Handwritten digits classification by matrix factorization

Anton Antonov

9/27/2016

Introduction

This document (R-Markdown file) is made for the R-part of the MathematicaVsR project "Handwritten digits classification by matrix factorization".

The main goal of this document is to demonstrate how to do in R: - the ingestion images from binary files the MNIST database of images of handwritten digits, and - using matrix factorization to built a classifier, and - classifier evaluation by accuracy and F-score calculation.

The matrix factorization methods used are Singular Value Decomposition (SVD) and Non-negative Matrix Factorization (NMF).

Concrete steps

The concrete steps taken follow.

1. Ingest the **binary** data files into arrays that can be visualized as digit images.

- The MNIST database have two sets: $60,000$ training images and $10,000$ testing images.

2. Make a linear vector space representation of the images by simple unfolding.
3. For each digit find the corresponding representation matrix and factorize it.
4. Store the matrix factorization results in a suitable data structure. (These results comprise the classifier training.)

- One of the matrix factors is seen as a new basis.

5. For a given test image (and its linear vector space representation) find the basis that approximates it best. The corresponding digit is the classifier prediction for the given test image.
6. Evaluate the classifier(s) over all test images and compute accuracy, F-Scores, and other measures.

More details about the classification algorithm are given in the blog post "Classification of handwritten digits", [2]. The method and algorithm are described Chapter 10 of [3].

Libraries and source code used

```r
library(plyr)
library(ggplot2)
library(irlba)
```

```
## Loading required package: Matrix
```

```r
library(MASS)
library(data.table)
```

```
##
## Attaching package: 'data.table'
## The following objects are masked from 'package:reshape2':
##
##     dcast, melt
```

```r
library(doParallel)
```

```
## Loading required package: foreach
```

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

```r
library(devtools)
source_url("https://raw.githubusercontent.com/antononcube/MathematicaForPrediction/master/R/NonNegative
```

```
## SHA-1 hash of file is e8ee968e4c02573c76a248e9835bd151c80a5cce
```

Details of the classification algorithm

Training phase

1. Optionally re-size, blur, or transform in other ways each training image into an array.
2. Each image array (raster image) is linearized — the rows (or columns) are aligned into a one dimensional array. In other words, each raster image is mapped into a $\mathbf{R}^m$ vector space, where m is the number of pixels of a transformed image.

- We will call these one dimensional arrays *image vectors*.

3. From each set of images corresponding to a digit make a matrix with m columns of the corresponding image vectors.
4. Using the matrices in step 3 use a thin Singular Value Decomposition (SVD) to derive orthogonal bases that describe the image data for each digit.

Classification phase

1. Given an image of an unknown digit derive its image vector v in the same was as in the training phase.
2. Find the residuals of the approximations of v with each of the bases found in step 4 of the training phase.
3. The digit with the minimal residual is the classification result.

Using Non-negative Matrix Factorization

In order to use Non-negative Matrix Factorization (NMF) instead of SVD, the classification phase has to be modified since the obtained bases are not orthogonal. See below for theoretical and algorithmic details.

Handwritten digits data ingestion

First we download the files given in the MNIST database site :

- train-images-idx3-ubyte.gz: training set images (9912422 bytes)
- train-labels-idx1-ubyte.gz: training set labels (28881 bytes)
- t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)
- t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes)

Definitions of integestion functions

The following code follows very closely the R code for MNIST ingestion written by Brendan O'Connor (brendano).

```
ReadMNISTImages <- function( fileName, .progress = "none" ) {

  toRead <- file( fileName, "rb")

  ## magic number, number of images, number of rows, number of columns
  readInfo <- as.list( readBin( toRead, 'integer', n=4, size=4, endian="big") )
  names(readInfo) <- c("MNum", "NImages", "NRows", "NColumns")

  images <-
    llply( 1:readInfo$NImages, function(i) {
      mat <- matrix( readBin( toRead, 'integer', size = 1,
                              n= readInfo$NRows * readInfo$NColumns, endian="big", signed=F ),
                     readInfo$NRows, readInfo$NColumns )
      mat[, nrow(mat):1]
    }, .progress = .progress )

  close(toRead)
  images
}

ReadMNISTImageLabels <- function( fileName ) {

  toRead <- file(fileName, "rb")

  readLabelsInfo <- as.list( readBin( toRead, 'integer', n=2, size=4, endian="big") )
  names(readLabelsInfo) <- c("MNum", "NImages")

  labels = readBin(toRead, 'integer', n = readLabelsInfo$NImages, size=1, signed=F )

  close(toRead)
  labels
}
```

Ingestion

```
if( FALSE || !exists("trainImages") ) {

  cat( "\tTime to read training images :", system.time( {
    trainImages <- ReadMNISTImages("~/Datasets/MNIST/train-images-idx3-ubyte")
    trainImagesLabels <- ReadMNISTImageLabels("~/Datasets/MNIST/train-labels-idx1-ubyte")
  } ), "\n")

  cat( "\tTime to read test images :", system.time( {
    testImages <- ReadMNISTImages("~/Datasets/MNIST/t10k-images-idx3-ubyte")
    testImagesLabels <- ReadMNISTImageLabels("~/Datasets/MNIST/t10k-labels-idx1-ubyte")
  } ), "\n")

  names(trainImages) <- paste("train", 1:length(trainImages), sep = "-" )
  names(trainImagesLabels) <- paste("train", 1:length(trainImages), sep = "-" )

  names(testImages) <- paste("test", 1:length(testImages), sep = "-" )
  names(testImagesLabels) <- paste("test", 1:length(testImages), sep = "-" )
}
```

```
##  Time to read training images : 4.549 0.151 4.734 0 0
##  Time to read test images : 0.539 0.034 0.58 0 0
```

Verification statistics and plots

Training set

Number of images per digit for the training set:

```
count(trainImagesLabels)
```

```
##    x freq
## 1  0 5923
## 2  1 6742
## 3  2 5958
## 4  3 6131
## 5  4 5842
## 6  5 5421
## 7  6 5918
## 8  7 6265
## 9  8 5851
## 10 9 5949
```

Visualize the first 100 images with their labels of the training set:

```
par( mfrow = c(10,10), mai = c(0,0,0,0))
for(i in 1:100){
  image( trainImages[[i]], axes = FALSE, col = gray( 0:255 / 255 ) )
  text( 0.2, 0, trainImagesLabels[[i]], cex = 1.4, col = 2, pos = c(3,4))
}
```

5 0 4 1 9 2 1 3 1 4
3 5 3 6 1 7 2 8 6 9
4 0 9 1 1 2 4 3 2 7
3 8 6 9 0 5 6 0 7 6
1 8 7 9 3 9 8 5 9 3
3 0 7 4 9 8 0 9 4 1
4 4 6 0 4 5 6 1 0 0
1 7 1 6 3 0 2 1 1 7
9 0 2 6 7 8 3 9 0 4
6 7 4 6 8 0 7 8 3 1

(If order to make the plot above with inverted colors use 'col = gray(255:0 / 255)'.)

Testing set

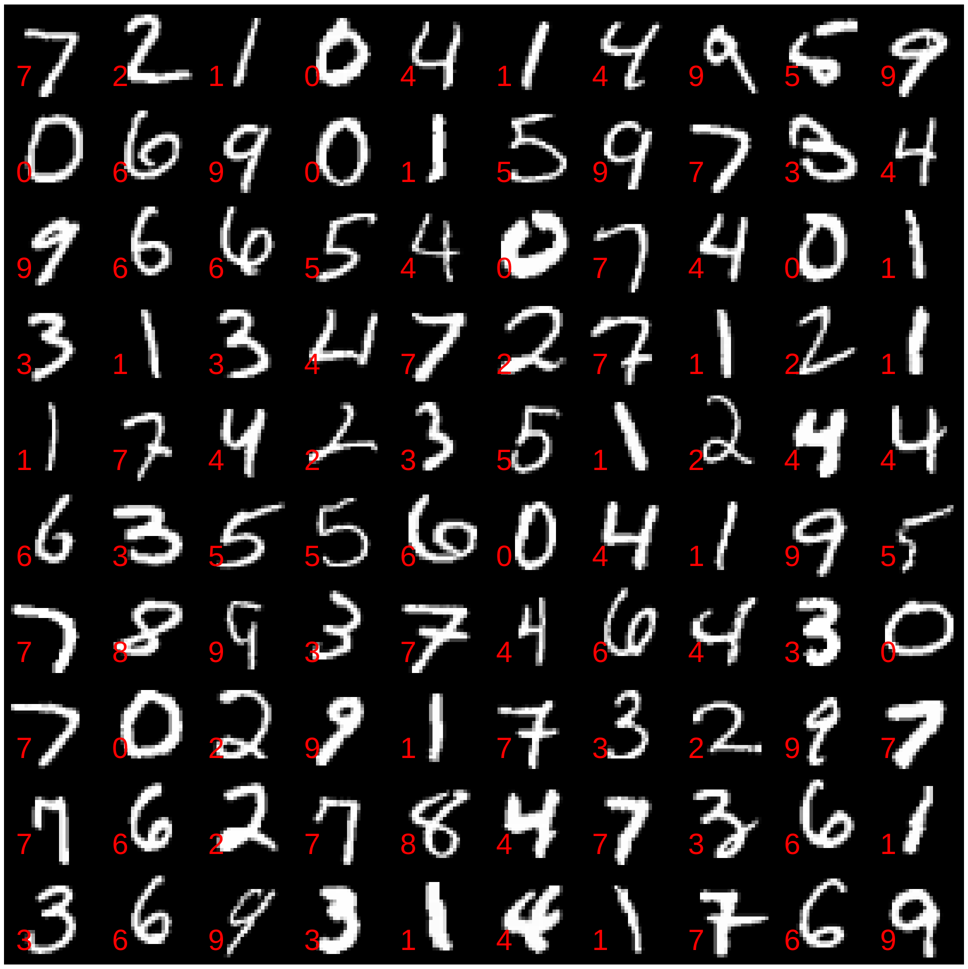
Number of images per digit for the testing set:

```
count(testImagesLabels)
```

```
##    x freq
## 1  0  980
## 2  1 1135
## 3  2 1032
## 4  3 1010
## 5  4  982
## 6  5  892
## 7  6  958
## 8  7 1028
## 9  8  974
## 10 9 1009
```

Visualize the first 100 images with their labels of the testing set:

```
par( mfrow = c(10,10), mai = c(0,0,0,0))
for(i in 1:100){
  image( testImages[[i]], axes = FALSE, col = gray( 0:255 / 255 ) )
  text( 0.2, 0, testImagesLabels[[i]], cex = 1.4, col = 2, pos = c(3,4))
}
```



(If order to make the plot above with inverted colors use `col = gray( 255:0 / 255 )`.)

Linear vector space representation

Each image is flattened and it is placed as a row in a matrix.

```
if ( FALSE || !exists("trainImagesMat") ) {
  cat( "\tTime to make the training images matrix:", system.time(
    trainImagesMat <- laply( trainImages, function(im) as.numeric( im ), .progress = "none" )
  ), "\n")
  cat( "\tTime to make the training images matrix:", system.time(
    testImagesMat <- laply( testImages, function(im) as.numeric( im ), .progress = "none" )
  ), "\n")
  classLabels <- sort( unique( trainImagesLabels ) )
}
```

```
##  Time to make the training images matrix: 41.89 1.275 43.661 0 0
##  Time to make the training images matrix: 3.582 0.152 3.759 0 0
```

Note here we also have computed the class labels from the data. (We expect them to be all digits from 0 to

9.)

Using Singular Value Decomposition

Factorization

The following code takes a sub-matrix for each digit and factorizes it using SVD. The factorization results are put in a list with named elements. (The element names being the digits.)

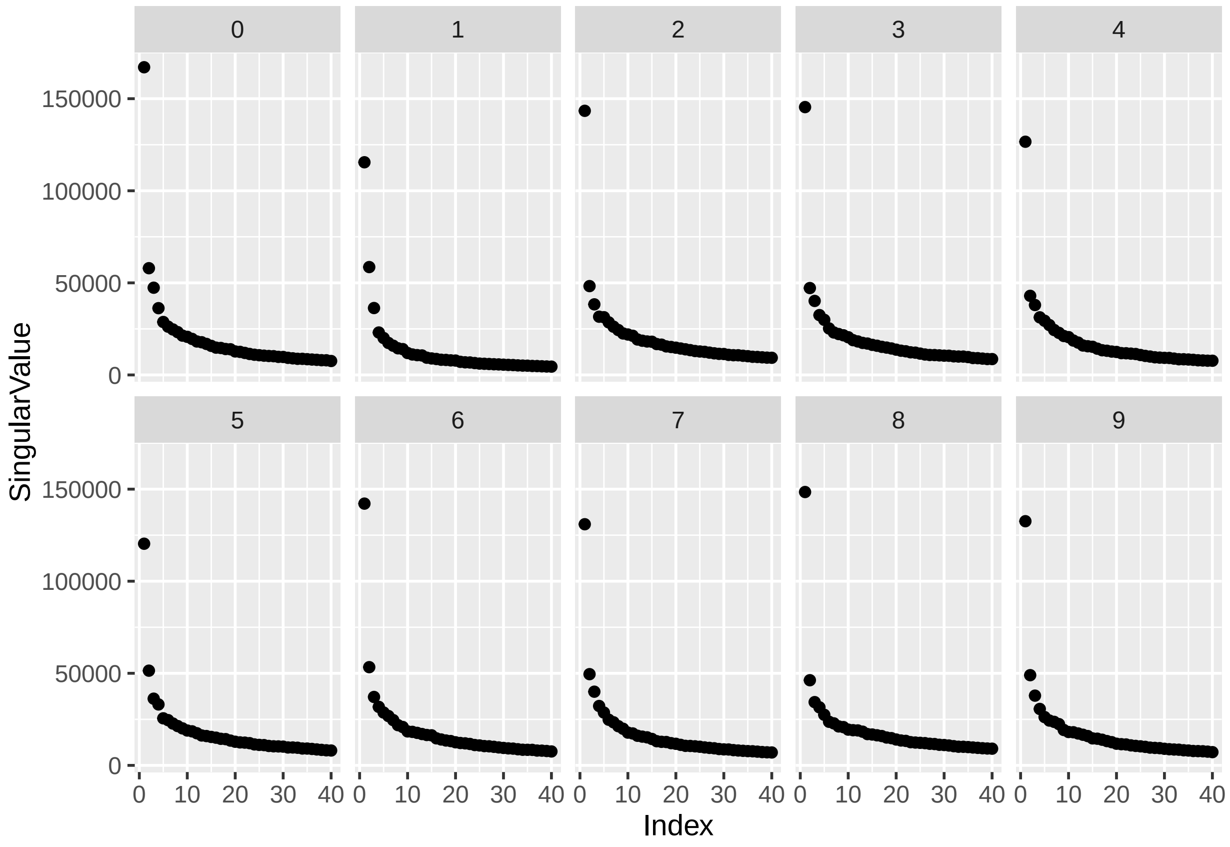
```
cat( "\tTime to SVD factorize the training images sub-matrices :", system.time( {
  svdRes <-
    llply( classLabels, function(cl) {
      inds <- trainImagesLabels == cl
      smat <- trainImagesMat[ inds, ]
      smat <- as( smat, "sparseMatrix")
      irlba( A = smat, nv = 40, nu = 40, maxit = 100, tol = 1E-6 )
    }, .progress = "none" )
  names(svdRes) <- classLabels
}), "\n")
```

```
##  Time to SVD factorize the training images sub-matrices : 5.463 0.342 5.842 0 0
```

Basis interpretation

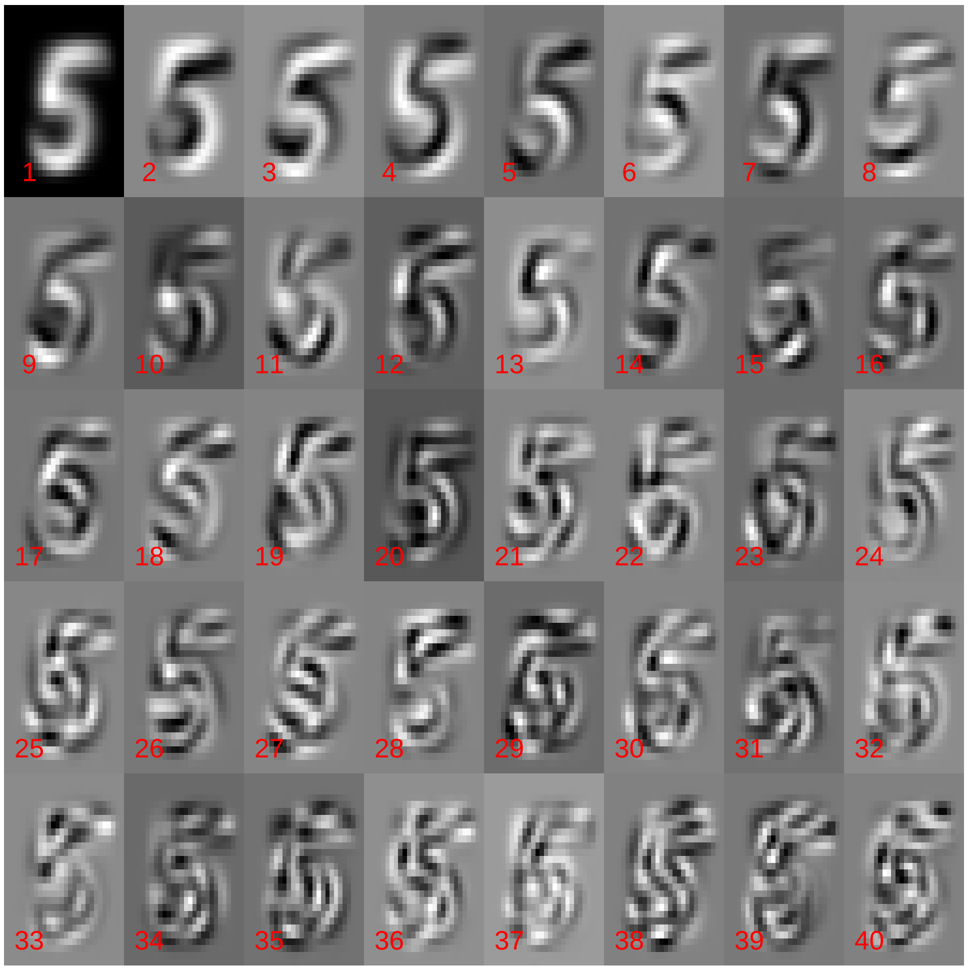
This multi-panel plot shows the singular values for each SVD factorization:

```
diagDF <- ldply( names(svdRes), function(x) data.frame( Digit = x, SingularValue = rev(sort(svdRes[[x]]$
ggplot(diagDF) + geom_point( aes( x = Index, y = SingularValue) ) + facet_wrap( ~ Digit, ncol = 5 )
```



Here is how the basis for 5 looks like:

```
dlbl <- "5"
U <- svdRes[[dlbl]]$U; V <- svdRes[[dlbl]]$v; D <- svdRes[[dlbl]]$d
par( mfrow = c(5,8), mai = c(0,0,0,0))
for(i in 1:40){
  image( matrix(V[,i], 28, 28), axes = FALSE, col = gray( 0:255 / 255 ) )
  text( 0.2, 0, i, cex = 1.2, col = 2, pos = c(3,4))
}
```



```r
par(mfrow=c(1,1), mai = c(1,1,1,1))
```

With SVD we get the first vector (corresponding to the largest singular value) to be the baseline image for a digit sub-set and the rest are corrections to be added or subtracted from that baseline image. Compare with NMF basis in which all vectors are interpret-able as handwritten images.

Defintion of the classification function

```r
vnorm <- function(x) sqrt( sum(x^2) )

#' @description Classify by representation over an SVD basis.
SVDClassifyImageVector <- function( factorizationsRes, vec ) {

  residuals <-
    laply( factorizationsRes, function(x){
      if( is.null(x$mean) ) { lv <- vec } else { lv <- vec - x$mean }
      rv <- lv - x$v %*% ( t(x$v) %*% lv )
      vnorm(rv)
    })
  names(residuals) <- names(factorizationsRes)

  list( Label = names(factorizationsRes)[ which.min(residuals) ], Residuals = residuals )
}
```

Here is an example classification with this function:

```
testImagesLabels[[123]]
```

```
## [1] 7
```

```
SVDClassifyImageVector( factorizationsRes = svdRes, vec = testImagesMat[123,] )
```

```
## $Label
## [1] "7"
##
## $Residuals
##         0         1         2         3         4         5         6
## 1214.8865  991.6804 1087.7768  955.3926  958.6550 1040.5943 1305.7999
##         7         8         9
##  544.9243 1041.6006  779.4606
```

Classification evaluation

The following command runs the classification function over each row (image) of the testing set matrix representation.

```
if(!exists("svdClRes")) {
  cat( "\tTime to compute the SVD classifier evalution :", system.time( {
    svdClRes <-
      ldply( 1:nrow(testImagesMat), function(i) {
        res <- SVDClassifyImageVector( factorizationsRes = svdRes, vec = testImagesMat[i,] )
        data.frame( Actual = testImagesLabels[[i]], Predicted = res$Label, stringsAsFactors = FALSE )
      }, .progress = "none")
  }), "\n")
}
```

```
##  Time to compute the SVD classifier evalution : 37.166 9.604 46.988 0 0
```

Overall accuracy:

```
mean( svdClRes$Actual == svdClRes$Predicted )
```

```
## [1] 0.957
```

Accuracy breakdown per digit:

```
svdClResDT <- data.table( svdClRes )
svdClResDT[ , .( .N, Accuracy = mean(Actual == Predicted) ), by = Actual]
```

```
##     Actual    N  Accuracy
##  1:      7 1028 0.9377432
##  2:      2 1032 0.9437984
##  3:      1 1135 0.9894273
##  4:      0  980 0.9928571
##  5:      4  982 0.9725051
##  6:      9 1009 0.9474727
##  7:      5  892 0.9316143
##  8:      6  958 0.9665971
##  9:      3 1010 0.9326733
## 10:      8  974 0.9507187
```

Confusion matrix:

```r
xtabs(~ Actual + Predicted, svdClRes)
```

```
##       Predicted
## Actual    0    1    2    3    4    5    6    7    8    9
##      0  973    0    0    0    0    0    2    1    4    0
##      1    0 1123    6    1    0    0    1    0    4    0
##      2    9    6  974    4    4    0    1   10   23    1
##      3    7    1   13  942    0   15    0    4   23    5
##      4    1    2    3    0  955    0    5    1    3   12
##      5    4    2    2   17    0  831    6    2   25    3
##      6    8    3    0    0    1   15  926    0    5    0
##      7    1    6   14    1    4    1    0  964    4   33
##      8    5    2    6   17    2    4    1    3  926    8
##      9    5    7    2    7   11    2    0    9   10  956
```

Using Non-negative Matrix Factorization

Factorization

I tried using the CRAN package NMF but it was too slow. Here we are using the implementation from MathematicaForPediction at GitHub.

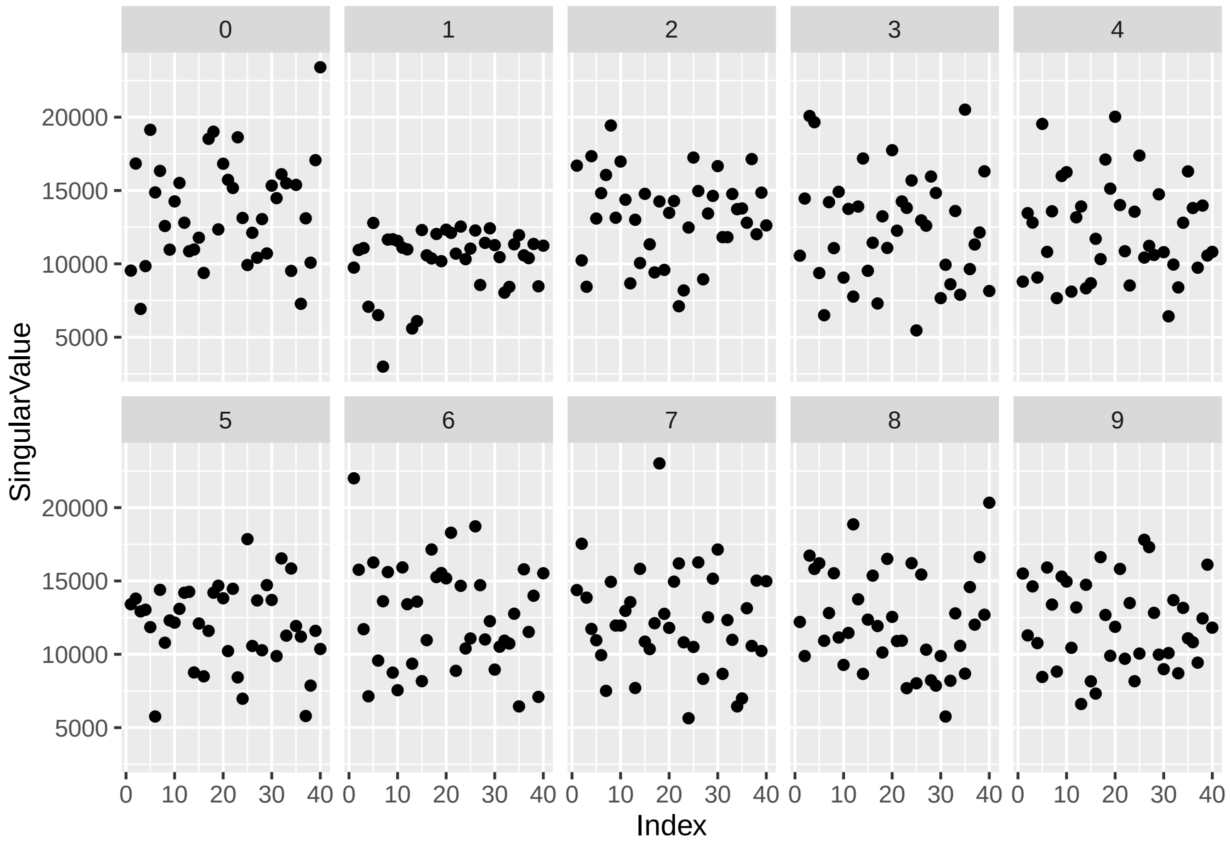
The following code takes a sub-matrix for each digit and factorizes it using SVD. The factorization results are put in a list with named elements. (The element names being the digits.)

```
if(!exists("nnmfRes")) {
  cat( "\tTime to compute the NMF of training images sub-matrices :", system.time( {
    nnmfRes <-
      llply( classLabels, function(cl) {
        inds <- trainImagesLabels == cl
        smat <- trainImagesMat[ inds, ]
        smat <- as( smat, "sparseMatrix")
        res <- NNMF( V = smat, k = 40, maxSteps = 20, tolerance = 1E-4, regularizationParameter = 0.1 )
        res <- NNMFNormalizeMatrixProduct( W = res$W, H = res$H, normalizeLeft = TRUE )
        bres <- NNMFNormalizeMatrixProduct( W = res$W, H = res$H, normalizeLeft = FALSE )
        bres$D <- aaply( as.matrix(res$H), 1, vnorm )
        bres$invH <- ginv( as.matrix( bres$H ) )
        bres$Wrn <- as( aaply( as.matrix(bres$W), 1, function(x) x / vnorm(x) ), "sparseMatrix")
        bres$M <- smat
        bres
      }, .progress = "none" )
    names(nnmfRes) <- classLabels
  }), "\n")
}
```

```
##  Time to compute the NMF of training images sub-matrices : 87.23 21.352 109.015 0 0
```

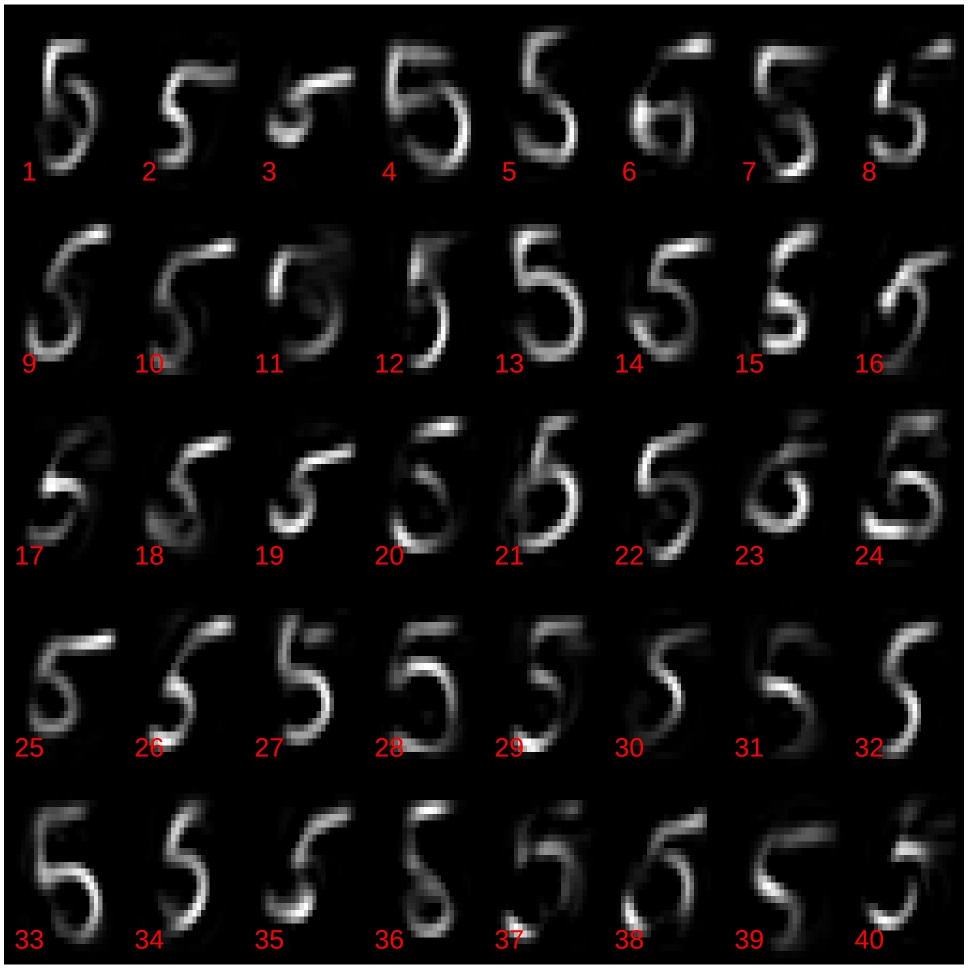
Basis interpretation

```
diagDF <- ldply( names(nnmfRes), function(x) data.frame( Digit = x, SingularValue = nnmfRes[[x]]$D, Inde
ggplot(diagDF) + geom_point( aes( x = Index, y = SingularValue) ) + facet_wrap( ~ Digit, ncol = 5 )
```



Here is how the basis for 5 looks like:

```
dlbl <- "5"
H <- nnmfRes[[dlbl]]$H
par( mfrow = c(5,8), mai = c(0,0,0,0))
for(i in 1:40){
  image( matrix(H[i,], 28, 28), axes = FALSE, col = gray( 0:255 / 255 ) )
  text( 0.2, 0, i, cex = 1.2, col = 2, pos = c(3,4))
}
```



```r
par(mfrow=c(1,1), mai = c(1,1,1,1))
```

As expected, with NMF the basis vectors are interpret-able as handwritten digits image “topics”.

Defintion of the classification function

Given a matrix $M \in \mathbf{R}^{n \times m}$ comprised of n image vectors, the classification process for NMF is more complicated than that with SVD because the rows of the factor H of the factorization $M = WH$ (the new basis) are not orthogonal to each other. Instead, for an image vector $v \in \mathbf{R}^m$ we have to look for the nearest neighbors in the matrix $W \in \mathbf{R}^{n \times k}$ of the vector $vH^{-1} \in \mathbf{R}^k$. The labels of those nearest neighbors are used to predict the label of v.

```r
#' @description Classify by representation over a NMF basis.
NNMFClassifyImageVector <- function( factorizationsRes, vec, numberOfTopBasisVectors = 4, distanceFuncti

  residuals <-
    laply( factorizationsRes, function(x) {
      approxVec <- matrix(vec,1) %*% x$invH
      if( distanceFunction == "euclidean" ) {
        dmat <- as.matrix(x$W) - matrix( rep(approxVec,nrow(x$W)), nrow = nrow(x$W), byrow = TRUE )
        ## inds <- order(apply( dmat, 1, vnorm))[1:numberOfTopBasisVectors]
        dmat <- rowSums(dmat^2)
        inds <- order(dmat)[1:numberOfTopBasisVectors]
      } else {
        approxVec <- approxVec[1,] / vnorm( approxVec[1,] )
        inds <- order( x$Wrn %*% approxVec )[1:numberOfTopBasisVectors]
      }
      approxVec <- colSums( x$M[inds,,drop=F] )
      rv <- vec / vnorm(vec) - approxVec / vnorm( approxVec )
      vnorm(rv)
    }, .progress = "none" )
  names(residuals) <- names(factorizationsRes)

  list( Label = names(factorizationsRes)[ which.min(residuals) ], Residuals = residuals )
}
```

Here is an example classification with this function:

```
testImagesLabels[[123]]
```

```
## [1] 7
```

```
NNMFClassifyImageVector( factorizationsRes = nnmfRes, vec = testImagesMat[123,] )
```

```
## $Label
## [1] "7"
##
## $Residuals
##         0         1         2         3         4         5         6
## 1.0525671 0.9812991 0.8692359 0.8986903 0.9256053 1.0295804 1.0668364
##         7         8         9
## 0.6226631 1.0814419 0.7259023
```

Classification evaluation

The classification process for NMF is slower and in order to speed it up we are going to use parallel computations.

```
if ( !exists("nnmfClRes") ) {
  if ( FALSE ) {

    cat( "\tTime to sequentielly compute the NMF classifier evalution :", system.time( {
      nnmfClRes <-
        ldply( 1:nrow(testImagesMat), function( i ) {
          res <- NNMFClassifyImageVector( factorizationsRes = nnmfRes, vec = testImagesMat[i,], numberO
          data.frame( Actual = testImagesLabels[[i]], Predicted = res$Label, stringsAsFactors = FALSE )
        }, .progress = "time" )
    }), "\n")

  } else {
    # Function definition for slicing a vector.
    Slice <- function(x, n) split(x, as.integer((seq_along(x) - 1) / n))

    if ( !exists("cl") || !("SOKcluster" %in% class(cl)) ) {
      mcCores <- 4
      cl <- makeCluster( mcCores )
      registerDoParallel( cl )
    }
    slicedIndsList <- Slice( 1:nrow(testImagesMat), ceiling( nrow(testImagesMat) / mcCores ) )

    startTime <- Sys.time()

    nnmfClRes <-
      foreach( parInds = slicedIndsList, .combine = rbind, .packages = c("Matrix") ) %dopar% {
        ldply( parInds, function( i ) {
          res <- NNMFClassifyImageVector( factorizationsRes = nnmfRes, vec = testImagesMat[i,], numberO
          data.frame( Actual = testImagesLabels[[i]], Predicted = res$Label, stringsAsFactors = FALSE )
        } )
      }

    endTime <- Sys.time()
    cat("\n\t\tNMF classification time in parallel on", mcCores, "cores is :", difftime( endTime, start
  }
}
```

```
##
##      NMF classification time in parallel on 4 cores is : 337.0186
```

Overall accuracy:

```
mean( nnmfClRes$Actual == nnmfClRes$Predicted )
```

```
## [1] 0.9677
```

Accuracy breakdown per digit:

```
nnmfClResDT <- data.table( nnmfClRes )
nnmfClResDT[ , .( .N, Accuracy = mean(Actual == Predicted) ), by = Actual]
```

```
##     Actual    N  Accuracy
##  1:      7 1028 0.9474708
##  2:      2 1032 0.9757752
##  3:      1 1135 0.9850220
##  4:      0  980 0.9826531
##  5:      4  982 0.9714868
##  6:      9 1009 0.9504460
##  7:      5  892 0.9573991
##  8:      6  958 0.9801670
##  9:      3 1010 0.9613861
## 10:      8  974 0.9630390
```

Confusion matrix:

```
xtabs(~ Actual + Predicted, nnmfClRes)
```

```
##       Predicted
## Actual    0    1    2    3    4    5    6    7    8    9
##      0  963    0    5    0    0    1    8    1    2    0
##      1    0 1118    5    1    0    1    3    1    6    0
##      2    3    0 1007    2    0    0    1   11    8    0
##      3    0    1    6  971    1   10    0    5   13    3
##      4    0    1    3    0  954    0    4    1    2   17
##       5    2    0    1   13    0  854    4    1   11    6
##       6    5    2    0    0    3    5  939    0    3    1
##       7    0    5   15    0    5    1    0  974    2   26
##       8    1    0    2   16    3    7    2    2  938    3
##       9    3    5    1    8   15    2    1   14    1  959
```

References

[1] Yann LeCun et al., MNIST database site.

[2] Anton Antonov, “Classification of handwritten digits”, (2013), blog post at MathematicaForPrediction at WordPress.

[3] Lars Elden, Matrix Methods in Data Mining and Pattern Recognition, 2007, SIAM.