MATH 728D: Machine Learning Lab #1: MATLAB

John Burkardt

February 11, 2019

How do we have a mathematical conversation with MATLAB?

This lab jumps right into MATLAB. We want to go over the most common techniques and commands that will be used in future computational lab work. Because machine Learning involves data, we will look at reading data from a file into an array, and performing some simple tasks on that data, and writing data back to a file. We will also look a little bit at MATLAB arrays, and control statements.
If you are completely new to MATLAB, then you should review the information at places like the MATHWORKS tutorial website: **https://www.mathworks.com/support/learn-with-matlab-tutorials.html**
The first labs will present you with examples of various MATLAB techniques. In many cases, there will be a lot of unexplained material. It is your responsibility to figure things out, to work out the details, and be able to understand what is going on, so that you can use these techniques on your own to solve the upcoming problems.

1 Unscramble a matrix

Locate the file *scrambled.txt*. Copy it into your MATLAB working directory. Start MATLAB. You should see the name of the file showing up within your MATLAB window. MATLAB can display the contents by the command

```
type 'scrambled.txt'
```

Load the data into a MATLAB array called A:

```
A = load ( 'scrambled.txt' );
```

Notice in your MATLAB Workspace window that the A variable shows up with dimensions 7x5, that is, an array or matrix with 7 rows and 5 columns. We can also check those values with the *size()* command:

```
[ m, n ] = size ( A );
```

Notice that we now have created two more variables, m and n!
To access a specific value in an array, we write something like *A(i,j)* where i and j specify the row and column. The row and column indices can be numbers, or ranges, or a colon:

```
A(3,2)       <-- a single value in row 3, column 2
A(5,2:4)     <-- row 5, columns 2 through 4
A(:,4)       <-- column 4, all entries
B(2,:)       <-- row 2, all entries
B(3:6,1)     <-- rows 3 through 6, column 1
```

Commands that don't terminate with a semicolon will print out the corresponding values computed, if any.
The A array has some mistakes in it, which we want to correct. The correct values are quite simple:

```
11  12  13  14  15
21  22  23  24  25
31  32  33  34  35
41  42  43  44  45
51  52  53  54  55
61  62  63  64  65
71  72  73  74  75
```

As a guide, we will load a copy of the correct data:

```
B = load ( 'correct.txt' );
```

Our goal is to modify the array A so that it matches **B**. To see whether we have a match, we could type:

```
A - B
```

but we can get a condensed answer by asking for the norm of the difference, which will only be zero if the matrices match:

```
norm ( A - B )
```

Let's try to patch the matrix A so that it equals B.

Exercise 1:

1. Print the norm of the difference of A and B;
2. Change the entry in row 6, column 4 of A to 64;
3. Replace all negative values of A by positive values; you can do this in a single command;
4. Round all noninteger values of A **down** (a single command)
5. Swap rows 2 and 4 of A;
6. Swap columns 3 and 5 of A
7. Now print the norm of the difference of A and B; Is it zero?

By default, variables created during a MATLAB session remain in memory until you exit. You can see them listed in your **Workspace**. You can erase unneeded variables by name:

```
clear A
```

or for a completely fresh start, type:

```
clear
```

2 Clean up the GEYSER Data

Locate the file *geyser_data.csv*. This data file uses the very popular format CSV (Comma Separated Value). The CSV format can include text as well as numeric data; Our file starts with a line of text that describes the data. The following lines are entirely numeric. Unfortunately, text data, such as header lines, personal names, or days of the week, will confuse the *load* command, but we can use MATLAB's *csvread()* function, as long as we tell it to skip that first text line. We will also skip the first column, which contains a boring index value. The command to copy the "interesting" data from a file into a MATLAB array is:

```
F = csvread ( 'geyser_data.csv', 1, 1 );
```

Notice in your MATLAB Workspace window that the F variable shows up with dimensions 273x2, that is, an array or matrix with 273 rows and 2 columns.
The first column of F records the length of an eruption of the Old Faithful geyser; the second records the waiting time until the next eruption. These two pieces of data might be completely unrelated. On the other

hand, it makes sense to think that a very long eruption might be followed by a longer than usual weight, and vice versa. Instead of looking at the list of numbers, let's try to make a plot. A scatter plot of unconnected dots is most appropriate here:

```
plot ( F(:,1), F(:,2), 'o' )
```

We do see a pattern, but we also see an odd thing. Most of the data is crammed into the upper left corner, and one dot appears in the lower right. This is called an **outlier**; it might represent some actual interesting physical phenomenon, but it could also just be a typing mistake! Let's assume it's a mistake, clean up our data, and try again.
First, let's diagnose our data. The commands

```
max(F), min(F), mean(F), std(F)
```

produce information about each column of F. Compute these values. Think of the mean as a typical value and the standard deviation as a typical reasonably small variation. Which maximum value is far from its mean? Which minimum?
To fix the data, we actually need to know which rows of the data contain bad values. While we could try to search it with an editor, let's instead try some automatic commands. First, let's ask for the location of any data in columns 1 and 2 that is more than about 2 standard deviations from the mean:

```
bad1 = find ( abs ( F(:,1) - 4 ) > 2 )
bad2 = find ( abs ( F(:,2) - 70 ) > 28 )
```

It's more useful to ask the reverse question, where is the good data:

```
good = find ( abs ( F(:,1) - 4 ) <= 2  & ...
                    abs ( F(:,2) - 70 ) <= 28 )
```

The variable *good* lists all the rows in which the entries in columns 1 and 2 seem reasonable. If we replace the colon symbol (all rows) by "good" (just the good rows), then we can try our plot again:

```
plot ( F(good,1), F(good,2), 'o' )
```

Now that the one bad data is gone, the relationship between the two variables should be much easier to see. To summarize:

Exercise 2:

1. Use *csvread()* to read the data *geyser_data.csv* into the array **F**;
2. plot F(:,1) versus F(:,2);
3. Compute minimum, maximum, mean and standard deviation of **F**;
4. Identify bad rows in column 1 and in column 2;
5. Identify good rows;
6. Repeat plot, using only the good data;

3 Compute, save and plot the HAILSTONE Data

Consider the following procedure:

1. Choose a positive integer n.
2. Set $m = n$, and set *count* to 0;
3. If m is 1, print n and *count*, and terminate;
4. If m is even, replace it by $m/2$; if odd, replace it by $3m + 1$;
5. Increase *count* by 1.
6. Go to step 3.

Here we have an example of an algorithm, which produces the value *count* for any positive integer n. We want to express this algorithm with MATLAB. Suppose, then, that the value of n has been chosen. Then the following commands would carry out the task:

```
m = n;
count = 0;
while ( m ~= 1 )      % Watch out. "Not equal" uses the "twiddle" character.
  if ( mod ( m, 2 ) == 0 )
    m = m / 2;
  else
    m = 3 * m + 1;
  end
  count = count + 1;
end
fprintf ( '  %d. %d\n', n, count );
```

Instead of picking a single value of n, suppose we want to make a table of the value of $count(n)$ for $1 \leq n \leq 150$? In that case, we can use the MATLAB *for* and *end* statements to make a "sandwich" around our previous algorithm:

```
for n = 1 : 150

  --- insert algorithm to compute count from n ---

  fprintf ( '  %d. %d\n', n, count );
end
```

Finally, suppose we wanted to create a hailstone data file, containing the 150 pairs of values we have computed? Then we have to:

1. Use *fopen* to open a file with a name we choose;
2. Use a modified *fprintf()* statement to write data to the file;
3. Use *fclose* to close the file when we are done.

```
file = fopen ( 'hailstone_data.txt', 'wt' );  % 'wt' means 'write text'

for n = 1 : 150

   --- insert algorithm to compute count from n ---

   fprintf ( file, '  %d. %d\n', n, count );
end

fclose ( file );
```

Exercise 3:

1. Write a MATLAB program that computes the first 150 hailstone counts;
2. Execute your program, creating the data file *hailstone_data.txt*;
3. Read the data back in by `h = load('hailstone_data.txt');`
4. plot h(:,1) versus h(:,2) as a dot plot;

4 Compute and plot the MOUNTAIN data

Suppose we are interested estimating the values (x, y) which maximize the function $z = f(x, y)$. We suppose we are given the formula $f(x, y)$, along with the requirement that $-3 \leq x, y \leq +3$. The formula here is particularly awful:

$$f(x,y) = 3(1-x)^2 e^{-x^2-(y+1)^2} - (2x - 10x^3 - 10y^5)e^{-x^2-y^2} - 1/3e^{-(x+1)^2-y^2}$$

MATLAB allows us to hide the details of such a formula inside a function, which should actually be stored as a separate file *mountain.m*:

```
function z = mountain ( x, y )

  z =  3*(1-x).^2.*exp(-(x.^2) - (y+1).^2) ...
    - 10*(x/5 - x.^3 - y.^5).*exp(-x.^2-y.^2) ...
    - 1/3*exp(-(x+1).^2 - y.^2);

  return
end
```

Notice that three dots mean *continued on next line* and that the "dotted" operator `.^` is a MATLAB convention that basically says *do normal squaring, not a linear algebra vector operation.* We will see what is going on with dotted operators in the next lab.
The main point is that we have encapsulated our complicated function so that it has a simple interface.
To start our search, we might do well to make a table of the function on a regularly spaced set of points within the domain. If we use a spacing $\Delta x = \Delta y = 1$, then we will need a 7x7 grid of x and y values for our table. We can get this using MATLAB's *meshgrid(xvalues,yvalues)* command:

```
x = linspace ( -3, +3, 7 );
y = linspace ( -3, +3, 7 );
[ X, Y ] = meshgrid ( x, y );
```

Because we were careful to use the "dot" operator within the *mountain* function, we can not only evaluate $f(x, y)$ at a single point, we can evaluate it with one command, at all the points in our table:

```
Z = mountain( X, Y );
```

Because Z is a two dimensional array, the statement *maxz = max (Z)* will actually return in maxz a list of the maximum value in each column of Z. If we want the overall maximum, we have to call *max* twice: *maxz=max(max(Z))*;
The call *[maxz,maxcol]=max(max(Z))*; function will actually supply both the maximum value and the column in which this occurs.

Exercise 4:

1. Write the MATLAB function *mountain.m* (Careful with all those dots!);
2. Use *meshgrid()* to create an X and Y grid;
3. Evaluate *Z = mountain (X, Y)*;
4. Determine the maximum value of Z, and the column in which this occurs;
5. Determine the row in which the maximum value of Z occurs (this can be easy if you use the right MATLAB command!);
6. Use the command *surf(X,Y,Z)* to see a plot of your data;