Classification of Handwritten Digits

Edwin Ramirez & Kandace Mok

October 31, 2018

Contents

Create Image Matrices **1**

Determine Singular Value Decomoposition **2**

Express Test Images As A Linear Combination **3**

Compute the Least-Squares Error of Each Image **4**

Classify Each Image **4**

Calculate Overall Classification Rate **5**

Create Image Matrices

```
library("pracma")
library("MASS")

#Create data frame of training input
#Each column represents an image
Input <- data.frame(read.csv("trainInput.csv", header = FALSE))
digit <- read.csv("trainOutput.csv", header = FALSE)
dim(Input)
```

```
## [1]  256 1707
```

```
dim(digit)
```

```
## [1]    1 1707
```

```
#Bind training output to dataframe
trainInput <- rbind(Input, digit)
dim(trainInput)
```

```
## [1]  257 1707
```

First we load the data into a data frame, and subset the data into ten matrices for each of the ten different digits between 0 to 9. We tranpose the matrices so that each row signifies an image of the number, and the columns represent the 256 pixels of each image.

```
#Based on digits stored in the last row, separate the dataframe into 10 matrices
#Where each matrix contains the images of the corresponding digit recorded in the last row
#of the dataFrame. Remove last row
A0 = as.matrix(trainInput[,trainInput[257,] == 0])
A0 = A0[-c(257),]
A0 = t(A0)
```

```
A1 = as.matrix(trainInput[,trainInput[257,] == 1])
A1 = A1[-c(257),]
A1 = t(A1)

A2 = as.matrix(trainInput[,trainInput[257,] == 2])
A2 = A2[-c(257),]
A2 = t(A2)

A3 = as.matrix(trainInput[,trainInput[257,] == 3])
A3 = A3[-c(257),]
A3 = t(A3)

A4 = as.matrix(trainInput[,trainInput[257,] == 4])
A4 = A4[-c(257),]
A4 = t(A4)

A5 = as.matrix(trainInput[,trainInput[257,] == 5])
A5 = A5[-c(257),]
A5 = t(A5)

A6 = as.matrix(trainInput[,trainInput[257,] == 6])
A6 = A6[-c(257),]
A6 = t(A6)

A7 = as.matrix(trainInput[,trainInput[257,] == 7])
A7 = A7[-c(257),]
A7 = t(A7)

A8 = as.matrix(trainInput[,trainInput[257,] == 8])
A8 = A8[-c(257),]
A8 = t(A8)

A9 = as.matrix(trainInput[,trainInput[257,] == 9])
A9 = A9[-c(257),]
A9 = t(A9)
```

Determine Singular Value Decomoposition

First we reshape our matrices by transforming each digit matrix into square symmetrical matrices. We can calculate the right-singular matrices below:

```
#Calculate the right singular value matrices for each matrix A
V0 = t(A0)%*%A0
V0 = eigen(V0)$vectors

V1 = t(A1)%*%A1
V1 = eigen(V1)$vectors

V2 = t(A2)%*%A2
V2 = eigen(V2)$vectors

V3 = t(A3)%*%A3
```

```
V3 = eigen(V3)$vectors

V4 = t(A4)%*%A4
V4 = eigen(V4)$vectors

V5 = t(A5)%*%A5
V5 = eigen(V5)$vectors

V6 = t(A6)%*%A6
V6 = eigen(V6)$vectors

V7 = t(A7)%*%A7
v7 = eigen(V7)$vectors

V8 = t(A8)%*%A8
V8 = eigen(V8)$vectors

V9 = t(A9)%*%A9
V9 = eigen(V9)$vectors

A <- list(A1,A2,A3,A4,A5,A6,A7,A8,A9,A0)
V <- list(V1,V2,V3,V4,V5,V6,V7,V8,V9,V0)
```

Once we have calculated each of the right-singular matrices, we store them in a list `V`, and store the original matrices in a list `A` that we could use to calculate the singular values, and left-singular matrix `U` for each digit. However, we are only concerned with using the list `V` because it contains the sets of singular images for each digit. Therefore, we can ignore U and Σ in the following singular value decomposition equation:

$$A = U\Sigma V^T$$

Express Test Images As A Linear Combination

First we read in `testInput.csv` and `testOutput.csv` into our matrix `test`. We tranpose `test` so that each row signifies a test image to match the structure of our original training data.

```
#Express test images as a linear combination of the first k=20 singular images of each digit


testInput <- data.frame(read.csv("testInput.csv", header = FALSE))
testOut <- data.frame(read.csv("testOutput.csv", header = FALSE))
test <- rbind(testInput, testOut)
test <- as.matrix(test)
test <- t(test)
```

We want to express each test image as a linear combination of the first 20 singular images. Thus, if `V` is our set of singular image, and test is the single image we will be trying to identify, then this becomes a least squares problem where

$$A\vec{x} = \vec{b}$$

Thus in this form `V` would be equivalent to `A`, the test image would be equivalent to `b`, and `x` is our vector of error values. In order to classify the test images, we will have to form a linear combination for each digit for

every single test image. This means that every test image will have ten linear combinations, and in order to classify the test image, we need to find the shortest length of the error vector `x` in each linear combination. The linear combination with the smallest error will be the digit that test image is classified as.

Compute the Least-Squares Error of Each Image

The total least-square computations that need to be handled is 20,070 because we have to classify 2007 images. To handle this many least-squares calculations, we created the function below:

```
#Calculate Least Squares distance
error_lengths = function(number, test_image){
  #Take first 20 singular images
  #from the number matrix V we will be
  #using in our least square calculations.
  #So if number == 1, then take index 1 from
  #our list of V matrices in variable V
  #10 signifies 0 as index 10 of our list V
  #contains the right-singular matrix of digit 0
  M = V[[number]][,1:20]

  #Convert the test image into a 256x1 column matrix
  b = matrix(test_image[1:256], nrow = 256)
  dim(b)
  #calculate vector X
  x_hat = solve(t(M)%*%M)%*%(t(M)%*%b)

  distance = norm(b - (M%*%x_hat), '2')
  return(distance)
}
```

The function `error_lengths` calculates the length of the error vector `x` and returns the least-square error value. `x` is computed as follows:

$$\vec{x} = (A^T A)^{-1} * A^T b$$

And the length of the error vector is computed by the `norm()` function which calculates the length of the error vector as follows:

$$||\vec{x}|| = \sqrt{x_1^2 + x_2^2 + ...x_n^2}$$

Classify Each Image

We utilize nested `sapply()` statements to prevent delayed run-time of our program. The outer `sapply()`statement iterates through each test image in our `test` matrix, and the nested `sapply()`iterates each individual image through the digits 1 to 10, and computes the least-square error with our `error_lengths()` function. However, because the nested `sapply()` statement is within the `which.min` function, this statement will return the digit between 1 and 10 that had the smallest least-square error. Therefore, `classification` contains a list of all the classifications that we made for each test image in `test`.

```
classification = sapply(1:2007, function(j) which.min(sapply(1:10, function(x) error_lengths(x, test[j,]
```

It should be noted that our original data had ten digits between 0 to 9. Therefore we need to adjust our results, and edit each value that was recorded as 10 with a 0. This statement replaces 10 with 0, and leaves the digit alone otherwise.

```
classification = sapply(1:length(classification), function(x) if(classification[x] == 10){classificatio
```

We can then see that `classification` now contains a list of all the classifications that we made for each test image in `test`. Each of these values represents the digit that had the smallest residual when calculating the least-squares error of our test images.

```
print(classification[1:50])
```

```
##  [1] 9 6 3 0 6 0 0 0 6 9 6 2 3 4 0 3 1 6 9 6 2 2 4 9 6 2 0 5 8 3 7 0 0 0 7
## [36] 9 8 0 0 7 0 8 1 0 7 1 0 4 2 0
```

Calculate Overall Classification Rate

In order to calculate the overall classification rate, we compare classification to our test output, which is the 257th column of our `test` matrix. If we do a comparison and take the sum, we will have the total successful classifications. Once we have this number, we divide it by the total number of test images we had, which is 2007 and divide.

```
#Calculate overall accuracy rate
overall_rate = sum(classification == test[,257])/2007
overall_rate
```

```
## [1] 0.935725
```

This is our overall classification rate, but if we wanted to calculate the classification rate for each image, we can do this by creating a confusion matrix. The diagonal of this matrix indicates the number of times each digit was correctly classified. Thus, each non-diagonal element signifies misclassifications. Therefore, if we wanted to find the classifications of each digit, we simply take the number of correctly classified digits that lie on the diagonal, and divide them by their corresponding column sum.

```
#Create confusion Matrix
confusion = data.frame(classification, test[,257])
conf = table(confusion)
colnames(conf) <- c("zero", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine")
conf_columns = c("one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "zero")
conf
```

```
##                test...257.
## classification zero one two three four five six seven eight nine
##              0  355   0   8     1    1    8   2     0     4    0
##              1    0 259   1     0    1    1   0     2     0    1
##              2    2   0 179     5    0    2   0     2     2    0
##              3    0   0   2   148    0    4   0     0     6    1
##              4    1   3   3     1  188    0   2     4     0    5
##              5    0   0   0     8    1  140   2     1     1    0
##              6    0   2   0     0    1    0 164     0     0    0
##              7    0   0   1     0    1    0   0   128     0    0
##              8    0   0   4     2    0    2   0     1   149    2
##              9    1   0   0     1    7    3   0     9     4  168
```

We can compute each individual classification rate with the function below. However, it should be noted that the tenth element in vector `each_digit_rate` will contain the classification rate for digit 0.

```
digit_rates = function(column_num){

  if(column_num == 10){
```

```
    row = 0
  }else{
    row = column_num
  }
  correctly_classified = conf[row + 1, conf_columns[column_num]]
  digit_rate = correctly_classified/sum(conf[, conf_columns[column_num]])
  return(digit_rate)

}

each_digit_rate = sapply(1:10, function(x) digit_rates(x))
for (i in each_digit_rate){
  print(i)
}
```

```
## [1] 0.9810606
## [1] 0.9040404
## [1] 0.8915663
## [1] 0.94
## [1] 0.875
## [1] 0.9647059
## [1] 0.8707483
## [1] 0.8975904
## [1] 0.9491525
## [1] 0.9888579
```