

# Applications of the Compact Matrix Decomposition to the MNIST Dataset

*Elijah Verdoorn*

*December 16, 2016*

## Introduction

The MNIST dataset is a popular dataset for research in machine learning. Datasets like MNIST are used to check the validity of techniques and models before applying them to real-world prediction situations. The dataset consists of 70,000 images, each a digitization of a handwritten digit (LeCun, Cortes, and Burges 1998). The standard for working with this data is to use 60,000 of the image for training, reserving the other 10,000 for testing and validating the predictive models. Each image in MNIST is 28 pixels by 28 pixels, yielding 784 distinct values. These values range from 0 to 255 in base 10, each representing the amount of black in that pixel. Since the images are grayscale, we can represent them with only one matrix rather than more complicated color images, which can either be represented with three distinct matrices (one for each of the color channels) or by a single matrix with all values in base 16. This reduces the amount of computation needed to work with the dataset, with the discarded color information being classified as out of scope for the project. Predictive results on the data set by various groups across the world can be found on the internet, sorted by type of classification algorithm used to make predictions. Simple models such as linear classifiers have error rates as high as 12.0%, while the most complicated models listed are able to drive the error rate as low as .23%. Some of the classification models use pre-processing to make the data more suitable for use with the chosen method. Especially popular are normalization techniques such as deskewing. Since the dataset has 784 predictors, feature selection methods such as least absolute shrinkage and selection operators (LASSO) are frequently used to eliminate variables that do not contribute to the final solution. Examples of such variables in the case of the MNIST data would be the value in the very first pixel - it is totally white the vast majority of the time. Such a value does not improve our ability to predict what class our data belongs to, so we can safely ignore it. While some models are adept at handling large sparse data like MNIST, decompositions rising from linear algebra can perform similar functions while taking much less time to compute, a major victory that allows for computation on larger datasets. One such method, first published by

## Compact Matrix Decomposition

In an effort to improve computation speed, specifically around the training of new models, optimization techniques are applied. These techniques range in effectiveness and complexity, and are a very active area of mathematic research. One recently developed technique is the Compact Matrix Decomposition (CMD). The CMD was developed at Carnegie Mellon University, and first published in the paper “Less is More: Compact Matrix Decomposition for Large Sparse Graphs”. This paper has been cited 94 times since its publication in 2007 according to Google Scholar, increasing interest in the topic. The paper claims that the CMD requires less than 1/10 of the space of the more traditional Singular Value Decomposition (SVD) and is 10x more computationally efficient, while maintaining the same level of reconstruction accuracy. Here I seek to apply the CMD to the MNIST dataset. This is a prudent operation since the matrices of images, while not as large as the data used in the original paper (Sun et al. 2007)(200+ GB of network traffic data), are large enough to benefit from manipulating the data before we attempt to apply statistical models to it.

## Motivation

Initial computation on the MNIST data backs up the use of CMD. In my initial testing, I tried various models to get a baseline. Attempting to execute one such model, ridge regression in R on the dataset without any form of precomputation, resulted in a system failure as my computer ground to a halt under the weight of the computation. Using the singular value decomposition in this situation is not effective, since the data is too sparse and does not lend itself to singular values that provide useful information without other operations on them.

## Derivation

The derivation of CMD stems from a basic central question: how can one approximate the matrix  $A \in \mathbb{R}^{m \times n}$  as the product of smaller matrices  $C \in \mathbb{R}^{m \times c}$ ,  $U \in \mathbb{R}^{c \times r}$  and  $R \in \mathbb{R}^{r \times n}$  such that we minimize  $|A - CUR|^2$ . This computation effectively seeks to reduce the rank of  $A$  while maintaining enough similarity between the result and the original. Symbolically we represent the approximation of  $A$  as  $\tilde{A}$ , we then claim that  $\tilde{A} = CUR$ , with dimensions defined above. To begin computation, we construct a subspace of the input matrix  $A$ , then reduce the rank of that subspace. Since the subspace is spanned by the columns of the input matrix, we choose to sample from the input matrix. We choose to use a biased sampling method, favoring columns that have a higher squared Frobenius norm (Drineas, Kannan, and Mahoney 2006), which is defined by  $|A| = \sum_{i,j} A(i,j)^2$ . This selection method favors columns that have higher entries, and by allowing repeated selection of the same column we arrive at a set of column vectors whose distribution is modeled by the norms of the columns in  $A$ . We'll call this matrix  $C_d$ . We know and accept that the sampling method will result in repeated columns, which we remove without reducing accuracy. We choose to remove the duplicates so as to reduce storage space required to store the data and thus the computation time on the data.

We know that the columns that are duplicated contain more information than the non-repeated columns, as they were selected at a higher rate. This, of course, rests on the assumption that the higher values (when normed) are more important, which is appropriate since the goal is to apply this method to sparse matrices, which we expect to have many values at or near 0. To address the issues created if we simply remove the duplicate columns, we find the unique columns, and scale them based on the number of times they occur in the sampled set. For a scaling factor, we use the square root of the number of occurrences. This process results in a much narrower matrix, which is preferable for computation. We call this matrix  $C_s$ . Before further computation, we want to ensure that we have not lost information in our process thus far. As we intend on performing a singular value decomposition on  $C_s$ , we want to ensure that the singular values of  $C_s$  and  $C_d$  are the same. We first define another matrix  $D = [e_1, \dots, e_1, \dots, e_{c'}, \dots, e_{c'}]$ . Each  $e_i$  is a column vector that is all 0s, except for the  $i$ -th element, which is 1. In our definition of  $D$ , we also note that there is one  $e_i$  for each  $C_i$  in  $C_d$ , ensuring that  $C_d$  and  $D$  have the same number of columns.

From the above definitions, we know that  $C_d = CD^T$  where  $C$  is the matrix of unique columns of  $C_d$ . Then we have

$$C_d C_s^T = CD^T (CD^T)^T = CD^T DC^T = C\Lambda C^T = C\Lambda^{\frac{1}{2}}\Lambda^{\frac{1}{2}}C^T = C\Lambda^{\frac{1}{2}}(C\Lambda^{\frac{1}{2}})^T = C_s C_s^T$$

where  $\Lambda$  is a  $c' \times c'$  matrix that has the number of occurrences of each column  $e_i \in D$  on the diagonal and 0s elsewhere. From here, we can diagonalize either  $C_d C_d^T$  or  $C_s C_s^T$  to find the singular values of  $C_d$  and  $C_s$ .

The next steps in the process involve projecting the original matrix  $X$  onto the space spanned by  $C_s$ . We then will remove duplicate rows much like we did earlier with duplicate columns, reducing the computational cost. We begin by constructing an orthonormal basis for  $C$  using the SVD, so  $C = U_C \Sigma_C V_C^T$ . From there we could project the original matrix into the large and dense orthonormal basis  $U_C$ , but we choose to instead compute a low-rank approximation for  $A$ , stemming from the observation that  $U_C = CV_C \Sigma_C^{-1}$  where  $C \in \mathbb{R}^{m \times c}$  is large but sparse,  $V_C \in \mathbb{R}^{c \times k}$  is dense but small, and  $\Sigma \in \mathbb{R}^{k \times k}$  is small and diagonal. This approximation  $\tilde{A}$  is then defined as

$$\tilde{A} = U_C U_C^T A = CV_C \Sigma_C^{-1} (CV_C \Sigma_C^{-1})^T A = C(V_C \Sigma_C^{-2} V_C^T C^T) A = CTA$$

where  $T = (V_C \Sigma_C^{-2} V_C^T C^T) \in \mathbb{R}^{c \times m}$  we know that  $C$  is sparse. and  $T$  is dense and large. We can continue optimisizing the calculation by reducing the overhead between  $T$  and  $A$ .

The process of reducing this overhead is defined in the ApprMultiplication process, which takes two matrices  $A$  and  $B$ , assuming that  $AB$  is defined, and then sampling the columns of  $A$  and the rows of  $B$ . We scale the rows and columns for multiplication, but still have the issue of duplicates. The CMD process removes duplicate rows by sampling and scaling rows from  $A$  and extracting the corresponding columns from  $C^T$ .

After removing the duplicate rows, we need to verify that our matrix multiplication is still correct. To do this we let  $I$  and  $J$  be the set of sampled rows,  $I$  with duplicates and  $J$  without duplicates. We define  $J = [1, \dots, 1, \dots, r', \dots, r']$  where there are  $d'_i$  instances of each element up to  $d'_{r'}$ . Additionally, we define  $I = [1, \dots, r']$ . Letting  $A$  and  $B$  be matrices with dimensions  $m_a \times n_a$  and  $m_b \times n_b$  respectively. Allowing  $i \leq \min(n_a, m_b)$  we have

$$A(:, J)B(J, :) = A(:, I)\Lambda' B(I, :)$$

where  $\Lambda'$  is a diagonal matrix with  $1, \dots, d'_{r'}$ . We can prove that the matrix multiplication is correct by stating that

$$A(:, J)B(J, :) = \sum_{k \in J} A(i, k)B(k, j) = \sum_{k \in J} d_{i_k} A(i, k)B(k, j) = A(:, I)\Lambda' B(I, :).$$

This verifies the multiplication.

Having proved that the multiplication is correct, we can now simply calculate the final portion of the decomposition,  $U$ , to complete the process. In this case

$$U = V_C \Sigma_C^{-2} V_C^T C_s.$$

From here, we have calculated all the parts of the CMD that we need. We got  $C$  from the construction of the CMD subspace, calculated  $U$  by combining the results of the ApprMultiplication process with  $\Sigma_C$  and  $V_C$  from the SVD, and included the  $R$  from ApprMultiplication to complete the decomposition.

## Programming

To apply the CMD to the MNIST dataset, we program a CMD implementation in R.

```
library(dplyr)
library(tidyr)
library(tree)
library(randomForest)
library(e1071)
library(moments)
library(glmnet)
library(doParallel)
```

First we import libraries.

```
initial_subspace_construction <- function(mat, c) {
  columnDistribution <- vector(length = ncol(mat))
  matrixSum <- 0
  # sum the entire matrix, we only do this once for efficiency
  for (i in 1:ncol(mat)) {
    columnSum <- 0
    for (j in 1:nrow(mat)) {
      element <- mat[j, i] ^ 2
      columnSum <- columnSum + element
    }
  }
}
```

```

    }
    matrixSum <- matrixSum + columnSum
  }
  # calculate the distribution for each column
  for (i in 1:ncol(mat)) {
    columnSum <- 0
    for (j in 1:nrow(mat)) {
      element <- mat[j,i] ^ 2
      columnSum <- columnSum + element
    }
    columnDistribution[i] <- columnSum / matrixSum
  }
  returner <- matrix(nrow = nrow(mat), ncol = c) # the matrix that we'll return, of dimension m x c
  # perform biased sampling
  for (i in 1:c) {
    j <- sample(1:ncol(mat), size = 1, replace = TRUE, prob = columnDistribution)
    returner[,i] <- mat[,j] / sqrt(c * columnDistribution[j])
  }
  return(returner)
}

```

This function performs the initial subspace construction provided the initial matrix `mat` and a sample size `c`.

```

cmd_subspace_construction <- function(mat, c) {
  initSubspace <- initial_subspace_construction(mat, c)
  initSubspace <- apply(initSubspace, c(1,2), round, digits = 5) # round the initSubspace so that uni
  uniqueCols <- t(unique(t(initSubspace)))
  numUniqueCols <- ncol(uniqueCols)
  returner <- matrix(nrow = nrow(initSubspace), ncol = numUniqueCols)
  for (i in 1:numUniqueCols) {
    numInstancesInInitSubspace <- 0
    for (j in 1:ncol(initSubspace)) { # count the instances of this column in the initial subspace
      if (identical(initSubspace[,j], uniqueCols[,i])) {
        numInstancesInInitSubspace <- numInstancesInInitSubspace + 1
      }
    }
    returner[,i] <- sqrt(numInstancesInInitSubspace) * t(uniqueCols[,i])
  }
  return(list("cmdSubspace" = returner, "initSubspace" = initSubspace))
}

```

This function uses `initial_subspace_construction()` to construct the scaled version that we use, with the unique columns.

```

appr_multiplication <- function(matA, matB, sampleSize) {
  q <- vector(length = nrow(matB)) # the row distribution (I think??? TODO: check this)
  # sum the entire matB matrix, we only do this once for efficiency
  matrixSum <- 0
  # get the sum of the matrix so we only have to do it once
  for (i in 1:ncol(matB)) {
    columnSum <- 0
    for (j in 1:nrow(matB)) {
      element <- matB[j, i] ^ 2
    }
  }
}

```

```

        columnSum <- columnSum + element
      }
      matrixSum <- matrixSum + columnSum
    }
    # row distribution of matB
    for (x in 1:nrow(matB)) {
      rowSum <- 0
      for (i in 1:ncol(matB)) {
        rowSum <- rowSum + matB[x,i] ^ 2
      }
      q[x] <- rowSum / matrixSum
    }
    R_d <- matrix(nrow = sampleSize, ncol = ncol(matB))
    C_d <- matrix(nrow = nrow(matA), ncol = sampleSize)
    rowsSampled <- vector(length = sampleSize)
    for (i in 1:sampleSize) {
      j <- sample(1:nrow(matB), size = 1, replace = TRUE, prob = q)
      rowsSampled[i] <- j
      R_d[i,] <- matB[j,] / sqrt(sampleSize * q[j])
      C_d[,i] <- matA[,j] / sqrt(sampleSize * q[j])
    }
    uniqueColsC <- t(unique(t(C_d)))
    uniqueRowsR <- unique(R_d)
    R_s <- matrix(nrow = sampleSize, ncol = ncol(matB))
    C_s <- matrix(nrow = nrow(matA), ncol = sampleSize)
    for (i in 1:nrow(uniqueRowsR)) {
      u <- 0 # number of instances of this row in the full set of Rs
      for (j in 1:nrow(R_d)) { # count the instances of this column in the initial subspace
        if (identical(R_d[j,], uniqueRowsR[i,])) {
          u <- u + 1
        }
      }
      R_s[i,] <- u * uniqueRowsR[i,]
      C_s[,i] <- uniqueColsC[,i]
    }
    return(list("C" = C_d, "R" = R_s, "rowsSampled" = rowsSampled))
  }

```

This rather long function performs the ApprMultiplication process that we described above, creating two of the components that we need to use the decomposition in practice.

```

cmd_decomposition <- function(matA, c, r) {
  cSubspace <- cmd_subspace_construction(matA, c)

  # extract the init subspace for later use
  initSubspace <- cSubspace$initSubspace
  cSubspace <- cSubspace$cmdSubspace

  cSubspace.svd <- svd(t(initSubspace) %*% initSubspace)
  cTranspose <- t(initSubspace)
  apprMult <- appr_multiplication(cTranspose, matA, r)
  c_s <- apprMult$C
  r_s <- apprMult$R

```

```

    u <- cSubspace.svd$v %*% qr.solve(diag(cSubspace.svd$d) %*% diag(cSubspace.svd$d)) %*% t(cSubspace.svd$u)
    return(list("C" = initSubspace, "U" = u, "R" = apprMult$R, "rows" = apprMult$rowsSampled))
}

```

This summary function makes calls to the above functions, and performs the calculation of  $U$  to complete the decomposition.

## Application

Having derived and programmed the CMD algorithm, we move to applying it to the MNIST dataset. We begin by manipulating the data into a structure that makes sense in this situation:

### Setup

```

# Load the MNIST digit recognition dataset into R
# http://yann.lecun.com/exdb/mnist/
# assume you have all 4 files and gunzip'd them
# creates train$n, train$x, train$y and test$n, test$x, test$y
# e.g. train$x is a 60000 x 784 matrix, each row is one digit (28x28)
# call: show_digit(train$x[5,]) to see a digit.
# brendan o'connor - gist.github.com/39760 - anyall.org
# modified by Elijah Verdoorn (elijahverdoorn.com) 11/30/16
load_mnist <- function() {
  load_image_file <- function(filename) {
    ret = list()
    f = file(filename, 'rb')
    readBin(f, 'integer', n=1, size=4, endian='big')
    ret$n = readBin(f, 'integer', n=1, size=4, endian='big')
    nrow = readBin(f, 'integer', n=1, size=4, endian='big')
    ncol = readBin(f, 'integer', n=1, size=4, endian='big')
    x = readBin(f, 'integer', n=ret$n*nrow*ncol, size=1, signed=F)
    ret$x = matrix(x, ncol=nrow*ncol, byrow=T)
    close(f)
    ret
  }
  load_label_file <- function(filename) {
    f = file(filename, 'rb')
    readBin(f, 'integer', n=1, size=4, endian='big')
    n = readBin(f, 'integer', n=1, size=4, endian='big')
    y = readBin(f, 'integer', n=n, size=1, signed=F)
    close(f)
    y
  }
  # Modified to handle changed file names, ERV 11/30/16
  train <- load_image_file('train_set_images')
  test <- load_image_file('test_set_images')

  train$y <- load_label_file('train_set_labels')
  test$y <- load_label_file('test_set_labels')
}

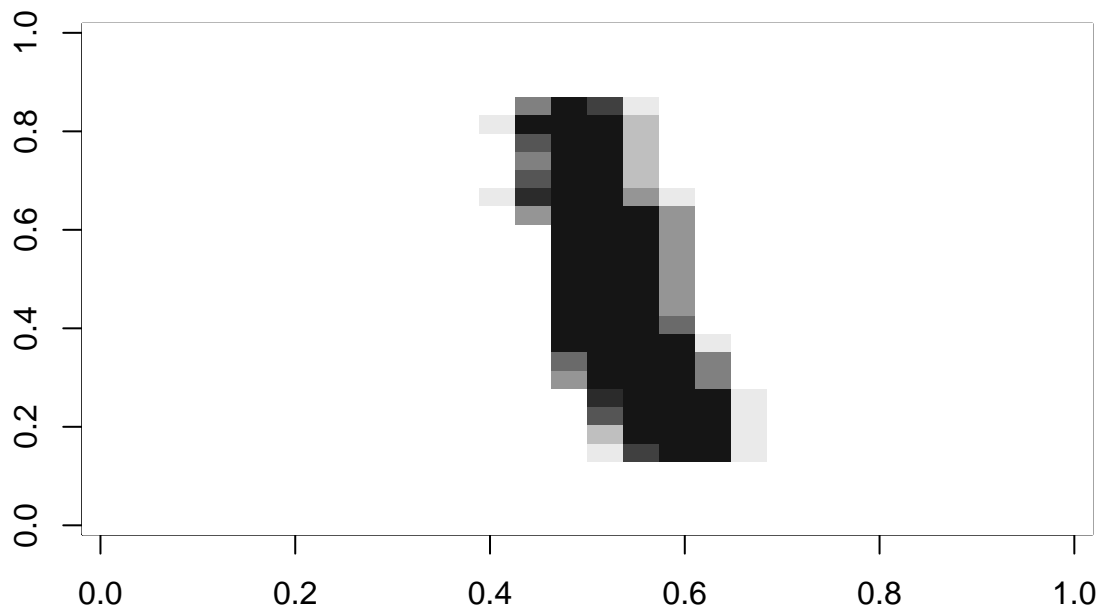
```

This function, used under the MIT license, imports the MNIST dataset into R (O'Connor 2015).

```
show_digit <- function(arr784, col=gray(12:1/12), ...) {  
  image(matrix(arr784, nrow=28)[,28:1], col=col, ...)  
}
```

This function can render the image matrices to the screen, allowing us to verify our calculations.

```
load_mnist()  
  
# Make data frames from the imported data described above  
train.df <- data.frame(train)  
test.df <- data.frame(test)  
  
# use the data frame to access the data and display images  
show_digit(train$x[7,])
```



This series of calls loads the data, then reformatted it into a data frame, making it easier to work with. After that, we call the `show_digit()` function to display an image as a test.

```
# average the images to get an idea of the distribution  
uniqueCols <- unique(train.df$y) # get the unique digits in the response col (y)  
# the matrix that will hold the averages  
avgVals <- matrix(nrow = length(uniqueCols), ncol = 784)  
for (i in 1:length(uniqueCols)) { # for each unique digit  
  # get values for the current digit, dropping the n and response columns
```

```

currentVals.df <- train.df[train.df$y == uniqueCols[i],]
drops <- c("n", "y")
currentVals.df <- currentVals.df[ , !(names(currentVals.df) %in% drops)]
for(j in 1:784) { # for each pixel (the images are 28x28, 28^2 = 784)
  avgVals[i,j] <- mean(currentVals.df[,j])
}
}

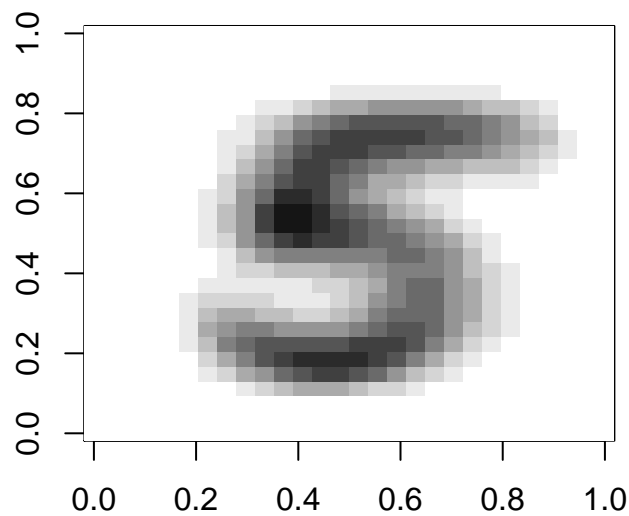
```

Given the matrices of images, to get a sense of how the digits are distributed, average all the instances of each digit.

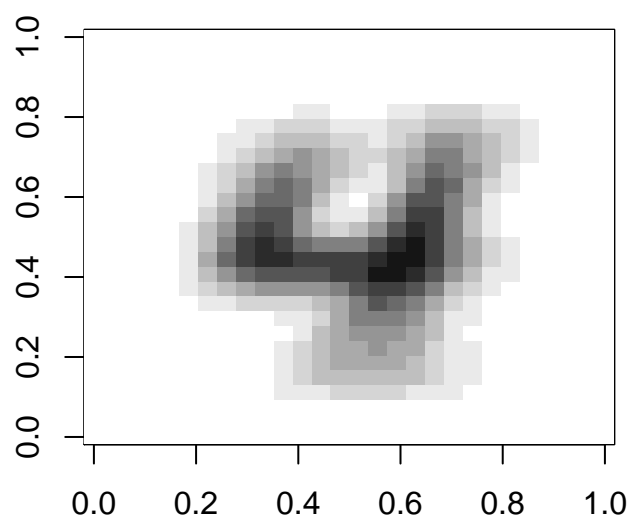
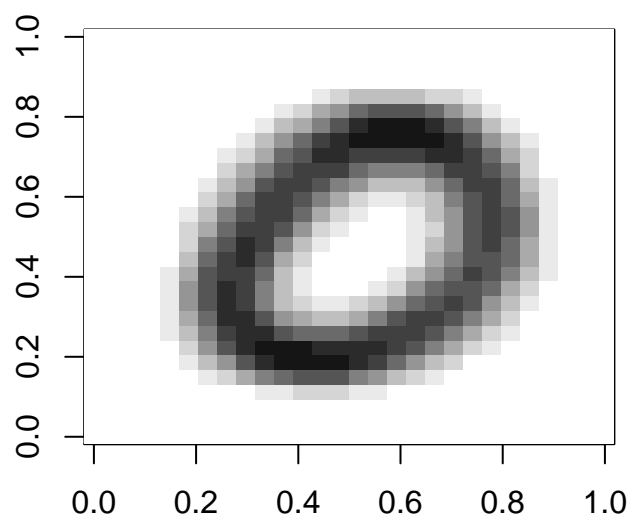
```

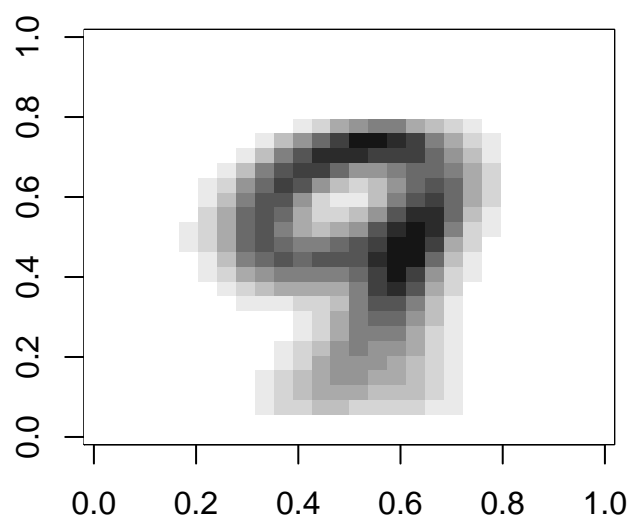
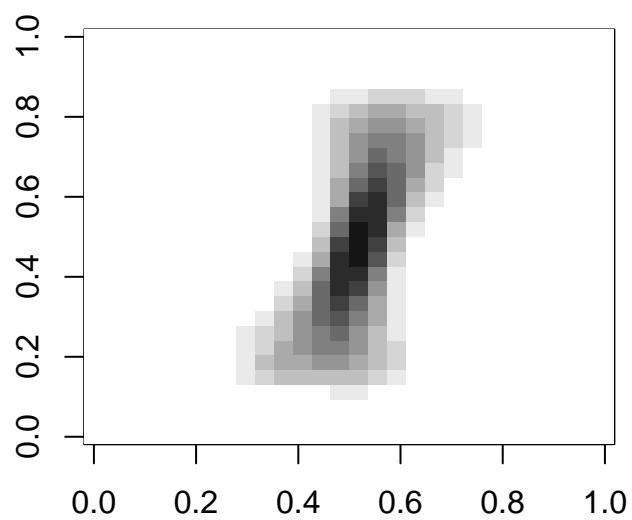
# display the image averages
for (i in 1:nrow(avgVals)) {
  show_digit(avgVals[i,])
}

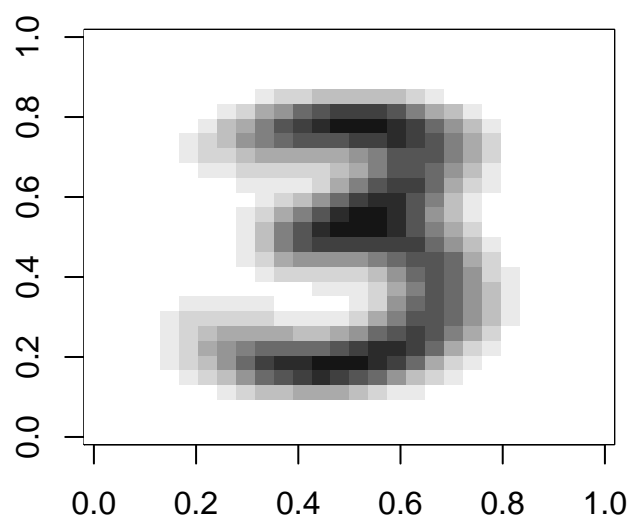
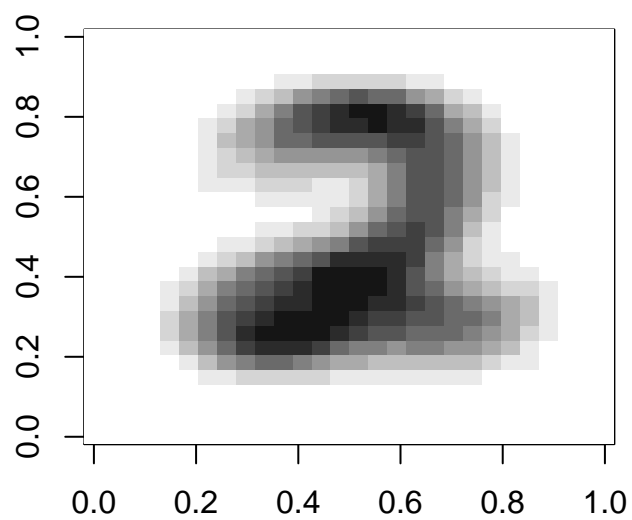
```

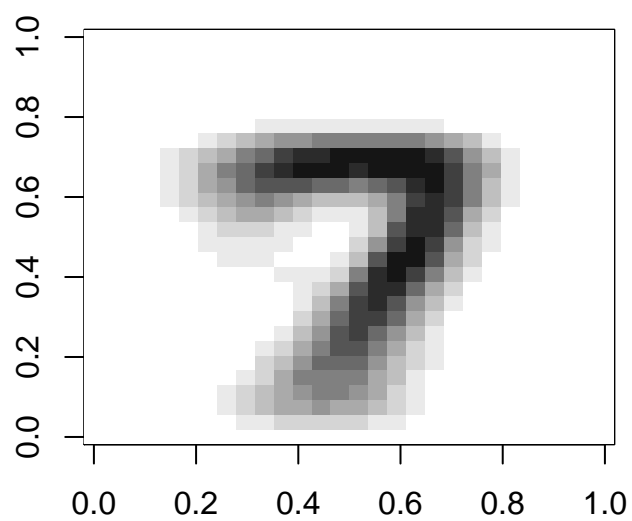
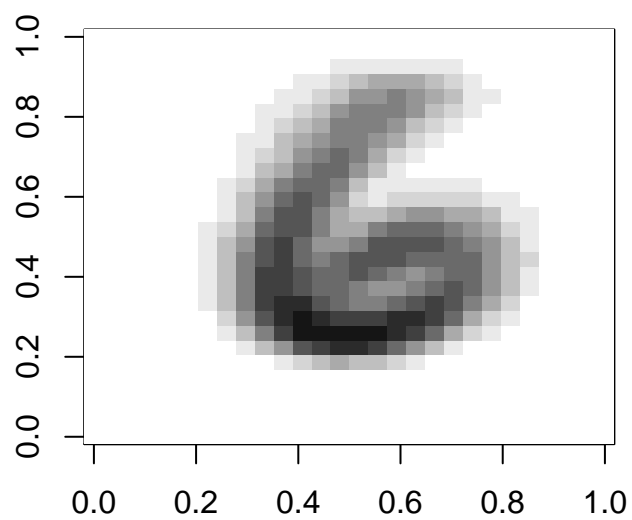


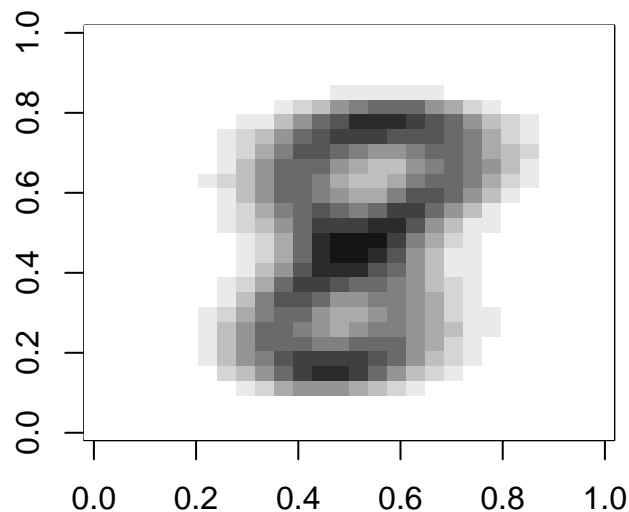












Display each average.

## CMD

```
# we need to do some precomputation here so that our models actually finish training. This is where CMD
c <- 78
r <- 6000
# drop the n column cause it's useless
drops <- c("n")
train.df <- train.df[, !(names(train.df) %in% drops)]
# add an id column
train.df$id <- seq.int(nrow(train.df))
decompMatrix <- as.matrix(train.df[,1:784])
decompTrain <- cmd_decomposition(decompMatrix, c, r)
```

This code applies the `cmd_decomposition()` function that we wrote earlier to the training dataset, which we will use to build our models.

```
responses <- train.df[decompTrain$rows, "y"]
cDecomp.df <- data.frame(decompTrain$C)
uDecomp.df <- data.frame(decompTrain$U)
rDecomp.df <- data.frame(decompTrain$R)

rTilde <- sample(1:nrow(decompMatrix), r, replace = FALSE)
cTilde <- sample(1:ncol(decompMatrix), c, replace = FALSE)
cDecompSubset <- decompTrain$C[rTilde, ]
```

```

uDecomp <- decompTrain$U
rDecompSubset <- decompTrain$R[, cTilde]
rDecompSubset[is.na(rDecompSubset)] <- 0
trainDecomp.df <- data.frame(cDecompSubset %*% uDecomp %*% rDecompSubset)
trainDecomp.df$y <- responses[rTilde]
# in theory we can build models on trainDecomp.df with y as the response variable

# this is most likely the wrong solution, but let's try anyway
testingData.df <- data.frame(t(decompTrain$U))
testingData.df$y <- train.df[decompTrain$rows, "y"] # can I build a model on this?

x.train <- as.matrix(testingData.df[, ! colnames(trainDecomp.df) %in% c("y")])
y.train <- as.matrix(testingData.df[, "y"])

```

Here we work with the results of the CMD, putting the data into a format that makes sense for the models that we want to build.

```

c <- 78
r <- 6000
# drop the n column cause it's useless
drops <- c("n")
test.df <- test.df[, !(names(test.df) %in% drops)]
# add an id column
test.df$id <- seq.int(nrow(test.df))
decompMatrix <- as.matrix(test.df[, 1:784])
decompTest <- cmd_decomposition(decompMatrix, c, r)

responses <- test.df[decompTest$rows, "y"]
cDecomp.df <- data.frame(decompTest$C)
uDecomp.df <- data.frame(decompTest$U)
rDecomp.df <- data.frame(decompTest$R)

rTilde <- sample(1:nrow(decompMatrix), r, replace = FALSE)
cTilde <- sample(1:ncol(decompMatrix), c, replace = FALSE)
cDecompSubset <- decompTest$C[rTilde, ]
uDecomp <- decompTest$U
rDecompSubset <- decompTest$R[, cTilde]
rDecompSubset[is.na(rDecompSubset)] <- 0
testDecomp.df <- data.frame(cDecompSubset %*% uDecomp %*% rDecompSubset)
testDecomp.df$y <- responses[rTilde]
# in theory we can build models on trainDecomp.df with y as the response variable

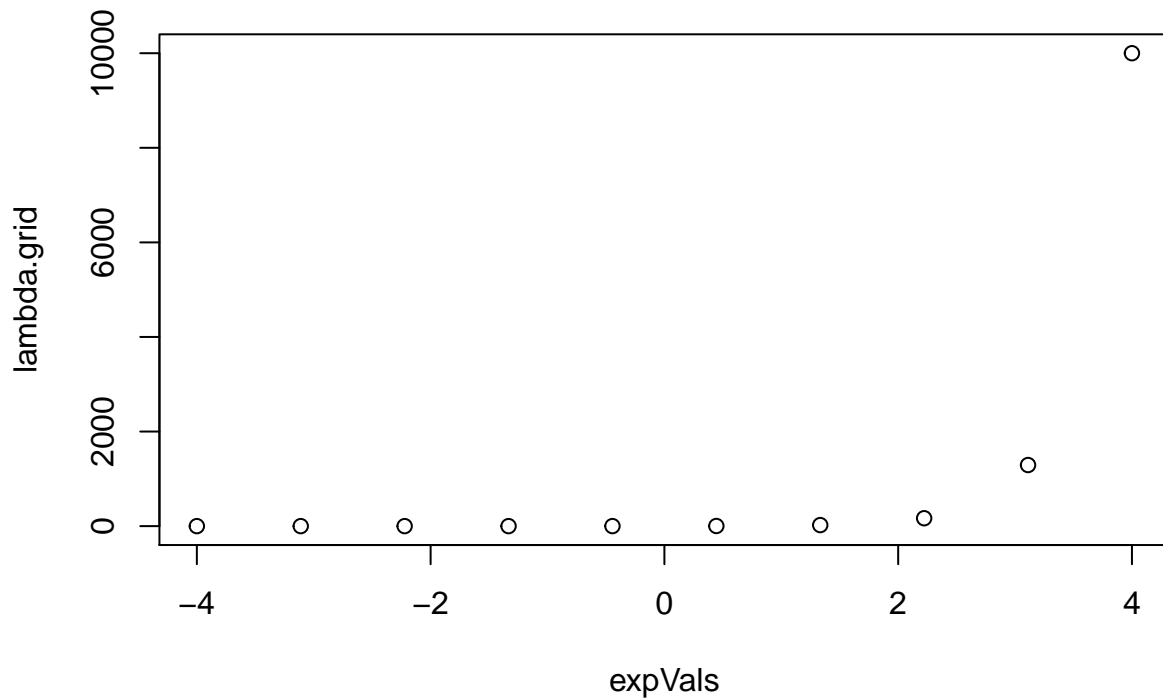
# this is most likely the wrong solution, but let's try anyway
testingData.df <- data.frame(t(decompTest$U))
testingData.df$y <- train.df[decompTest$rows, "y"] # can I build a model on this?

# Test
x.test <- as.matrix(testingData.df[, ! colnames(testingData.df) %in% c("y")])
y.test <- as.matrix(testingData.df[, "y"])

```

This code is very similar to the above work that we did on the training data, but now we do it for the testing data.

```
numLambda <- 10
expVals <- seq(-4,4,length=numLambda)
lambda.grid <- 10^expVals
plot(expVals,lambda.grid)
```

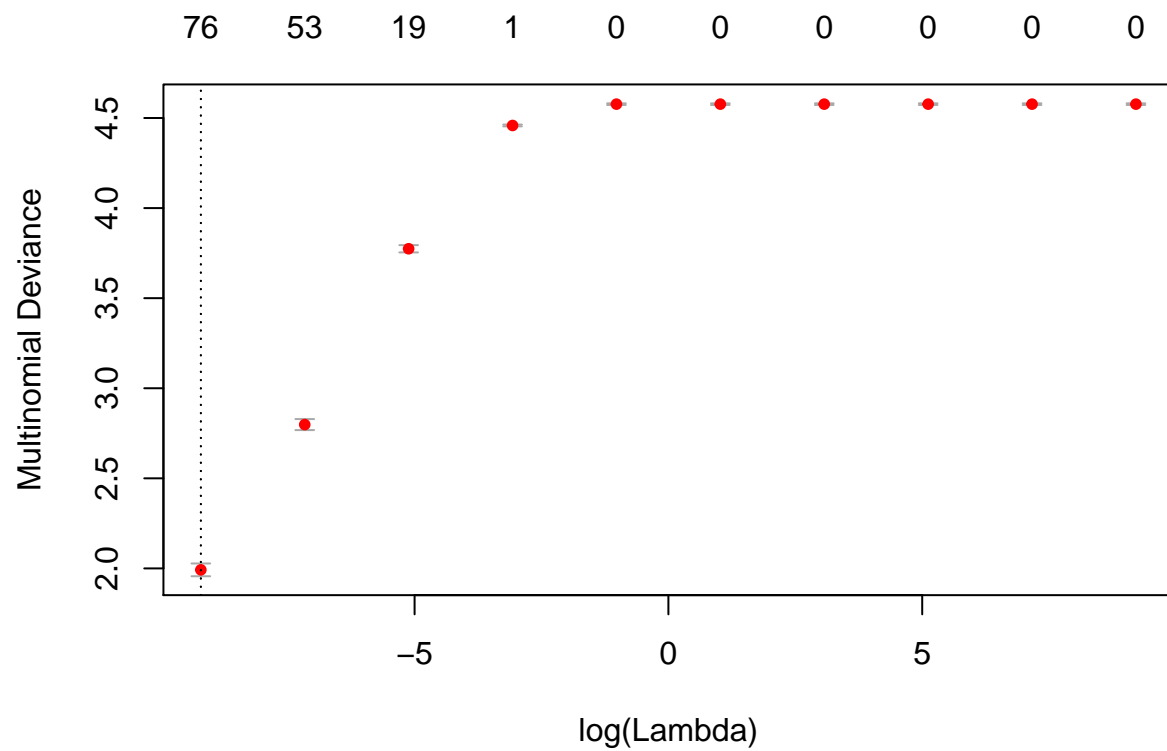


The cross-validation that we're going to do requires a "grid" of  $\lambda$  values that will be used as ridges in the algorithm to determine which variables contribute most to the prediction.

```
cv.ridge <- cv.glmnet(x = x.train, y = as.factor(y.train), lambda = lambda.grid, family = "multinomial")
```

```
## Warning: executing %dopar% sequentially: no parallel backend registered
```

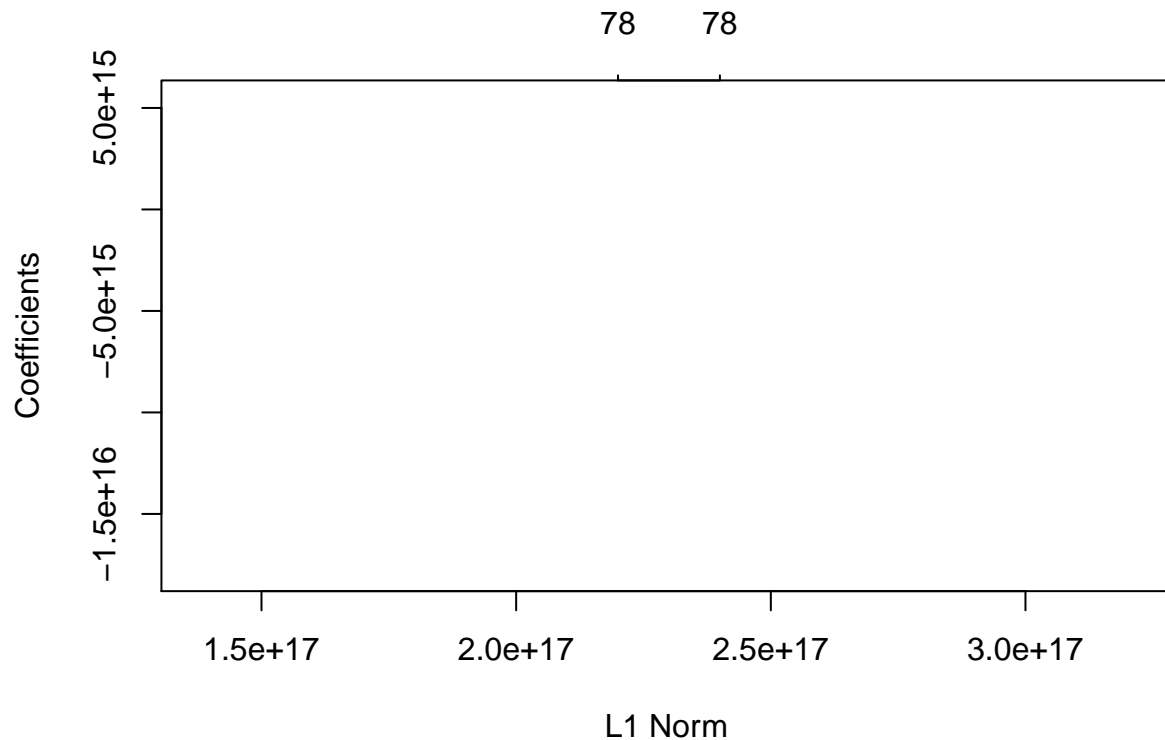
```
plot(cv.ridge)
```



```
bestLambda <- cv.ridge$lambda.min
mod.bestLambda <- glmnet(x.train, y.train, alpha = 0, lambda = bestLambda, intercept = TRUE)

# Ridge prediction for optimal lambda
phat <- predict(cv.ridge, newx = x.test, type = "class")
plot(mod.bestLambda)
```





Train the model using the grid of lambdas for cross-validation, then run predictions on the best of those models.

```
correct <- 0
for(i in 1:nrow(phat)) {
  if (phat[i] == y.test[i]) {
    correct <- correct + 1
  }
}
correct
```

```
## [1] 592
```

Here we find out how well we did by counting the number of predictions that we got correct.

## References

Drineas, Petros, Ravi Kannan, and Michael W. Mahoney. 2006. “Fast Monte Carlo Algorithms for Matrices I: Approximating Matrix Multiplication” 36 (1). Society for Industrial; Applied Mathematics: 132–57. [http://www.stat.berkeley.edu/~mmahoney/pubs/matrix1\\_SICOMP.pdf](http://www.stat.berkeley.edu/~mmahoney/pubs/matrix1_SICOMP.pdf).

LeCun, Yann, Corianna Cortes, and Christopher J.C. Burges. 1998. “The MNIST Database of Handwritten Digits.” [Thyann.lecun.com/exdb/mnist/](http://thyann.lecun.com/exdb/mnist/).

O’Connor, Brendan. 2015. “Load MNIST.” <https://gist.github.com/brendano/39760>.

Sun, Jimeng, Yinglian Xie, Hui Zhang, and Christos Faloutsos. 2007. *Less Is More: Compact Matrix*

*Decompositions for Large Sparse Graphs.* Edited by Carnegie Mellon University Library Staff. Carnegie Mellon University. <https://www.cs.cmu.edu/~jimeng/papers/SunSDM07.pdf>.