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
```

```
ans = 4×1
     4
     5
     6
     7
```

Adding two arrays

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

```
ans = 4×1
     4
     6
     8
    10
```

Element-wise multiplication

```
A.*B
```

```
ans = 4×1
     3
     8
    15
    24
```

Matrix multiplication

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

```
A*C
```

```
ans = 4×4
     5    10    15    20
    10    20    30    40
    15    30    45    60
    20    40    60    80
```

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)
```

```
ans = struct with fields:
       name: 'Jane Smith'
        age: 28
    results: [3×3 double]
```

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)
```

```
Variables:

    LastName: 100×1 cell array of character vectors

    Gender: 100×1 cell array of character vectors

    Age: 100×1 double

        Values:

            Min          25
            Median       39
            Max          50

    Location: 100×1 cell array of character vectors

    Height: 100×1 double

        Values:
```

```
        Min          60
        Median       67
        Max          72

Weight: 100×1 double

    Values:

        Min           111
        Median      142.5
        Max           202

Smoker: 100×1 double

    Values:

        Min          0
        Median       0
        Max          1

Systolic: 100×1 double

    Values:

        Min          109
        Median       122
        Max          138

Diastolic: 100×1 double

    Values:

        Min            68
        Median       81.5
        Max            99

SelfAssessedHealthStatus: 100×1 cell array of character vectors
```

Display data for the first four patients:

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

ans = 4×10 table

...

	LastName	Gender	Age	Location	Height	Weight	Smoker	Systolic
1	'Smith'	'Male'	38	'County Ge...	71	176	1	124
2	'Johnson'	'Male'	43	'VA Hospital'	69	163	0	109
3	'Williams'	'Female'	38	'St. Mary'...	64	131	0	125
4	'Jones'	'Female'	40	'VA Hospital'	67	133	0	117

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'}
```

T2 = 100×3 table

	Age	Height	Weight
1	38	71	176
2	43	69	163
3	38	64	131
4	40	67	133
5	49	64	119
6	46	68	142
7	33	64	142
8	40	68	180
9	28	68	183
10	31	66	132

⋮

Framingham 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)
```

```
ans = 1×2
       11627          39
```

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
```

```
ans = 1×39 cell array
    {'RANDID'}    {'SEX'}    {'TOTCHOL'}    {'AGE'}    {'SYSBP'}    {'DIABP'}    {'CURSMOKE'}    {'CIGPDAY
```

Extracting Simple Properties

Determine the mean, median and range of BMI:

```
mean(fram.BMI)
```

```
ans = NaN
```

```
median(fram.BMI)
```

```
ans = NaN
```

```
range(fram.BMI)
```

```
ans = 42.3700
```

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)
```

```
ans = 25.8773
```

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

```
ans = 25.4800
```

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,:)
```

```
ans = 5029×39 table
```

...

	RANDID	SEX	TOTCHOL	AGE	SYSBP	DIABP	CURSMOKE	CIGPDAY
1	9428	1	245	48	127.5000	80.0000	1	20
2	9428	1	283	54	141.0000	89.0000	1	30
3	10552	2	225	61	150.0000	95.0000	1	30
4	10552	2	232	67	183.0000	109.0000	1	20
5	11252	2	285	46	130.0000	84.0000	1	23
6	11252	2	343	51	109.0000	77.0000	1	30
7	11252	2	NaN	58	155.0000	90.0000	1	30
8	12806	2	313	45	100.0000	71.0000	1	20
9	12806	2	NaN	51	109.5000	72.5000	1	30
10	12806	2	320	57	110.0000	46.0000	1	30

⋮

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

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

```
ans = 318×39 table
```

...

	RANDID	SEX	TOTCHOL	AGE	SYSBP	DIABP	CURSMOKE	CIGPDAY
1	12629	2	220	70	149.0000	81.0000	0	0

	RANDID	SEX	TOTCHOL	AGE	SYSBP	DIABP	CURSMOKE	CIGPDAY
2	43522	2	NaN	55	129.0000	76.0000	0	0
3	82188	1	225	37	124.5000	92.5000	0	0
4	82188	1	244	43	156.0000	109.0000	0	0
5	82188	1	226	49	190.0000	123.0000	0	0
6	83398	1	178	52	160.0000	98.0000	0	0
7	83398	1	155	58	173.0000	90.0000	0	0
8	83398	1	NaN	64	205.0000	90.0000	0	0
9	174973	2	206	42	130.0000	80.0000	1	3
10	174973	2	208	48	122.0000	74.0000	1	3

⋮

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,:)
```

```
ans = 84×39 table
```

...

	RANDID	SEX	TOTCHOL	AGE	SYSBP	DIABP	CURSMOKE	CIGPDAY
1	174973	2	206	42	130.0000	80.0000	1	3
2	174973	2	208	48	122.0000	74.0000	1	3
3	202101	2	326	61	200.0000	104.0000	1	1
4	610021	2	180	60	200.0000	122.5000	1	20
5	935116	1	229	44	177.5000	120.0000	1	10
6	968222	1	153	56	182.0000	95.0000	1	1
7	977985	2	268	48	117.5000	80.0000	1	10
8	977985	2	247	60	164.0000	104.0000	1	10
9	1186959	2	NaN	73	200.0000	100.0000	1	2
10	1225217	2	233	55	128.0000	94.0000	1	20

⋮

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)
```

```
new_table = 84×3 table
```

	AGE	TOTCHOL	HEARTRTE
1	42	206	70
2	48	208	75
3	61	326	57
4	60	180	88
5	44	229	104
6	56	153	75
7	48	268	72
8	60	247	78
9	73	NaN	NaN
10	55	233	80

⋮

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)
```

```
new_array = 84×3
    42   206    70
    48   208    75
    61   326    57
    60   180    88
    44   229   104
    56   153    75
    48   268    72
    60   247    78
    73   NaN   NaN
    55   233    80
     ⋮
```

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))
```

```
rows = 52×1
           2
         265
         347
         433
         795
        1216
        1300
```

```
1527
1894
2631
  ⋮
```

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)
```

```
ans = 25.8773
```

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

```
sort(new_BMI)
```

```
ans = 11575×1
   14.4300
   14.5300
   15.1600
   15.3200
   15.3300
   15.5400
   15.6400
   15.9200
   15.9600
   15.9600
     ⋮
```

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

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

```
ans = 11575×1
   56.8000
   56.8000
   56.8000
   55.3100
   52.9400
   51.2800
   48.6400
   47.2200
   46.5200
   45.9800
     ⋮
```

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

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

```
TF = 11575×1 logical array
   0
   0
   0
   0
   0
   0
   0
   0
   0
   0
      ⋮
L = 14.5384
U = 36.4216
C = 25.4800
```

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)
```

new_table = 84×4 table

	AGE	TOTCHOL	HEARTRTE	DIABETES
1	42	206	70	0
2	48	208	75	0
3	61	326	57	0
4	60	180	88	1
5	44	229	104	0
6	56	153	75	1
7	48	268	72	0
8	60	247	78	0
9	73	NaN	NaN	0
10	55	233	80	0

⋮

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)
```

```
ans = 1×2
```

```
    65     2
```

Another method:

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

```
ans = 65
```

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

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

```
ans = 409
```

```
nanmedian(fram.TOTCHOL)
```

```
ans = 238
```

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

```
ID = fram.RANDID
```

```
ID = 11627×1
        2448
        2448
        6238
        6238
        6238
        9428
        9428
       10552
       10552
       11252
         ⋮
```

```
num_ID = numel(ID)
```

```
num_ID = 11627
```

```
num_unq = numel(unique(ID))
```

```
num_unq = 4434
```

Sorting Through Our Data

Intersect

The *intersect* function can find the overlap between lists of numbers or strings. This is helpful for finding data that satisfies multiple criteria.

For example, find the ID numbers of patients over the age of 40 and the ID numbers for patients with glucose levels over 130 mg/dL. Then use *intersect* to find the overlap in these lists.

```
age_IDs = fram(fram.AGE > 40,"RANDID");
glucose_IDs = fram(fram.GLUCOSE > 130, "RANDID");
age_glucose_IDs = intersect(age_IDs,glucose_IDs,'rows')
```

age_glucose_IDs = 227×1 table

	RANDID
1	23727
2	43770
3	83398
4	95541
5	97026
6	162207
7	170881
8	205391
9	210362
10	276073

⋮

Notice that *intersect* sorts the overlapping ID numbers. We can also specify to output the indices from each list where the matching values occur.

```
[age_glucose_IDs, ia, ib] = intersect(age_IDs,glucose_IDs)
```

age_glucose_IDs = 227×1 table

	RANDID
1	23727
2	43770
3	83398
4	95541
5	97026
6	162207
7	170881
8	205391
9	210362
10	276073

⋮

```
ia = 227×1
    32
    52
    91
```

```
     110
     112
     163
     174
     222
     242
     295
      ⋮

ib = 227×1
       1
       2
       3
       5
       7
       8
      11
      12
      13
      14
      ⋮
```

RegExp

The *regexp* function is used to find patterns in a data set. It is often used with string arrays, but can be used with numeric data as well. There are many metacharacters and operators which can be used to specify what pattern you are looking for. For a full list of these options, use the *help* function.

Use *regexp* to find all of the headers in the Framingham dataset that begin with "TIME".

```
position = regexp(fram.Properties.VariableNames,'^(TIME)\w*','match')
```

```
position = 1×39 cell array
    {0×0 cell}    {0×0 cell}    {0×0 cell}    {0×0 cell}    {0×0 cell}    {0×0 cell}    {0×0 cell}    {0×0
```

Next, find the patient ID numbers that begin with the number 1, end with 5, and have 23 in the middle.

First, we must convert the array of ID numbers to a string array:

```
IDs = mat2str(fram.RANDID)
```

```
IDs =
'[2448;2448;6238;6238;6238;9428;9428;10552;10552;11252;11252;11252;11263;11263;11263;12629;12629;12806;128
```

Next, we specify the pattern to search for:

```
pattern_1 = '1\d*23\d+5';
```

How to interpret the above expression:

- `1` must be the first character
- `/d*` means any numeric digits may follow 1
- the ID must contain 23
- `/d+` means at least one number must be between 23 and the final number, 5

```
match_ID = regexp(IDs,pattern_1,'match')
```

```
match_ID =

  0×0 empty cell array
```

The function returns a cell array of IDs which match the pattern we generated. Note that we specified `'match'`. This indicates that we want the actual cell values. The default option instead returns the starting indices of the cell values.

```
match_ID_indices = regexp(IDs,pattern_1)
```

```
match_ID_indices = 1×13
        9603        9611        9619       10595       10603       10611 ···
```

Importing and Exporting Datasets

Exporting

As of MATLAB 2019, the write*matrix* and *writetable* functions are recommended for exporting datasets as opposed to previously used functions *csvrwrite, xlswrite,* and *dlmwrite*. These functions can be used to write several different file types, such as .txt, .dat, .csv, and .xls.

Write an Excel file which contains the data for all male smokers:

```
male_smoker = fram(fram.SEX==1 & fram.CURSMOKE==1,:);
writetable(male_smoker,'male_smoker.xls');
```

Now create an array of the male smoker table and write it to a csv file:

```
male_smoker_array = table2array(male_smoker);
writematrix(male_smoker_array,'male_smoker.csv')
```

Importing

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

```
readmatrix("male_smoker.csv")
```

```
ans = 2594×39
10^6 ×
    0.0094    0.0000    0.0002    0.0000    0.0001    0.0001    0.0000    0.0000 ···
    0.0094    0.0000    0.0003    0.0001    0.0001    0.0001    0.0000    0.0000
    0.0164    0.0000    0.0002    0.0000    0.0002    0.0001    0.0000    0.0000
    0.0204    0.0000    0.0003    0.0000    0.0001    0.0001    0.0000    0.0000
    0.0204    0.0000    0.0003    0.0001    0.0002    0.0001    0.0000    0.0000
    0.0331    0.0000    0.0002    0.0000    0.0001    0.0001    0.0000    0.0000
    0.0331    0.0000    0.0002    0.0001    0.0001    0.0001    0.0000    0.0000
    0.0331    0.0000    0.0002    0.0001    0.0001    0.0001    0.0000    0.0000
    0.0476    0.0000    0.0003    0.0000    0.0001    0.0001    0.0000    0.0000
```

```
0.0476    0.0000    0.0003    0.0001    0.0001    0.0001    0.0000    0.0000
⋮
```

```
readtable('male_smoker.xls')
```

```
ans = 2594×39 table
```

...

	RANDID	SEX	TOTCHOL	AGE	SYSBP	DIABP	CURSMOKE	CIGPDAY
1	9428	1	245	48	127.5000	80	1	20
2	9428	1	283	54	141.0000	89	1	30
3	16365	1	225	43	162.0000	107	1	30
4	20375	1	294	46	142.0000	94	1	15
5	20375	1	288	52	165.0000	92	1	10
6	33077	1	232	48	138.0000	90	1	10
7	33077	1	222	54	139.5000	82	1	6
8	33077	1	215	60	144.5000	80	1	10
9	47561	1	270	44	137.5000	90	1	30
10	47561	1	300	50	134.0000	88	1	35

⋮

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)
```

```
first_rows = 5×39 table
```

...

	RANDID	SEX	TOTCHOL	AGE	SYSBP	DIABP	CURSMOKE	CIGPDAY
1	2448	1	195	39	106	70.0000	0	0
2	2448	1	209	52	121	66.0000	0	0
3	6238	2	250	46	121	81.0000	0	0
4	6238	2	260	52	105	69.5000	0	0
5	6238	2	237	58	108	66.0000	0	0

```
last_rows = tail(fram,3)
```

```
last_rows = 3×39 table
```

...

	RANDID	SEX	TOTCHOL	AGE	SYSBP	DIABP	CURSMOKE	CIGPDAY
1	9999312	2	196	39	133	86	1	30
2	9999312	2	240	46	138	79	1	20
3	9999312	2	NaN	50	147	96	1	10

Sparce Matrices

Sparse matrices are useful when your dataset is comprised of mostly zeros. As opposed to normal matrices which store every element 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 of 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)
```

```
prev_array = 11627×5
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     1     0     0
     0     0     1     0     0
     0     0     0     0     0
     ⋮
```

```
nnz(prev_array) / numel(prev_array)
```

```
ans = 0.1262
```

```
whos prev_array
```

```
  Name                Size              Bytes  Class     Attributes

  prev_array      11627x5              465080  double
```

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
```

```
  Name           Size              Bytes  Class     Attributes

  S          11627x5              117472  double    sparse
```

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)
```

```
A = 11627×5
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     1     0     0
     0     0     1     0     0
     0     0     0     0     0
     ⋮
```

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)
```

```
sp =
   (3,1)        1
   (2,2)        2
   (3,2)        3
   (4,3)        4
   (1,4)        5
```