Matlab Bootcamp

Recap

Numeric Arrays

- e.g. list of phone numbers
- B = [4670456, 7640600, 123456];

Matrices

- e.g. list of GPS coordinates
- A = [42.3086,-83.6921; 82.208, 23.692];

Cell Arrays

- e.g. list of names
- C = {'Duck','Goose','Crane'};

Matrix Manipulation

Adding a scalar to an array

```
A = [1;2;3;4];
A + 3
```

Adding two arrays

```
B = [3;4;5;6];
A + B
```

Element-wise multiplication

```
A.*B
```

Matrix multiplication

```
C = [5 10 15 20];
A*C
```

Organizing variables into structures

Structures

Structure arrays can be used to group related data together. The data is structure arrays is accessed by name.

```
patient(1).name = 'Jane Smith';
patient(1).age = 28;
patient(1).results = [68, 70, 68; 72, 81, 69; 172, 170, 169];
```

```
patient(1)
```

Tables

Tables are used for storing data in rows and column-oriented variables.

- Tables can contain different data types, such as strings and doubles
- Each variable in the table must have the same number of rows

Use *readtable* to import example patient data and then *summary to* examine its contents:

```
T=readtable('patients.dat');
summary(T)
```

Display data for the first four patients:

```
T(1:4,:)
```

Now create a table that only includes the patient age, height and weight:

```
T2=table(T.Age,T.Height,T.Weight);
T2.Properties.VariableNames={'Age','Height','Weight'}
```

Framington Heart Disease Dataset

This dataset comes from a landmark study that analyzed ~14,000 people from three generations. The findings have informed the understanding of factors that impact cardiovascular health.

Import the dataset and determine its size:

```
fram = readtable('frmgham2.xls');
size(fram)
```

The output of *size* indiciates that the dataset includes 11,627 rows and 39 columns. Display the names of the 39 columns:

```
fram.Properties.VariableNames
```

Extracting Simple Properties

Determine the mean, median and range of BMI:

```
mean(fram.BMI)
median(fram.BMI)
range(fram.BMI)
```

The mean and median functions return *NaN* because there is data missing from the BMI array. MATLAB also has functions that ignore these missing data points:

```
nanmean(fram.BMI)
nanmedian(fram.BMI)
```

We will now look at extracting data from the framingham data set based on certain criteria. For example, what if we only want to examine individuals who smoke?

```
smoker_rows = fram.CURSMOKE==1;
fram(smoker_rows,:)
```

Similarly, we can pick out individuals with a BMI above 30:

```
BMI_rows = fram.BMI>35;
fram(BMI_rows,:)
```

This selection criteria method can also be applied to multiple variables at once:

```
SMOKER_BMI_rows = fram.CURSMOKE==1 & fram.BMI>35;
fram(SMOKER_BMI_rows,:)
```

By scrolling through the CURSMOKE and BMI columns of the above table, we can confirm that the new table only includes data for individuals that both smoked and have a BMI above 35. Now create a table which only includes the age, cholesterol and heartrate for these individuals:

```
vars = {'AGE','TOTCHOL','HEARTRTE'};
new_table = fram(SMOKER_BMI_rows,vars)
```

We can also convert this table to a matrix, but we will lose the table headers, as MATLAB arrays can only contain one type of data.

```
new_array = table2array(new_table)
```

Manipulating Datasets

As seen earlier in the BMI array, there are missing values throughout the dataset, which are signified by *NaN.* It is important to know how to both find and replace these missing values.

We can use the *find* and *isnan* functions to determine the rows where data is missing:

```
rows=find(isnan(fram.BMI))
```

Now we want to create a new array for BMI without the missing data points:

```
new_BMI = fram.BMI;
new_BMI(find(isnan(fram.BMI)))=[];
```

To confirm that this worked, we can check that the size of the BMI array has decreased and that the *mean* function now works:

```
mean(new_BMI)
```

Now that the missing data points have been removed, sort the BMI array:

```
sort(new_BMI)
```

We see that the default setting is to sort in ascending order, however there are multiple ways to use *sort.*

```
sort(new_BMI,'descend')
```

Next, let's look at the outliers in the BMI array:

```
[TF,L,U,C] = isoutlier(new_BMI)
```

Similar to the *isnan* function, *isoutlier* creates a new array, *TF,* and identify outlier points with ones. We can also see the lower and upper thresholds and the center value used to determine the outliers (variables L, U, and C).

We can also add new data to our table. Add the *DIABETES* data to table *T4:*

```
new_table.DIABETES = fram.DIABETES(SMOKER_BMI_rows)
```

Simple Analyses of the Dataset

We will now answer a few questions about the Framingam data set.

1.) How many patients over 45 have a BMI over 40?

```
patients = fram(fram.AGE>45 & fram.BMI>40,["AGE","BMI"]);
size(patients)
```

Another method:

```
sum(fram.AGE>45 & fram.BMI>40)
```

2.) How data points are missing for the patient cholesterol? What is the median value for the cholesterol data that we do have?

```
numel(find(isnan(fram.TOTCHOL)))
nanmedian(fram.TOTCHOL)
```

3.) Create a structure with all of the patient IDs and find the number of unique IDs.

```
ID = fram.RANDID
num_ID = numel(ID)
num_unq = numel(unique(ID))
```

Importing and Exporting Datasets

Exporting

As of MATLAB 2019, the write*matrix* function is recommended for exporting datasets as opposed to previously used functions *csvrwrite, xlswrite,* and *dlmwrite*. *writematrix* can be used with many different file types, such as .txt, .dat, .csv, and .xls. However, only numeric data can imported.

Write the array we made earlier, *new_array,* to an Excel file:

```
writematrix(new_array,'new_array.xls')
```

Now do the same thing with *new_table,* except write to a text file:

```
writetable(new_table,'new_table.txt','WriteRowNames',true)
```

Importing

Similar to exporting, the functions *xlsread, dlmread,* and *csvread* are no longer recommended in MATLAB 2019. Instead, you should use *readtable* or *readmatrix.* Import the .xls and .txt files that we just created:

```
readmatrix("new_array.xls")
readtable('new_table.txt')
```

Handling Big Datasets

Head and Tail

The *head* and *tail* functions can be used to view the first and last rows of a table or array. The default for *heads* and *tails* is to display the first and last eight rows, respectively. However, the number of rows can be altered for each function. Consider the framingham dataset we used earlier:

```
first_rows = head(fram, 5)
last_rows = tail(fram,3)
```

Textscan

textscan can be used to read data from a text file or string and insert it into a cell array. Below, we use *textscan* to read the string *chr* and use ' %f ' to indicate that we want to output double-precision floating-point numbers.

```
chr = '0.41 8.24 3.57 6.24 9.27';
C = textscan(chr,'%f');
celldisp(C)
```

Sparces Matrices

Sparse matrices are useful when your dataset is comprised of mostly zeros. As opposed to normal matrices which store every elemnt in the matrix, sparce matrices only store the nonzero elements along with their row indices. Sparce matrices therefor require much less memory for storage than full matrices.

The first step in creating a sparse matrix is determining the density on nonzero elements. The lower the density, the more it makes sense to create a sparce matrix. Consider the variables in the Framingham dataset which detail prevalent diseases:

- Prevalent Angina Pectoris (PREVAP)
- Prevalent Coronary Heart Disease (PREVCHD)
- Prevalent Myocardial Infarction (PREVMI)
- Prevalent Stroke (PREVSTRK)
- Prevalent Hypertensive (PREVHYP)

The section of the data set containing these variables is mostly comprised of zeros. Create an array that includes these five variables, find the density, and find the memory required for the table.

```
prev_table = fram(:,{'PREVAP','PREVCHD','PREVHYP','PREVMI','PREVSTRK'});
prev_array = table2array(prev_table)
nnz(prev_array) / numel(prev_array)
whos prev_array
```

The memory required for the *prev_array* variable is 465 kB. Now, convert *prev_array* into a sparse matrix. You will see that the output is an array of the nonzero elements and their respective indeces, sorted by column. What happens to the requried storage space of the array?

```
S = sparse(prev_array);
whos S
```

The size of the array decreased to 117.4 kB, or about 25% of the original required memory.

If necessary, you can then convert the sparse matrix back into the full matrix with the *full* command:

```
A = full(S)
```

It is also possible to create a sparse matrix directly from the nonzero elements, without needing the full matrix:

S = sparse(i, j, s, m, n);

- i, j = row and column indices, respectively
- s = vector of nonzero values with indices i, j
- m, n = row and column dimensions of resulting matrix, respectively

```
sp = sparse([3 2 3 4 1],[1 2 2 3 4],[1 2 3 4 5],4,4)
```