 / 50 :  1.00003 1.000061   9.237184e-10
## 44 / 50 :  1.00003 1.000061   9.233445e-10
## 45 / 50 :  1.00003 1.000061   9.229709e-10
## 46 / 50 :  1.00003 1.000061   9.225976e-10
## 47 / 50 :  1.00003 1.000061   9.222244e-10
## 48 / 50 :  1.00003 1.000061   9.218513e-10
## 49 / 50 :  1.00003 1.000061   9.214784e-10
## 50 / 50 :  1.00003 1.000061   9.211057e-10
## [1] 1.000030 1.000061
```

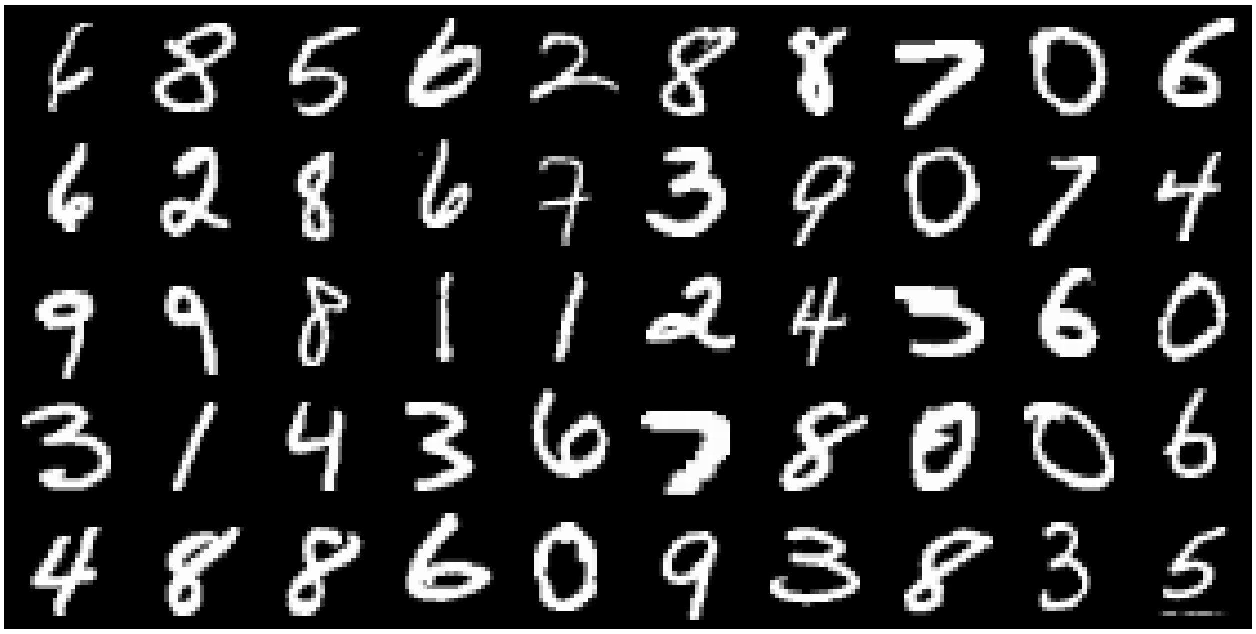
With only 50 iterations it can be seen that minimum value is 0 when $x = (1, 1)$.

Q2

a)

```r
library(grid)
load("mnist.tiny.RData")

train.X <- train.X / 255
test.X <- test.X / 255
grid.raster(array(aperm(array(train.X[1:50,], c(5, 10, 28, 28)), c(4, 1, 3, 2)),
                  c(140, 280)), interpolate = F)
```



First we run a SVM using a linear kernel:

```r
library(e1071)
set.seed(1)
svm(train.X, train.labels, type = "C", kernel = "linear", cross = 3)$tot.accuracy
```

```
## [1] 87.2
```

This results in an accuracy of 87.2%. Now we run a SVM using a polynomial kernel with different polynomial degrees:

```r
# degree = 2
set.seed(1)
svm(train.X, train.labels, type = "C-classification", kernel = "poly",
    degree = 2, coef = 1, cross = 3)$tot.accuracy
```

```
## [1] 81.5
```

```r
# degree = 6
set.seed(1)
svm(train.X, train.labels, type = "C-classification", kernel = "poly",
    degree = 6, coef = 1, cross=3)$tot.accuracy
```

```
## [1] 87.7
```

```r
# degree = 15
set.seed(1)
svm(train.X, train.labels, type = "C-classification", kernel = "poly",
    degree = 15, coef = 1, cross = 3)$tot.accuracy
```

```
## [1] 89.2
```

```r
# degree = 16
set.seed(1)
svm(train.X, train.labels, type = "C-classification", kernel = "poly",
    degree = 16, coef = 1, cross = 3)$tot.accuracy
```

```
## [1] 88.9
```

We can see that polynomial kernel of second degree has worse accuracy than the linear kernel. The 6th degree is the smallest that ensures that polynomial kernel is more accurate than linear kernel. For the 15th degree, accuracy is highest with 89.2%.

For the radial kernel:

```r
set.seed(1)
svm(train.X, train.labels, type = "C-classification", kernel = "radial",
    gamma = 1, coef = 1, cross = 3)$tot.accuracy
```

```
## [1] 12.4
```

```r
set.seed(1)
svm(train.X, train.labels, type = "C-classification", kernel = "radial",
    gamma = 0.03, coef = 1, cross = 3)$tot.accuracy
```

```
## [1] 90.9
```

For $gamma = 0.03$, the radial kernel provides a higher accuracy compared to the polynomial and linear kernels.

b)

To find the model with the best cross validation accuracy, various gamma and cost values are tested with 3-fold cross validation.

```r
log.C.range <- log(c(0.000001, 0.00001, 0.0001, 0.001, 0.001, 0.01,
                     0.1, 1, 10, 100, 100, 1000, 10000, 100000))
log.gamma.range <- log(c(0.000001, 0.00001, 0.0001, 0.001, 0.001, 0.01,
                         0.1, 1, 10, 100, 1000, 10000, 100000))

counter <- 0
combination<-matrix(nrow = length(log.C.range) * length(log.gamma.range), ncol = 2)
result_accurracy <- c()

# finds all possible pairs
for (i in 1:length(log.C.range)) {
  for(j in 1:length(log.gamma.range)) {
    counter <- counter + 1
    combination[counter,] <- c(gamma = exp(log.C.range[i]), exp(log.gamma.range[j]))
  }
}

# input cost and gamma into svm() for the 'rad' kernel
for (i in 1:nrow(combination)) {
  acc <- svm(train.X, train.labels, type = "C-classification", kernel = "rad",
             cost = combination[i, 1],  gamma = combination[i, 2], degree = 2,
             coef = 1, cross = 3)$tot.accuracy
  result_accurracy <- c(result_accurracy, acc)
}

# the highest accuracy obtained is:
max(result_accurracy)
```

```
## [1] 91.1
```

```r
# now we find the values of lc and lg that attain the maximum
h_nrow <- which.max(result_accurracy)
combination[h_nrow,]
```

```
## [1] 10.00  0.01
```

Doing this, we find that `(cost, gamma) = (10.00, 0.01)` provides an accuracy of 91.1% which was the the highest out of all the pairs of `cost` and `gamma` tested.

Now an `svm` is trained on the full training set using `(cost, gamma) = (10.00, 0.01)`.

```r
calculate_accuracy <- function(true, pred) {
  return(NA)
}

model <- svm(train.X, train.labels, type = "C-classification", kernel = "rad",
             cost = 10.0, gamma = 0.01, degree = 2, coef = 1)

test_set_predictions <- as.vector(predict(model, test.X), mode = 'numeric')

# Calculate a confusion matrix with the predicted and actual values from the test set.
# The values along the diagonal represent correct predictions and all other values are
# false-positives or false-negatives
confusion_matrix <- table(test.labels, test_set_predictions)
print(confusion_matrix)
```

```
##            test_set_predictions
## test.labels   1   2   3   4   5   6   7   8   9  10
##           0  93   0   1   0   0   2   0   0   1   0
##           1   0 115   0   0   0   0   3   0   4   0
##           2   1   1  72   3   0   0   1   2   0   0
##           3   1   1   0  88   0   4   1   2   1   1
##           4   0   0   0   0 103   0   1   1   0   3
##           5   0   1   0   5   1  75   1   0   2   0
##           6   1   1   0   0   0   0  94   0   0   0
##           7   0   0   2   1   0   0   0 102   1   7
##           8   0   4   1   2   2   2   2   0  78   2
##           9   2   0   0   1   3   0   0   6   2  93
```

```r
# calculate the test accuracy
accuracy <- sum(diag(confusion_matrix)) / sum(confusion_matrix)
print(accuracy)
```

```
## [1] 0.913
```

As we can see, the model results in a test accuracy of 91.3%.