Using GNU Octave for Handwritten Digit Recognition

Michael J. M. Mazack
Department of Scientific Computation
University of Minnesota - Twin Cities

April 22, 2010

Introduction

GNU Octave[1] is a free high-level language for numerical computation largely compatible with MathWorks Inc.'s MATLAB. In this paper, we discuss how to use GNU Octave for handwritten digit recognition using a database of training and test digits from the United States Postal Service[2]. We test a modified version of the SVD classification algorithm described by Savas in [1] as well as the NMF classification algorithm described by Mazack in [2]. It is assumed that the reader is already familiar with the basics of the MATLAB language. Public domain source code for the two algorithms is also provided in the appendix.

Handwritten Digit Recognition

Background

The postal services of various countries rely on algorithms for the automatic classification and sorting of letters by machine. Several such algorithms have been developed, most notably those discussed by Savas [1]. The computational aspects of the sorting problem can reduced to the automatic classification of a single, unknown, handwritten "test digit" using a database of known "training digits". The databases we consider consist of 7291 grayscale images of training digits and 2007 grayscale images of test digits. For more details on the databases, see [3].

Loading the Databases

Correct loading of the databases is paramount to obtaining correct classification results. In solving this problem, we first note that the two databases `zip.train` and `zip.test` are ASCII text files of the same format. The lines stored in the databases consist of 257 floating point numbers terminated by a line break. The first entry of a line is an integer-valued floating point number correctly identifying the digit, while the remaining 256 entries are the column-wise entries of a 16×16 matrix corresponding to the 2D monochrome image of the digit. The digits in the database are unsorted. Thus, it is necessary to use some sort of indexing scheme to properly identify where they should be placed.

Files can be opened in GNU Octave and MATLAB by using the `fid = fopen(filename)` command. This command takes the path of the file[3] as its argument and returns a value identifying the opened file. Once a file has been opened, data can be read using a variety of commands. The most notable (and perhaps most useful) command is `tline = fgetl(fid)`. This command will read the next line of the file identified by `fid` into the string variable `tline`. Success of the read can be checked with the `ischar(tline)` command, which will return false if the end of the file has been reached or true otherwise. After the ASCII string data has been read from the file `fid` and stored into the variable `tline`, it can be converted into a vector of floating point numbers using the command `place_holder = sscanf(tline, '%f')`.

[1] GNU Octave on the web: `http://www.octave.org`
[2] Digit databases available at: `http://www-stat.stanford.edu/~tibs/ElemStatLearn/data.html`
[3] The path is given as a string in single quotes. For example, `fid = fopen('data/zip.train');`

At this point the correct digit identifier is stored in `place_holder(1)`, and a vector in $\mathbb{R}^{256}$ containing the stacked columns is stored in `place_holder(2:257)` – which can be normalized to be valued between 0 (white) and 1 (black). These vectors are placed column-wise into a three dimensional array where the third dimension denotes the correct digit identifier[4]. For the exact structure and implementation of loading the database files, please see `make_digit_mats.m` in the appendix.

Displaying Digits

Displaying images in GNU Octave is somewhat different than what is done in MATLAB. GNU Octave first requires a color map to display images. A default color map is provided (`colormap('default')`) along with a gray color map which can be set for our 0 (white) and 1 (black) scheme by using the command `colormap(1 .- gray)`. Colors of the images are determined by the closest index in the colormap to the entries in the image. Thus, it is necessary to rescale the values in the image between 0 and the length of the columns of the colormap (usually 256). This will ensure that the full range of grays will be acheived by the image. After choosing a proper colormap, images can be rendered with the `image(A)` command, where `A` is the image matrix to be displayed. In our case of a vector in $\mathbb{R}^{256}$ it is necessary to decompose the stacked columns into a 16×16 matrix. This result can be acheived using the built-in `reshape(b, 16, 16)` command for a vector `b` of length 256. Below are several digits from the database after being renormalized. For further details, please see the source code `show_digit.m` in the appendix.

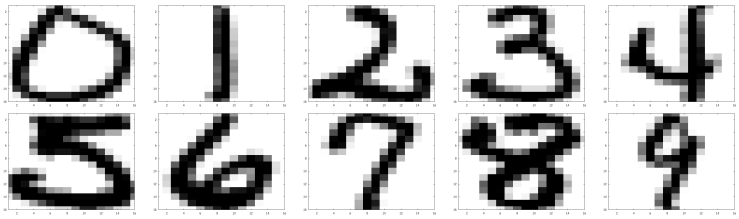

Figure 1: Several Images from the Database. Similar images can be generated using the file `show_digit.m`.

Algorithm Overview

We test a modified version of the the SVD-based algorithm in [1] and the NMF-based algorithm in [2]. We begin by constructing the digit matrices D_i for every $i \in \{0, 1, ..., 9\}$ whose columns consist of all digits of type i from the training digit database. These matrices are built with the file `make_digit_mats.m`.

$$D_5 = \begin{bmatrix} | & | & | & \dots & | \\ 5 & 5 & 5 & \dots & \mathbf{5} \\ | & | & | & \dots & | \end{bmatrix} \quad D_5 \in \mathbb{R}^{256 \times 556}$$

Figure 2: Unrolled 16×16 matrices are stored as vectors in $\mathbb{R}^{256}$ which are columns of D_5.

Once every D_i matrix has been formed, we open the test digit database and sequentially read the digits into a vector $d \in \mathbb{R}^{256}$ and consider the least squares problem

$$\rho_i = \min_x \|D_i x - d\|_2^2.$$

Our goal is to classify d by finding ρ_i for every $i \in \{0, 1, ..., 9\}$ and classifying d as the i given by $\min_i\{\rho_i\}$. In principle, this can only be done for low-rank approximations of D_i due to many of the D_i matrices spanning $\mathbb{R}^{256}$. As shown by Mazack in [3] and [2], the structure of the SVD and NMF can be used to exploit effecient solving of the least squares problem by using only the left-most matrix from the factorizations. We summarize the algorithm below.

[4]For the digit 0, an array index value of 10 is used to prevent underflow.

Let $i \in \{0, 1, ..., 9\}$.

Do once at startup:

- Form the D_i matrices for every i.
- Compute a rank-k approximation for the column space M_i for each D_i.

Let $d \in \mathbb{R}^{256}$ be a test digit to classify.

- For every i, compute $q_i = \min_y \|M_{i_k} y - d\|_2^2$.
- Compute $\min_i\{q_i\}$ and classify d as an "i".

An implementation of the algorithm is given in `run_algorithm.m`. Since GNU Octave does not natively support NMF as of version 3.2.4, a simple public-domain implementation of the multiplicative update algorithm for NMF from [2] has been provided in the file `nmf_mu.m`.

Classification Results

Below are a few statistical results obtained by running both the SVD-based and NMF-based classification algorithm with a rank-10 approximation. The rank can be adjusted by changing the value of `rnk` in the file `run_algorithm.m`.

Accuracy Results for SVD-Based Algorithm

Digit	Sample Size	Correct	Incorrect	Success Rate
0	359	354	5	98.607%
1	264	260	4	98.485%
2	198	175	23	88.384%
3	166	142	24	85.542%
4	200	183	17	91.500%
5	160	142	18	88.750%
6	170	164	6	96.471%
7	147	138	9	93.878%
8	166	150	16	90.361%
9	177	170	7	96.045%

Average Success Rate for SVD: 93.572%.

Accuracy Results for NMF-Based Algorithm

Digit	Sample Size	Correct	Incorrect	Success Rate
0	359	352	7	98.050%
1	264	260	4	98.485%
2	198	179	19	90.404%
3	166	143	23	86.145%
4	200	174	26	87.000%
5	160	139	21	86.875%
6	170	162	8	95.294%
7	147	134	13	91.156%
8	166	146	20	87.952%
9	177	167	10	94.350%

Average Success Rate for NMF: 92.476%.

Concluding Remarks

GNU Octave provides a free high-level language capable of allowing for the simple implementation of common low-rank approximation based handwritten digit recognition algorithms. We encourage the use of GNU Octave for such projects. We also release the source code in the appendix into the public domain and humbly request the citation of this paper if the code should be used.

References

[1] B. Savas. "Analyses and Tests of Handwritten Digit Recognition Algorithms." Master's thesis, Mathematics Department, Linköping University, 2003.

[2] M. Mazack. "Non-negative Matrix Factorization with Applications to Handwritten Digit Recognition." Department of Scientific Computation, University of Minnesota, 2009.

[3] M. Mazack. "Algorithms for Handwritten Digit Recognition." Master's colloquium, Mathematics Department, Western Washington University, 2009.

Appendix

The code implementing the SVD-based algorithm described in [3] and the NMF-based algorithm in [2] is given below. This code is hereby released into the public domain with permission for limitless redistribution and modification. All that is asked for in return is a citation of this paper.

Execution Details and Instructions:

- Place the database files `zip.train` and `zip.test` in the `./data/` directory.
- To run the SVD-based algorithm: `[stats, success_rate] = run_algorithm('svd');`
- To run the NMF-based algorithm: `[stats, success_rate] = run_algorithm('nmf');`
- General statistical data is stored in `stats` according to the tables in the body of the paper.
- The average success rate is stored in `success_rate`.
- The rank can be adjusted by changing the value of `rnk` in `run_algorithm.m`.

NOTE: The code below is not completely copy-and-paste safe. The ' character must be changed to the ASCII value of the apostrophe. To fix this, simply run a find/replace replacing ' with the keyboard value of apostrophe after copy-and-paste.

run_algorithm.m

```
function [stats, success_rate] = run_algorithm(algo_type, digit_mats)

rnk = 10;

if ~exist('digit_mats')
  digit_mats = make_digit_mats('data/zip.train');
end

if strcmp(algo_type, 'svd')
  for i = 1:10
    [u, s, v] = svd(digit_mats(:, :, i));
    M(:, :, i) = u(:, 1:rnk);
  end
```

```
elseif strcmp(algo_type, 'nmf')
  for i = 1:10
    [w, h] = nmf_mu(digit_mats(:, :, i), rnk, 100);
    M(:, :, i) = w;
  end
end

fid = fopen('data/zip.test');

stats = zeros(10, 5); stats(:, 1) = 1:10; stats(10, 1) = 0;

while 1

  tline = fgetl(fid);
  if ~ischar(tline)
    break;
  end

 td = sscanf(tline, '%f');

 digit_num = td(1, 1);
 if digit_num == 0
    digit_num = 10;
 end

 test_dig = td(2:257);
 test_dig = 0.5.*(test_dig .+ 1);

 stats(digit_num, 2) = stats(digit_num, 2) + 1;

 resids = zeros(10, 1);

 if strcmp(algo_type, 'svd')
   for i = 1:10
     x = M(:, :, i)'*test_dig;
     resids(i) = norm(M(:, :, i)*x - test_dig);
   end
 elseif strcmp(algo_type, 'nmf')
   for i = 1:10
     x = M(:, :, i)\test_dig;
     resids(i) = norm(M(:, :, i)*x - test_dig);
   end
 end

 [smallest_resid, digit_result] = min(resids);

 if digit_num == digit_result
   stats(digit_num, 3) = stats(digit_num, 3) + 1;
 else
   stats(digit_num, 4) = stats(digit_num, 4) + 1;
 end

end

fclose(fid);

stats(:, 5) = stats(:, 3)./stats(:, 2);
success_rate = sum(stats(:, 3))/sum(stats(:, 2));
```

make_digit_mats.m

```
function [digit_mats] = make_digit_mats(filename)

fid = fopen(filename);
counter = zeros(10);

while 1

  tline = fgetl(fid);
  if ~ischar(tline)
    break;
  end

 place_holder = sscanf(tline, '%f');
 dig_num = place_holder(1, 1);

 if dig_num == 0
   dig_num = 10;
 end

 counter(dig_num) = counter(dig_num) + 1;
 digit_mats(:, counter(dig_num), dig_num) = place_holder(2:257);
end

fclose(fid);

digit_mats = 0.5.*(digit_mats .+ 1);
```

---

nmf_mu.m

```
function [w, h] = nmf_mu(a, k, maxiter)

[m, n] = size(a);

w = rand(m, k);
h = rand(k, n);

for i = 1:maxiter
  h = h.*(w'*a)./(w'*w*h + 1e-9);
  w = w.*(a*h')./(w*h*h' + 1e-9);
end
```

---

show_digit.m

```
function show_digit(img)

img_copy = img .- min(img);
img_copy = (256/max(img_copy)).*img_copy;

colormap(1 .- gray);
image(reshape(img_copy, 16, 16)');
```
