

# Handwritten digits classification by matrix factorization

Anton Antonov

MathematicaVsR project at GitHub

September, 2016

---

## Introduction

This document is made for the *Mathematica*-part of the MathematicaVsR project “Handwritten digits classification by matrix factorization”.

The main goal of this document is to demonstrate how to do in *Mathematica*:

- (i) the ingestion images from binary files the MNIST database of images of handwritten digits, and
- (ii) using matrix factorization to built a classifier, and
- (iii) 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].

## 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  $\mathbb{R}^m$  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 way 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 (<http://yann.lecun.com/exdb/mnist/>) :

- train-images-idx3-ubyte.gz (<http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz>): training set images (9912422 bytes)
- train-labels-idx1-ubyte.gz (<http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz>): training set labels (28881 bytes)
- t10k-images-idx3-ubyte.gz (<http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz>): test set images (1648877 bytes)
- t10k-labels-idx1-ubyte.gz (<http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz>): test set labels (4542 bytes)

# Handwritten digits data ingestion

## Definitions

```

Clear[ReadMNISTImages]
ReadMNISTImages[fileName_String, maxImages_: All] :=
  Block[{n, toRead, readInfo, raw, images},
    If[
      TrueQ[maxImages === All] || TrueQ[maxImages === Automatic], n = ∞, n = maxImages];
    toRead = OpenRead[fileName, BinaryFormat → True];
    readInfo = BinaryReadList[toRead, "Integer32", 4, ByteOrdering → 1];
    readInfo =
      AssociationThread[{"MNum", "NImages", "NRows", "NCOLUMNS"} → readInfo];
    images = Table[
      raw = BinaryReadList[toRead, "UnsignedInteger8",
        readInfo["NRows"] * readInfo["NCOLUMNS"], ByteOrdering → 1];
      Partition[raw, readInfo["NCOLUMNS"]]
      , {i, Min[n, readInfo["NImages"]]}];
    Close[toRead];
    images
  ];

Clear[ReadMNISTImageLabels]
ReadMNISTImageLabels[fileName_String, maxImages_: All] :=
  Block[{n, toRead, readInfo, labels},
    If[
      TrueQ[maxImages === All] || TrueQ[maxImages === Automatic], n = ∞, n = maxImages];
    toRead = OpenRead[fileName, BinaryFormat → True];
    readInfo = BinaryReadList[toRead, "Integer32", 2, ByteOrdering → 1];
    readInfo = AssociationThread[{"MNum", "NImages"} → readInfo];
    labels = Table[
      First@BinaryReadList[toRead, "UnsignedInteger8", 1, ByteOrdering → 1]
      , {i, Min[n, readInfo["NImages"]]}];
    Close[toRead];
    labels
  ];

```

## Ingestion

```

AbsoluteTiming[
  trainImages = ReadMNISTImages["~/Datasets/MNIST/train-images-idx3-ubyte"];
]
{0.781196, Null}

```

```

AbsoluteTiming[
  trainImagesLabels =
    ReadMNISTImageLabels["~/Datasets/MNIST/train-labels-idx1-ubyte"];
]
{0.125052, Null}

AbsoluteTiming[
  testImages = ReadMNISTImages["~/Datasets/MNIST/t10k-images-idx3-ubyte"];
]
{0.204194, Null}

AbsoluteTiming[
  testImagesLabels =
    ReadMNISTImageLabels["~/Datasets/MNIST/t10k-labels-idx1-ubyte"];
]
{0.022582, Null}

```

## Verification statistics and plots

Number of images in the training set :

```

Tally[Dimensions /@ trainImages]
{{{28, 28}, 60 000}}

```

Number of images in the testing set:

```

Tally[Dimensions /@ testImages]
{{{28, 28}, 10 000}}

```

Number of images per digit for (1) the training set, and (2) the testing set:

```





























































Row[{GridTableForm[SortBy[#, 1] &@Tally[trainImagesLabels],
  TableHeadings → {"Label", "Count"}], " ", GridTableForm[
  SortBy[#, 1] &@Tally[testImagesLabels], TableHeadings → {"Label", "Count"}]]]

```

| #  | Label | Count | #  | Label | Count |
|----|-------|-------|----|-------|-------|
| 1  | 0     | 5923  | 1  | 0     | 980   |
| 2  | 1     | 6742  | 2  | 1     | 1135  |
| 3  | 2     | 5958  | 3  | 2     | 1032  |
| 4  | 3     | 6131  | 4  | 3     | 1010  |
| 5  | 4     | 5842  | 5  | 4     | 982   |
| 6  | 5     | 5421  | 6  | 5     | 892   |
| 7  | 6     | 5918  | 7  | 6     | 958   |
| 8  | 7     | 6265  | 8  | 7     | 1028  |
| 9  | 8     | 5851  | 9  | 8     | 974   |
| 10 | 9     | 5949  | 10 | 9     | 1009  |

## Training and testing images samples


```
In[1059]:= {n, k} = {30, 6};
Grid[{"Training images sample", "Testing images sample"}, Magnify[#, 1] &@
  Grid[Partition[MapThread[ColumnForm[{Style[#2, Red, Small], #1}] &,
    {(*ColorNegate@***)Image /@ Take[#[[1]], n], Take[#[[2]], n]}], k],
    Dividers -> All, FrameStyle -> LightGray] & /@
  {{trainImages, trainImagesLabels}, {testImages, testImagesLabels}}}]
```

| Training images sample  |   |   |   |   |   | Testing images sample   |   |   |  |   |   |
|---|---|---|---|---|---|---|---|---|--|---|---|
| 5   | 0   | 4   | 1   | 9   | 2   | 7   | 2   | 1   | 0  | 4   | 1   |
|    |    |    |    |    |    |    |    |    |    |    |    |
| 1   | 3   | 1   | 4   | 3   | 5   | 4   | 9   | 5   | 9  | 0   | 6   |
|    |    |    |    |    |    |    |    |    |    |    |    |
| 3   | 6   | 1   | 7   | 2   | 8   | 9   | 0   | 1   | 5  | 9   | 7   |
|    |    |    |    |    |    |    |    |    |    |    |    |
| 6   | 9   | 4   | 0   | 9   | 1   | 3   | 4   | 9   | 6  | 6   | 5   |
|    |    |    |    |    |    |    |    |    |    |    |    |
| 1   | 2   | 4   | 3   | 2   | 7   | 4   | 0   | 7   | 4  | 0   | 1   |
|  |  |  |  |  |  |  |  |  |  |  |  |


## Linear vector space representation

Each image is flattened and it is placed as a row in a matrix. Note here we also have computed the class labels from the data. (We expect them to be all digits from 0 to 9.)

```
trainImagesMat = N@SparseArray[Flatten /@ trainImages]
```

```
SparseArray[
  +  Specified elements: 8994156
  Dimensions: {60000, 784}
]
```

```
testImagesMat = N@SparseArray[Flatten /@ testImages]
```

```
SparseArray[
  +  Specified elements: 1511219
  Dimensions: {10000, 784}
]
```

```
classLabels = Sort@Union[trainImagesLabels]
```

```
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

---

## Using Singular Value Decomposition

### The matrix factorization

The following code takes a sub-matrix for each digit and factorizes it using SVD. The factorization results are put in an association. (The keys being the digits.)

```
AbsoluteTiming[
  svdRes =
    Map[Function[{ cl},
      pos = Flatten@Position[trainImagesLabels, cl];
      SingularValueDecomposition[trainImagesMat[[pos, All]], 40, Tolerance  $\rightarrow$   $10^{-6}$ ]
    ], classLabels];
  svdRes = AssociationThread[classLabels  $\rightarrow$  svdRes];
]
{31.0659, Null}

Keys[svdRes]
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

### Basis interpretation

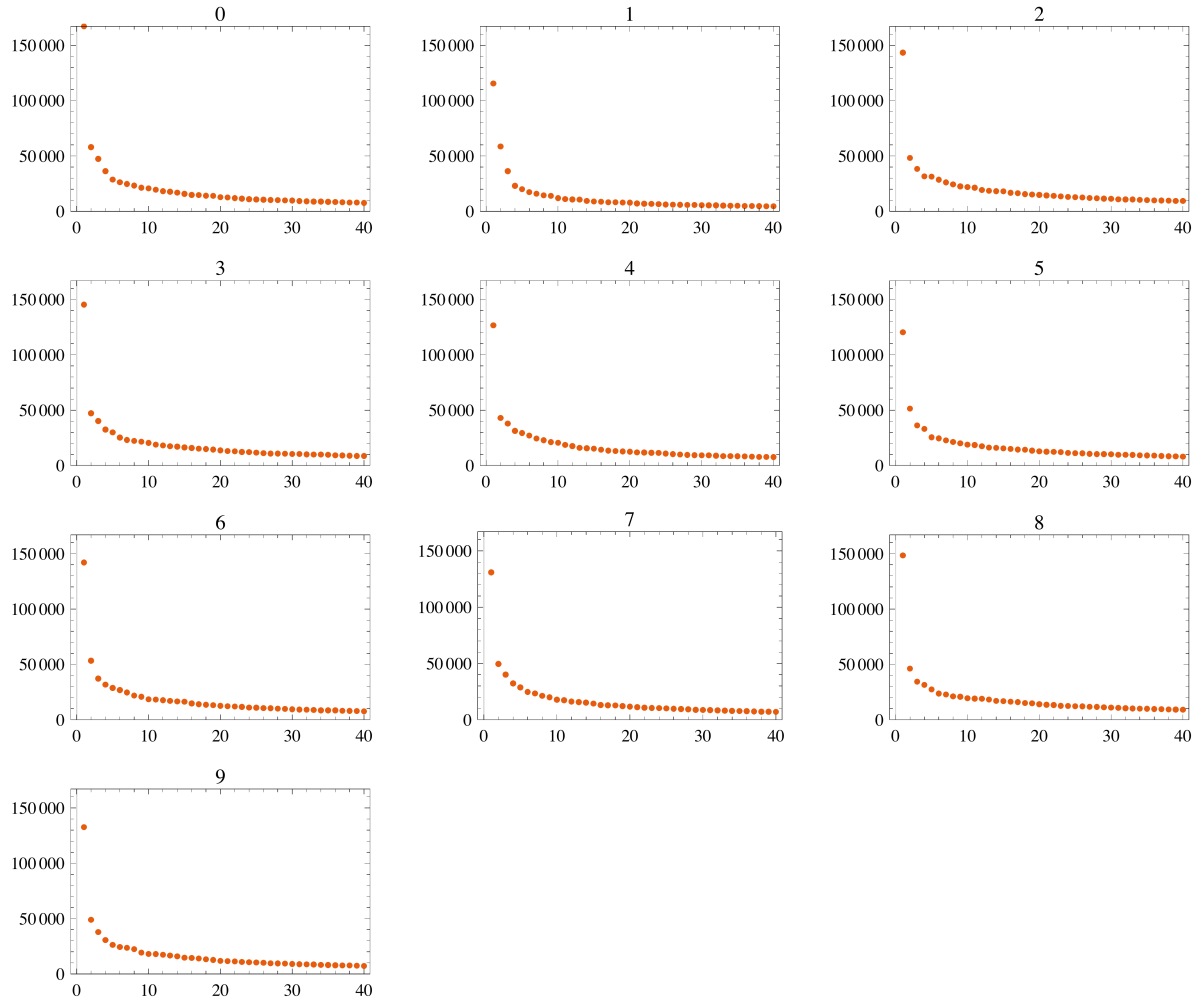
This multi-panel plot shows the singular values for each SVD factorization:

```

In[1062]:= GraphicsGrid[
  ArrayReshape[#, {4, 3}, ""] &@Map[ListPlot[Normal@Diagonal@svdRes[#] [[2]],
    PlotRange → MinMax[Flatten@Normal@Values[svdRes] [[All, 2]]],
    PlotStyle → PointSize[0.02], PlotTheme -> "Scientific",
    ImageSize → 190, PlotRange → All, PlotLabel → #] &, Keys[svdRes]]]

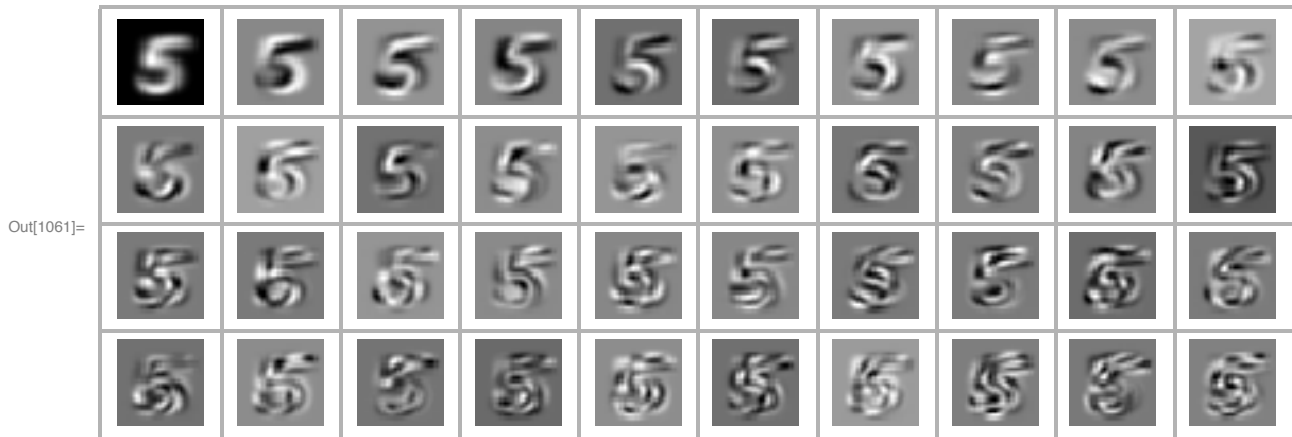
```

Out[1062]=



Here is how the basis for 5 looks like:

```
In[1061]:= Magnify[#, 1.6] &@Grid[Partition[#, 10] &@
  Map[ImageAdjust@Image@Partition[#, 28] &, Transpose@svdRes[5][[3]],
  Dividers → All, FrameStyle → GrayLevel[0.7]]
```



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.

## Classification function

```
Clear[SVDClassifyImageVector]
SVDClassifyImageVector[factorizationRes_Association, vec_?VectorQ] :=
  Block[{residuals, V, rv},
    residuals =
      Map[ (
        V = factorizationRes[#][[3]];
        rv = vec - V.(Transpose[V].vec);
        Norm[rv]
      ) &, Keys[factorizationRes]];
    {Keys[factorizationRes][[Position[residuals, Min[residuals]][[1, 1]]],
      AssociationThread[Keys[factorizationRes] -> residuals]}
  ];
```

Here is an example classification with this function :

```
In[1020]:= ind = 123;
Grid[{{"Image", "Label"},
  {Image[testImages[[ind]]], testImagesLabels[[ind]]}}, Dividers → All]
```





```
In[1022]:= SVDClassifyImageVector[svdRes, testImagesMat[[ind, All]]]
Out[1022]= {7, <| 0 → 1214.89, 1 → 991.68, 2 → 1087.78, 3 → 955.392, 4 → 958.655,
           5 → 1040.59, 6 → 1305.8, 7 → 544.925, 8 → 1041.6, 9 → 779.461 |> }
```

## Classification evaluation

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

```
AbsoluteTiming[
  svdClRes = Map[SVDClassifyImageVector[svdRes, #] &, # & /@ testImagesMat];
]
{5.90115, Null}

svdClResDT = Transpose[{testImagesLabels, svdClRes[[All, 1]]}];
```

## Total accuracy

```
N@Mean[(Equal@@@ svdClResDT) /. {True → 1, False → 0}]
0.957
```

## Accuracy per digit

```
t = Map[Association@Flatten@{"Label" -> #[[1, 1]], "NImages" -> Length[#],
  "Accuracy" -> N@Mean[(Equal@@@ #) /. {True → 1, False → 0}]} &,
  GatherBy[svdClResDT, First]];
t = SortBy[t, First];
Dataset[t]
```

| Label | NImages | Accuracy |
|-------|---------|----------|
| 0     | 980     | 0.992857 |
| 1     | 1135    | 0.989427 |
| 2     | 1032    | 0.943798 |
| 3     | 1010    | 0.932673 |
| 4     | 982     | 0.972505 |
| 5     | 892     | 0.931614 |
| 6     | 958     | 0.966597 |
| 7     | 1028    | 0.937743 |
| 8     | 974     | 0.950719 |
| 9     | 1009    | 0.947473 |

## Confusion matrix (accurate vs predicted)

```
svdCFMat = Normal[SparseArray[Map[(#[[1]] + {1, 1}) → #[[2]] &, Tally[svdClResDT]]]];
MatrixForm[svdCFMat, TableHeadings → {Range[0, 9], Range[0, 9]}]
```

|   | 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

## The matrix factorization

Mathematica does not have NMF implementation (yet.) Here we are using the implementation from MathematicaForPrediction at GitHub:

<https://github.com/antononcube/MathematicaForPrediction/blob/master/NonNegativeMatrixFactorization.m>.

```
Import[
  "https://raw.githubusercontent.com/antononcube/MathematicaForPrediction/master/
    NonNegativeMatrixFactorization.m"]
```

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.)

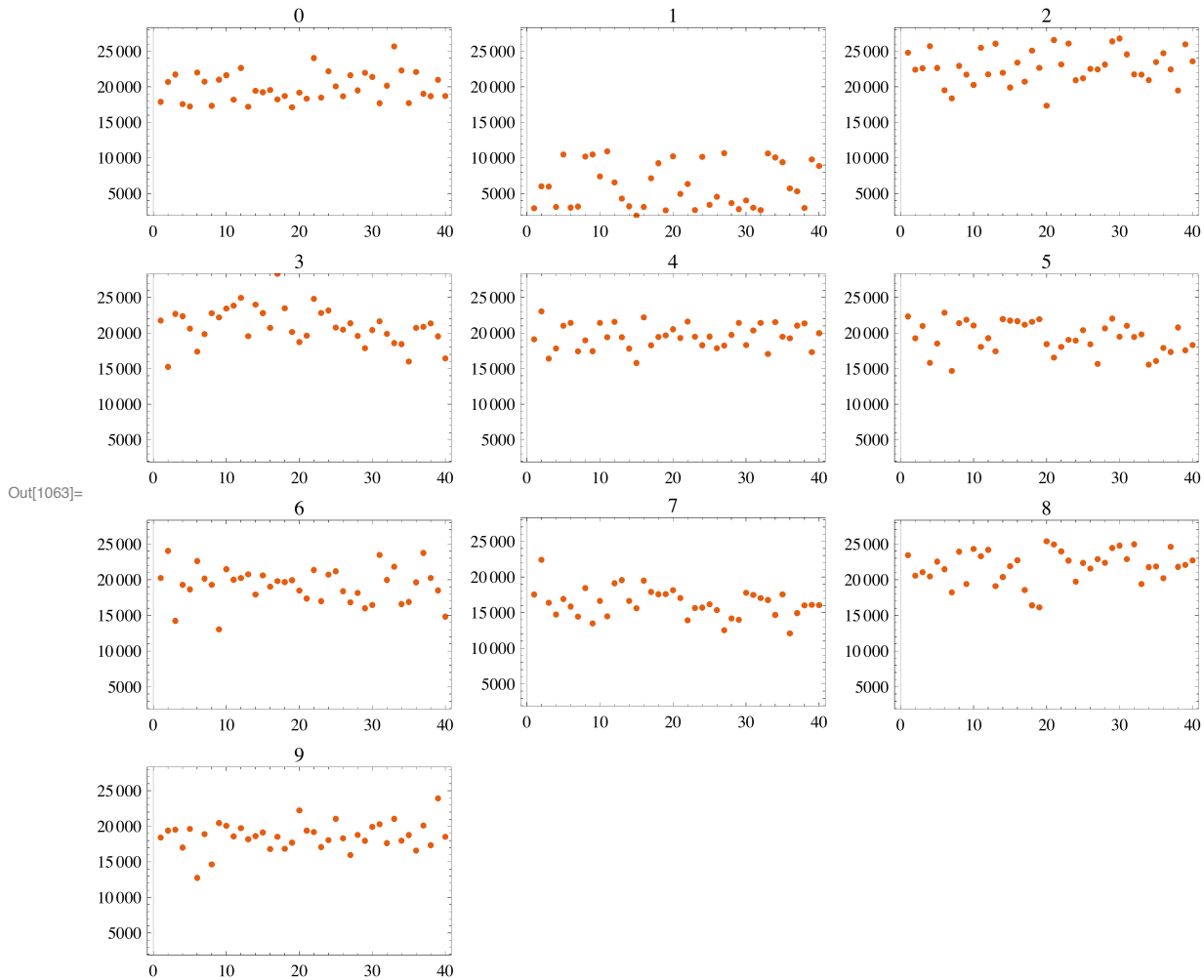
```
In[1042]:= AbsoluteTiming[
  nnmfRes =
    ParallelMap[Function[{ cl },
      Print[Style[cl, Bold, Red]];
      pos = Flatten@Position[trainImagesLabels, cl];
      tmat = trainImagesMat[[pos, All]];
      res = GDCLS[tmat, 40, PrecisionGoal → 4, "MaxSteps" → 20,
        "RegularizationParameter" → 0.1, "PrintProfilingInfo" → False];
      bres = RightNormalizeMatrixProduct@@res;
      Join[bres, { (Norm /@ res[[2]]) / (Norm /@ bres[[2]]), PseudoInverse[bres[[2]], tmat}]]
    ], classLabels, DistributedContexts → Automatic];
  nnmfRes = AssociationThread[classLabels → nnmfRes];
]
```

```
Out[1042]:= {122.247, Null}
```

## Basis interpretation

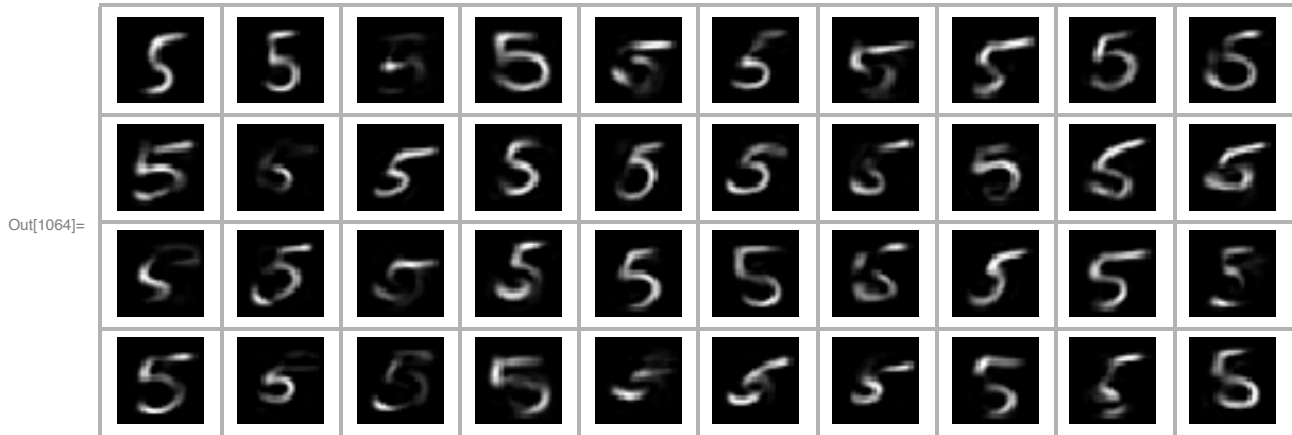
The following multi-panel plot shows the significance of the obtained NMF basis vectors for each digit.

```
In[1063]:= Grid[ArrayReshape[#, {4, 3}, ""] &@Map[ListPlot[nnmfRes[#] [[3]],
  PlotRange → MinMax[Flatten@Normal@Values[nnmfRes] [[All, 3]]],
  PlotStyle → PointSize[0.02], PlotTheme -> "Scientific",
  ImageSize → 190, PlotRange → All, PlotLabel → #] &, Keys[svdRes]]]
```



Here is how the basis for 5 looks like:

```
In[1064]:= Magnify[#, 1.6] &@
Grid[Partition[#, 10] &@Map[ImageAdjust@Image@Partition[#, 28] &, nnmfRes[5][[2]]],
Dividers → All, FrameStyle → GrayLevel[0.7]]
```



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

## Classification function

Given a matrix  $M \in \mathbb{R}^{n \times m}$  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  $v \in \mathbb{R}^m$  we have to look for the nearest neighbors in the matrix  $W \in \mathbb{R}^{n \times k}$  of the vector  $v H^{-1} \in \mathbb{R}^k$ . The labels of those nearest neighbors are used to predict the label of  $v$ .

```

In[1025]:= Clear[NNMFClassifyImageVector]
Options[NNMFClassifyImageVector] = {"PositiveDifference" → False,
  "NumberOfNNs" → 4, "WeightedNNsAverage" → False, "RepresentationNorm" → False};
NNMFClassifyImageVector[factorizationRes_Association,
  vec_?VectorQ, opts : OptionsPattern[]] :=
Block[{residuals, invH, nW, nf, approxVec, scores, pos,
  rv, nnns = OptionValue["NumberOfNNs"], inds, ws},
  residuals =
  Map[(
    invH = factorizationRes[#][[4]];
    (*nW = (#/Norm[#])&/@factorizationRes[#][[1]];*)
    nf = Nearest[
      factorizationRes[#][[1]] → Range[Dimensions[factorizationRes[#][[1]][[1]]];
      If[TrueQ[OptionValue["RepresentationNorm"]],
        approxVec = vec.invH;
        CosineDistance[
          Flatten[factorizationRes[#][[1]][nf[approxVec]], approxVec],
          (*ELSE*)
          inds = nf[vec.invH, OptionValue["NumberOfNNs"]];
          If[TrueQ[OptionValue["WeightedNNsAverage"]],
            ws = Map[Norm[vec - #] &, (factorizationRes[#][[5])[inds]];
            approxVec = ws.(factorizationRes[#][[5])[inds],
            (*ELSE*)
            approxVec = Total[(factorizationRes[#][[5])[inds]]
          ];
          rv = vec / Norm[vec] - approxVec / Norm[approxVec];
          If[TrueQ[OptionValue["PositiveDifference"]], rv = Clip[rv, {0, ∞}]];
          Norm[rv]
        ]
      ) &, Keys[factorizationRes]];
  {Keys[factorizationRes][Position[residuals, Min[residuals]][[1, 1]],
    AssociationThread[Keys[factorizationRes] -> residuals]}
];

```


Here is an example classification with this function :

```

In[1047]:= ind = 123;
Grid[{{"Image", "Label"},
  {Image[testImages[[ind]], testImagesLabels[[ind]]}, Dividers → All]

```

Out[1048]=

| Image   | Label |
|---|-------|
|  | 7     |

```
In[1049]:= NNMFClassifyImageVector[nnmfRes,
      testImagesMat[[ind, All]], "RepresentationNorm" → True]
Out[1049]= {7, <| 0 → 0.634219, 1 → 0.329333, 2 → 0.474811, 3 → 0.500061, 4 → 0.538733,
      5 → 0.561278, 6 → 0.563989, 7 → 0.324574, 8 → 0.555481, 9 → 0.503617 |> }
```

## Classification

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

```
In[1050]:= AbsoluteTiming[
      nnmfClResInv =
        ParallelMap[NNMFClassifyImageVector[nnmfRes, #, "RepresentationNorm" → False,
          "NumberOfNNs" → 30, "WeightedNNsAverage" → False] &, # & /@ testImagesMat];
      ]
Out[1050]= {74.8331, Null}

In[1051]:= nnmfClResDT = Transpose[{testImagesLabels, nnmfClResInv[[All, 1]]}];
```

## Total accuracy

```
In[1052]:= N@Mean[(Equal@@@nnmfClResDT) /. {True → 1, False → 0}]
Out[1052]= 0.9663
```

## Accuracy per digit

```
In[1053]:= t = Map[Association@Flatten@{"Label" -> #[[1, 1]], "NImages" -> Length[#],
    "Accuracy" -> N@Mean[(Equal@@@#) /. {True -> 1, False -> 0}]] &,
    GatherBy[nnmfClResDT, First]];
t = SortBy[t, First];
Dataset[t]
```

Out[1055]=

| Label | NImages | Accuracy |
|-------|---------|----------|
| 0     | 980     | 0.986735 |
| 1     | 1135    | 0.99207  |
| 2     | 1032    | 0.969961 |
| 3     | 1010    | 0.943564 |
| 4     | 982     | 0.965377 |
| 5     | 892     | 0.956278 |
| 6     | 958     | 0.988518 |
| 7     | 1028    | 0.949416 |
| 8     | 974     | 0.973306 |
| 9     | 1009    | 0.93558  |

## Confusion matrix (accurate vs predicted)

```
In[1056]:= nnmfCFMat = Normal[SparseArray[Map[(#[[1]] + {1, 1}) -> #[[2]] &, Tally[nnmfClResDT]]]];
MatrixForm[nnmfCFMat, TableHeadings -> {Range[0, 9], Range[0, 9]}]
```

Out[1057]//MatrixForm=

|   | 0   | 1    | 2    | 3   | 4   | 5   | 6   | 7   | 8   | 9   |
|---|-----|------|------|-----|-----|-----|-----|-----|-----|-----|
| 0 | 967 | 0    | 2    | 1   | 0   | 2   | 7   | 0   | 1   | 0   |
| 1 | 0   | 1126 | 4    | 2   | 0   | 0   | 2   | 0   | 0   | 1   |
| 2 | 5   | 2    | 1001 | 2   | 1   | 0   | 1   | 5   | 15  | 0   |
| 3 | 0   | 0    | 8    | 953 | 1   | 17  | 0   | 5   | 21  | 5   |
| 4 | 1   | 1    | 3    | 0   | 948 | 0   | 3   | 4   | 2   | 20  |
| 5 | 2   | 0    | 1    | 8   | 1   | 853 | 4   | 3   | 13  | 7   |
| 6 | 4   | 2    | 0    | 0   | 1   | 2   | 947 | 0   | 2   | 0   |
| 7 | 1   | 8    | 11   | 2   | 5   | 0   | 0   | 976 | 2   | 23  |
| 8 | 3   | 0    | 4    | 4   | 0   | 8   | 0   | 3   | 948 | 4   |
| 9 | 2   | 3    | 3    | 5   | 21  | 6   | 1   | 19  | 5   | 944 |



## Comparison with the built-in classifiers

The result obtained are fairly competitive with *Mathematica*'s built in classifiers. Using SVD we get less but close accuracy with much faster computations. Below is given an example using `Classify`'s neural network classifier.

### Build classifier over image vectors

```
AbsoluteTiming[
  netCFV =
    Classify[(# & /@ trainImagesMat) → trainImagesLabels, Method -> "NeuralNetwork"]
]
```

```
{665.292, ClassifierFunction[ Method: NeuralNetwork  
Number of classes: 10}]}
```

### Measure accuracies

```
In[1035]:= cm = ClassifierMeasurements[netCFV, Thread[(# & /@ testImagesMat) → testImagesLabels]]
```

```
Out[1035]= ClassifierMeasurementsObject[ Classifier: NeuralNetwork  
Number of test examples: 10000]
```

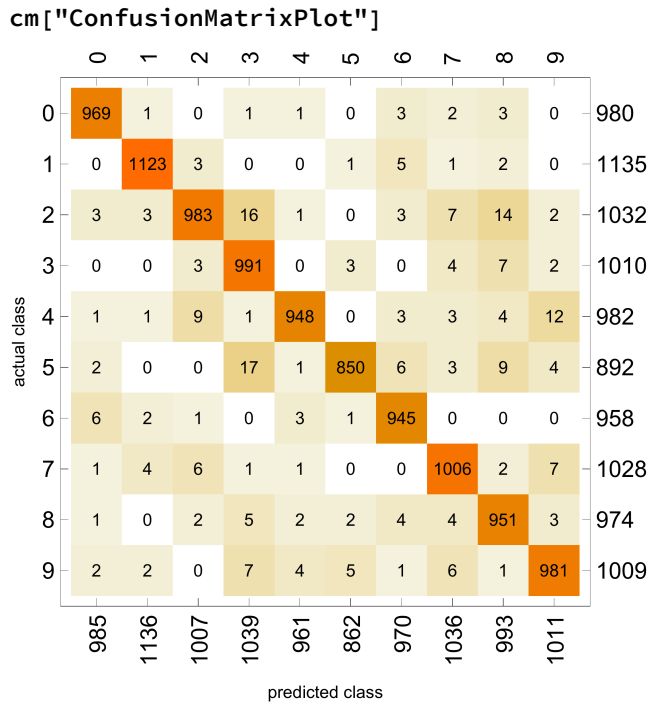
```
cm["Accuracy"]
```

```
0.9747
```

```
In[1040]:= Dataset[cm["FScore"]]
```

```
Out[1040]=
```

|   |          |
|---|----------|
| 0 | 0.98626  |
| 1 | 0.988992 |
| 2 | 0.964198 |
| 3 | 0.967301 |
| 4 | 0.975811 |
| 5 | 0.969213 |
| 6 | 0.98029  |
| 7 | 0.974806 |
| 8 | 0.966955 |
| 9 | 0.971287 |



## References

- [1] Yann LeCun et al., MNIST database site, <http://yann.lecun.com/exdb/mnist/>.
- [2] Anton Antonov, "Classification of handwritten digits", (2013), blog post at MathematicaForPrediction at WordPress (<https://mathematicaforprediction.wordpress.com/>).
- [3] Lars Elden, Matrix Methods in Data Mining and Pattern Recognition, 2007, SIAM.