Matlab Machine Learning Tutorial Notes:

Basics

Creating Matrices:

To create a matrix by hand columns are separated by space(or commas) and rows by semicolons:

```
>>A = [1 2 3 4; 5 6 7 8; 9 10 11 12; 13 14 15 16]

A =

     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

Indexing:

Matlab indexing ***begins at 1***! To index into a matrix:

```
>>A(1,1)
ans =

     1

>>A(1,4)
ans =

     8
```

Submatrices:

Matlab supports efficient and versatile matrix slicing, by specifying vectors of rows and columns to select. The **:** operator specifies all of the rows or all of the columns.

```
>> A(:,:)

ans =

     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

```
>>A(1,:)
ans =

     1     2     3     4


>> A(:,1)

ans =

     1
     5
     9
    13

>> A([4 2 1], :)

ans =

    13    14    15    16
     5     6     7     8
     1     2     3     4
```

Matrix operations

Matrix multiplication:

```
>>A*A

ans =

    90   100   110   120
   202   228   254   280
   314   356   398   440
   426   484   542   600
```

Matrix addition:

```
>> A+A

ans =

     2     4     6     8
    10    12    14    16
    18    20    22    24
    26    28    30    32
```

Solving linear equations:

If your system of equations is overdetermined (has no solution) matlab will provide a least squares solution.

```
>> A\[4; 5; 6; 7]
ans =

   -1.2500
    0.2500
    0.2500
    1.0000
```

Elementwise Operations:

Matlab supports elementwise matrix operations using the **.** Operator:

```

>> A.*A

ans =

     1     4     9    16
    25    36    49    64
    81   100   121   144
   169   196   225   256

>> A./A

ans =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1
```

Matrix Powers:

There is also a power operator which can be used to take matrix powers:

```
>>A^2

ans =

    90   100   110   120
   202   228   254   280
   314   356   398   440
   426   484   542   600

>> A^3

ans =

        3140        3560        3980        4400
        7268        8232        9196       10160
       11396       12904       14412       15920
       15524       17576       19628       21680
```

The power operator can be employed to raise each element of a matrix to a power with the .operator

```
>> A.^2
ans =

     1     4     9    16
    25    36    49    64
    81   100   121   144
   169   196   225   256
```

Matrix Transpose:

Taking the transpose of a matrix is done by a single quote following the matrix:

```
>> A'

ans =

     1     5     9    13
     2     6    10    14
     3     7    11    15
     4     8    12    16

>> A'*A

ans =

   276   304   332   360
   304   336   368   400
   332   368   404   440
   360   400   440   480
```

Concatenating matrices:

We can make matrices by concatenating to the rows or columns of a matrix:

```
>>[A A]

ans =

     1     2     3     4     1     2     3     4
     5     6     7     8     5     6     7     8
     9    10    11    12     9    10    11    12
    13    14    15    16    13    14    15    16

>>[A; A]

ans =

     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16
```

Common matrices:

There are many built in matrix creation routines:

```
>>ones(4,4)

ans =

     1     1     1     1
     1     1     1     1
     1     1     1     1
     1     1     1     1

>> zeros(3,2)

ans =

     0     0
     0     0
     0     0
```

```
>> eye(4)

ans =

     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
```

Functions:

You can script your own functions for use in matlab scripts or the interpreter. The syntax is pretty streamlined. There are no type declarations and there is no explicit return statement:

```
function y = sigmoid(x)
  y = 1 ./(1 + exp(-x));
```

```
>> sigmoid(A)

ans =

    0.7311    0.8808    0.9526    0.9820
    0.9933    0.9975    0.9991    0.9997
    0.9999    1.0000    1.0000    1.0000
    1.0000    1.0000    1.0000    1.0000
```

Notice that the matrix x is getting a scalar added to it and then a scalar is divided by a matrix. For elementwise operations Matlab will broadcast a scalar to the appropriate shape by 'copying' the scalar value into an appropriately sized matrix so that the operation is mathematically well defined.

Up to now we have been printing all the output of operations to screen. In order to suppress output use a semicolon at the end of a line. Also inline comments are begun with a **%** symbol.

```
>> sigmoid(A); % presents no output
```

Dimensionality Reduction and Visualization

Word embeddings:

A word embedding is a representation of a word which tries to represent the semantics of a word by a real valued vector. We have some 128 dimensional vector representations of words we can map down into 2-space using tSNE dimensionality reduction technique. For fun we'll plot the results to see if our embeddings make sense semantically.

First we load the vector representations of the words. Matlab can easily load space and line delimited text files of values like we have been using in class, into matrix variables:

```
>>load wordvecs.densetxt; % gets loaded into a variable wordvecs
```

Now we reduce the dimension of wordvecs using the third party tsne reduction script (for demonstration we will just do the first 500 words as the full set takes about 15 minutes to reduce).

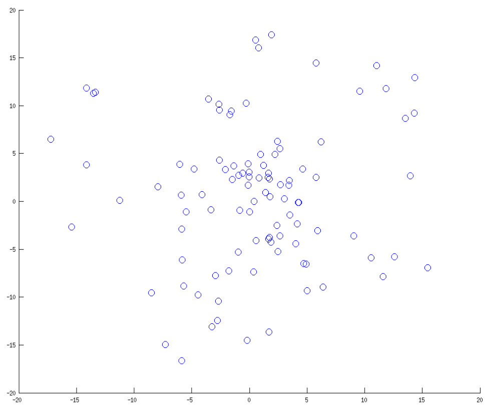
```
>>mappedsmall = tsne(wordvecs(1:500, :), [], 2);
```

Since the script is in the same folder matlab knows where to find the code. If you have matlab scripts in other folders use the addpath command:

```
>>addpath <path to script>
```

Let's make a scatter plot of the reduced word vectors.

```
>> scatter(mappedsmall(1:100,1), mappedsmall(1:100,2))
```



It would be nice to see what words these points represent and check to see if similar words are close to them. To get the labels we'll load the words from a text file.
You can read an entire ascii file into a string like so:

```
>>labeltext = fileread('labels.txt');
```

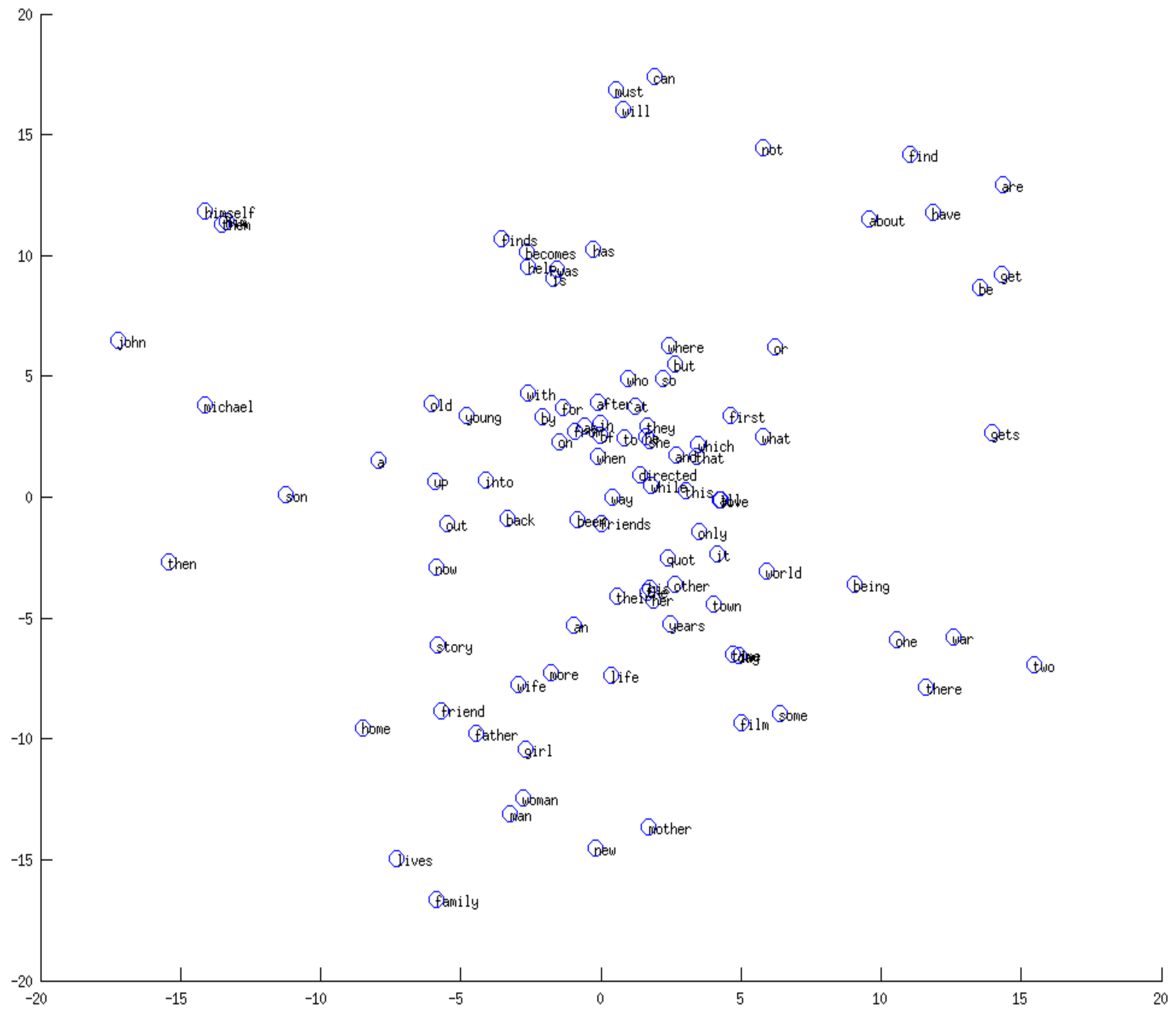
Each word is on its own line on the file so we split the words into a matlab cell array.

```
>> labels = strsplit(labeltext, '\n');
```

And label the points:

```
>> text(mappedsmall(1:100,1) + .1, mappedsmall(1:100,2), labels(2:101));
```

Notice we added a little to the x-dimension when specifying the label locations so that our labels don't sit directly on top of the points.

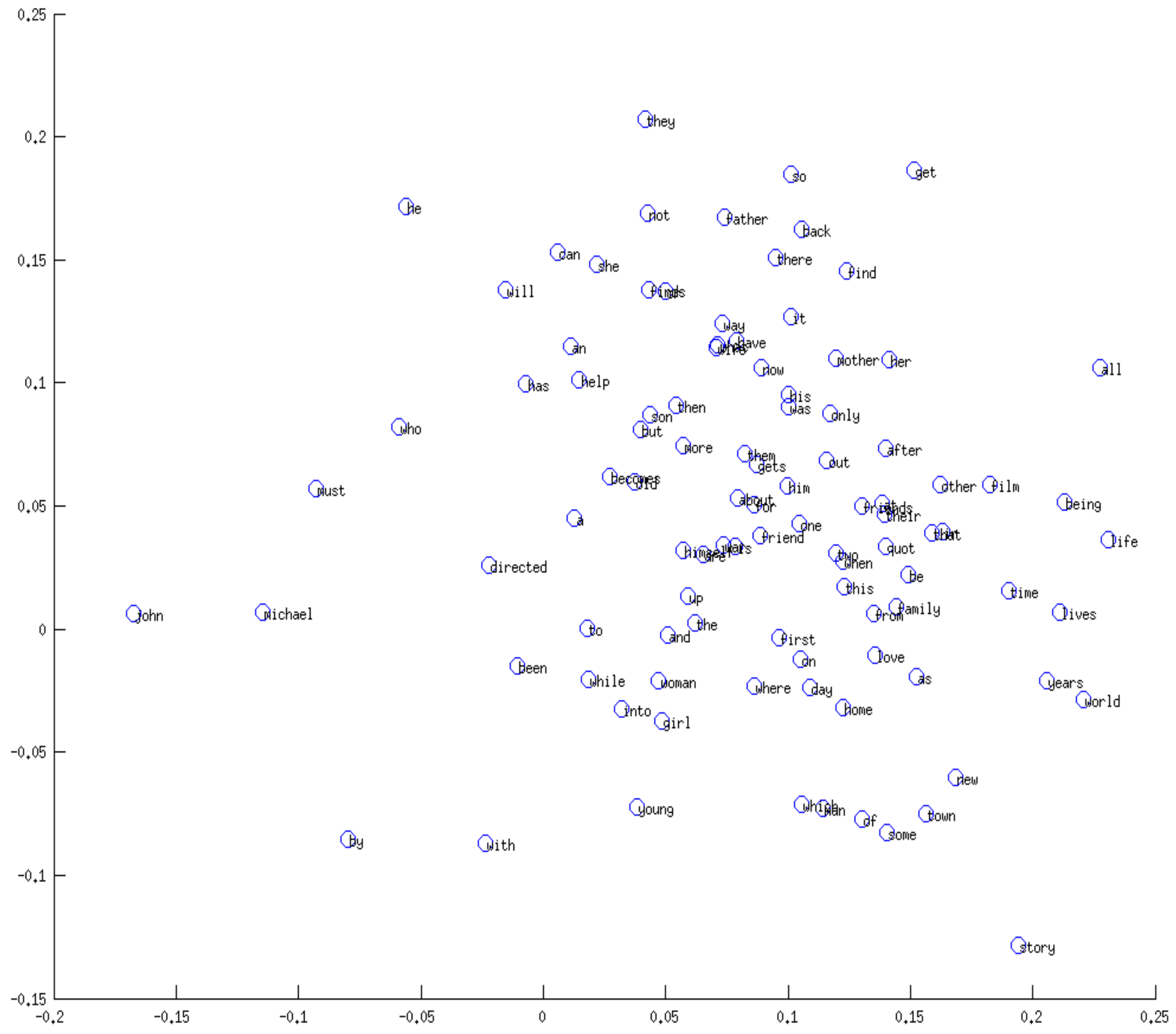


We can also use a simpler method of reduction, PCA, to get a different two dimensional representation of our word vectors:

```
>>[coeff,score] = pca(wordvecs);
>>reduced_vecs = score(:,1:2);
```

We can plot this 2D representation of words like before:

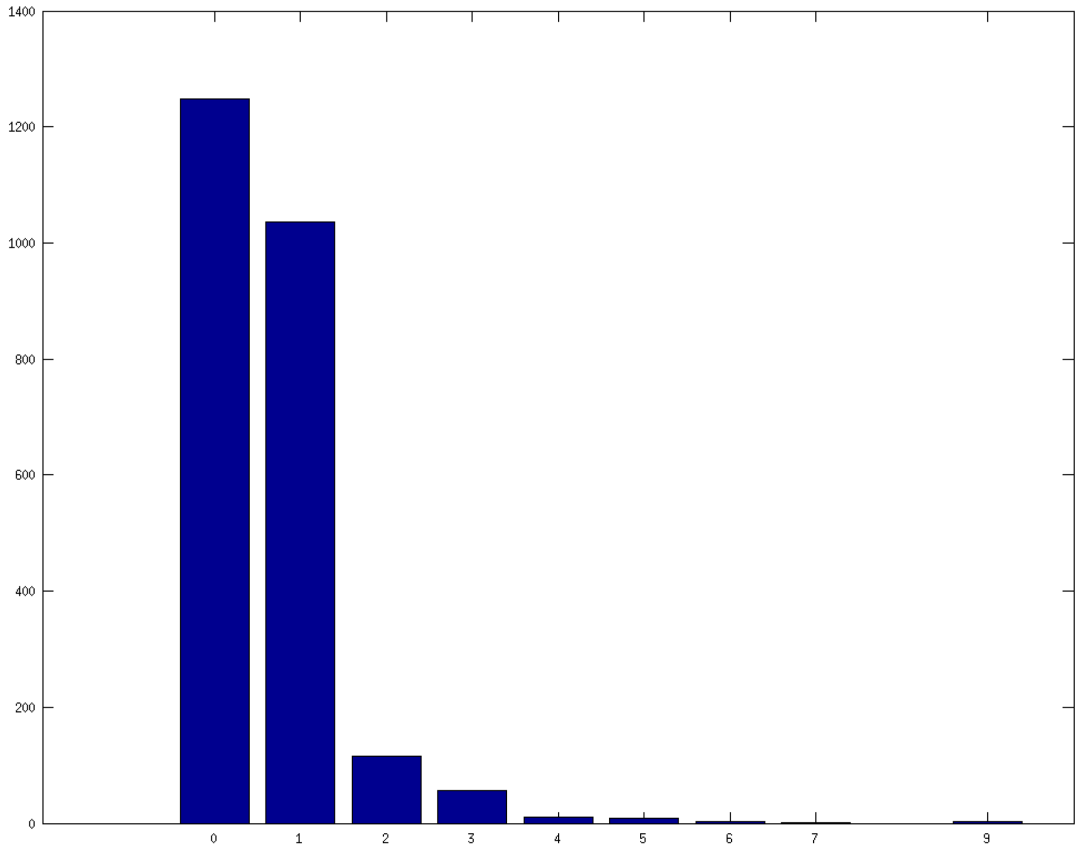
```
>> scatter(reduced_vecs(1:100,1), reduced_vecs(1:100,2))
>> text(reduced_vecs(1:100,1) + .001, reduced_vecs(1:100,2),
labels(2:101));
```



Visualizing Distribution of Labels:

For project 3 we had a bunch of classification data. Matlab provides a nice way to plotting the distribution of the labels using a bar chart.

```
>> load dataset1.dev_targets.txt;
>> x = dataset1_dev_targets;
>> z = unique(x)

z =
     0
     1
     2
     3
     4
     5
     6
     7
     9

>> [n, xout] = hist(x, z);
>> bar(xout, n);
```



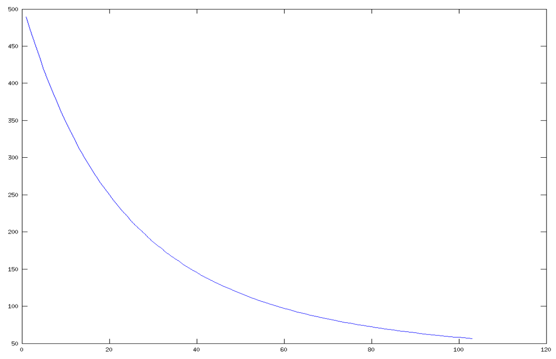
We can save this plot using the print function (the first argument specifies a name for the file and the second specifies the format to save to).

```
>> print('barplot', '-dpng')
```

Visualizing Training:

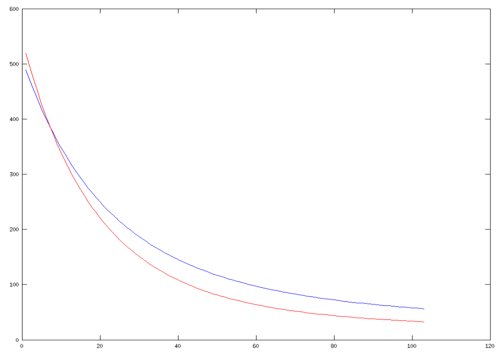
We might want to see how training progresses for a particular set of hyperparameters. I have a linear regression model that prints dev ('eval.txt') and train rmse('obj.txt') to a text file for each iteration of gradient descent.:

```
>> linear_regression(1, 'g', 0.01, 0.01);
>> deveval = load('eval.txt');
>> obj = load('obj.txt');
>> s = size(deveval)

s =

   103     1

>> plot(1:s(1), deveval)
```



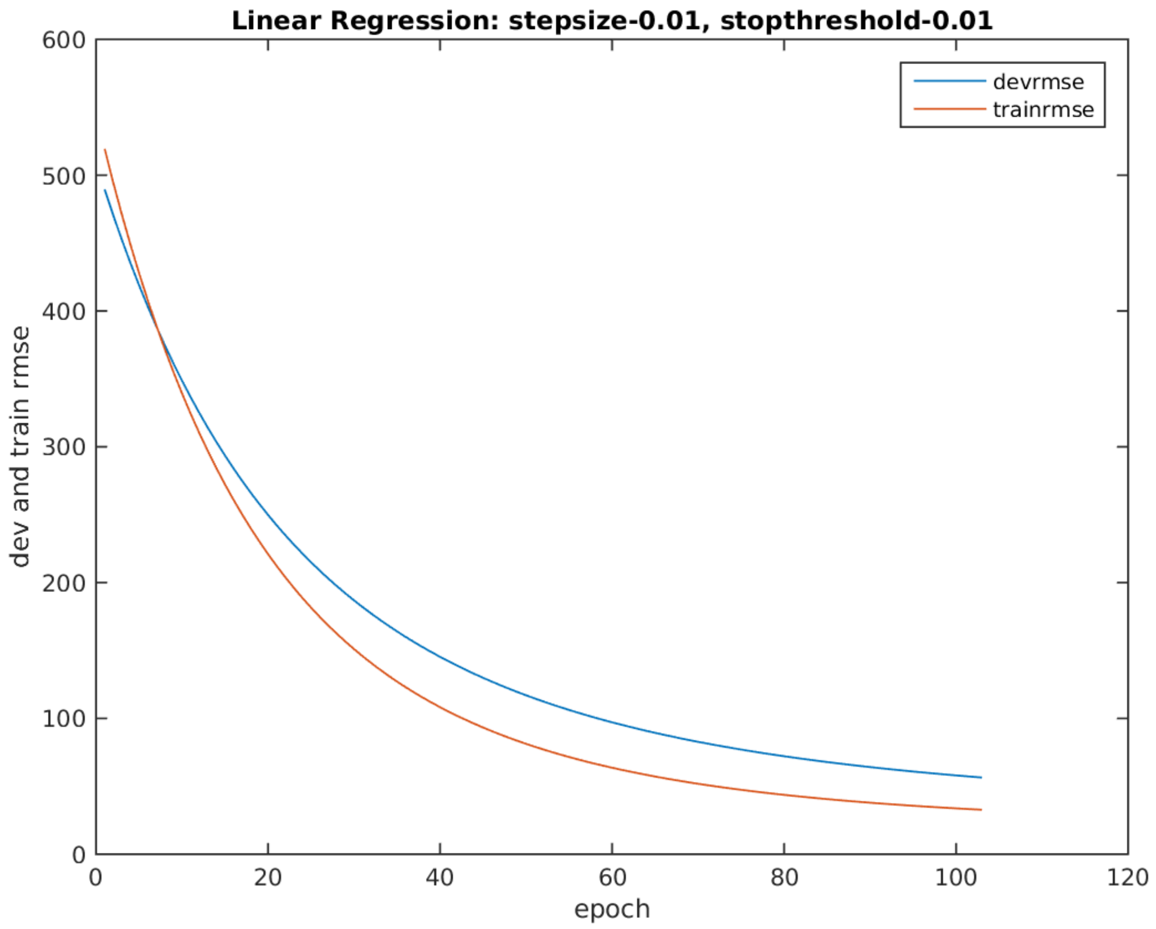
We can plot on top of the dev rmse the train rmse by using the hold command:

```
>> hold on
>> plot(1:s(1), obj, 'red')
```



Now we can add axes labels, a legend and a title to the plot.

```
>> title('Linear Regression: stepsize-0.01, stopthreshold-0.01')
>> xlabel('epoch') % x-axis label
>> ylabel('dev and train rmse') % y-axis label
>> legend('devrmse', 'trainrmse')
```



We can save the figure in matlab's format and load it later to work on it some more like so:

```
>>savefig('traindeveval');
>>openfig traindeveval
```

Sparse representations

Matlab has a datatype for matrices which contain mostly zeros, known as 'sparse' matrices. In order to save space, these matrices are represented in <row, col, value> format, so the size of the matrix is O(3*num_nonzeros) which can greatly reduce strain on memory.

You can convert a full matrix to sparse format like so:

```
>> A = sparse(A)

A =

   (1,1)         1
   (2,1)         5
   (3,1)         9
   (4,1)        13
   (1,2)         2
   (2,2)         6
   (3,2)        10
   (4,2)        14
   (1,3)         3
   (2,3)         7
   (3,3)        11
   (4,3)        15
   (1,4)         4
   (2,4)         8
   (3,4)        12
   (4,4)        16
```

Matrix operations work between sparse and dense matrices, with the caveat that :

op(sparse, dense) → dense
op(sparse, sparse) → sparse

A dense matrix can be converted to sparse like so:

```
>> A = full(A)

A =

     1     2     3     4
     5     6     7     8
     9    10    11    12
    13    14    15    16

```

Oftentimes sparse data is stored in <row, col, val> format in text files.
We can load such files into matrices but notice these aren't the dimensions we expect.

```
>> load sparsematrix.txt

>> sparsematrix(1:3, :)
ans =

     1     6     5
     1    10     3
     1    12     5

>> size(sparsematrix)
```

```
ans =

       20000            3

```

We can get our matrix in the right shape as follows:

```
>>sparsematrix = sparse(sparsematrix(:,1), sparsematrix(:,2), sparsematrix(:,3));

>> size(sparsematrix)
ans =

         462        1591
```