Session 3: Data exploration

Aaron Ponti

The standard MATLAB variables are not specifically designed for statistical data. Statistical data generally involves observations of multiple variables, with measurements of heterogeneous type and size.

Data may be numerical, categorical, or in the form of descriptive metadata. Fitting statistical data into basic MATLAB variables, and accessing it efficiently, can be cumbersome.

Statistics Toolbox software offers two additional types of container variables specifically designed for statistical data: *categorical arrays* and *dataset arrays*.

1 Data organization

1.1 Categorical arrays

Categorical data take on values from only a finite, discrete set of categories or **levels**. Levels may have associated **labels**.

If no ordering is encoded in the levels, the data are **nominal**. Nominal labels typically indicate the type of an observation. Examples of nominal labels are { false, true }, { male, female }, { Afghanistan, ..., Zimbabwe }. For nominal data, the numeric or lexicographic order of the labels is irrelevant (Afghanistan is not considered to be less than, equal to, or greater than Zimbabwe).

If an ordering is encoded in the levels (for example, if levels labeled {poor, satisfactory, outstanding} represent the result of an examination), the data are **ordinal**. Labels for ordinal levels typically indicate the position or rank of an observation. (In our example, an outstanding score is better than a poor one.)

For the sake of presenting the different categorical array types, we will load the Iris flower data set (or Fisher's Iris flower data set) introduced by Sir Ronald Aylmer Fisher as an example of discriminant analysis in 1936 (see http://en.wikipedia.org/wiki/Iris_flower_data_set). The dataset consists of 50 samples from each of three species of Iris flowers (*Iris setosa*, *Iris virginica* and *Iris versicolor*). Four features were measured from each sample, they are the length and the width of sepal and petal, respectively (see Figure 1).

```
>> load fisheriris     % Fisher's Iris data (1936)
```

loads the variables `meas` and `species` into the MATLAB workspace. The `meas` variable is a 150-by-4 numerical matrix, representing the 50 x 3 = 150 observations of 4 the different measured variables. In statistics, **observations** are represented by the **rows** of the matrix, while the **measured variables** (in our case *sepal length*, *sepal width*, *petal length*, and *petal width*) are represented by the (four) **columns**.

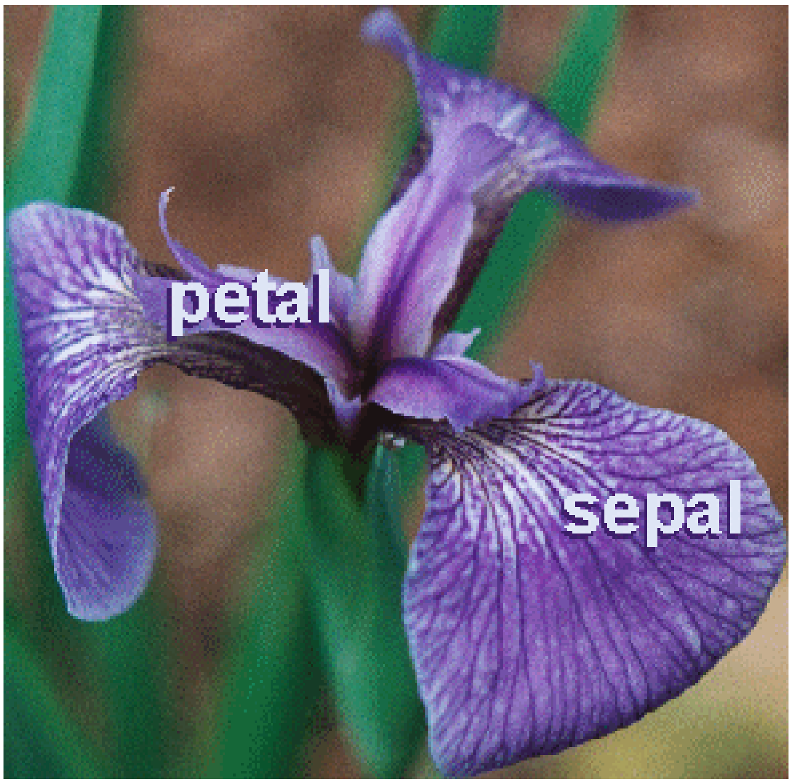

Figure 1: Petals and sepals in Iris.

The 150-by-1 cell array `species` contains 50 x 3 = 150 labels ({’setosa’, ’setosa’, ..., ’setosa’, ’versicolor’, ’versicolor’, ..., ’versicolor’, ’virginica’, ’virginica’, ..., ’virginica’ }) that relates each of the measurement to one of the three Iris species.

```
>> meas( 1 : 10, : )

ans =
    5.1000 3.5000 1.4000 0.2000
    4.9000 3.0000 1.4000 0.2000
    4.7000 3.2000 1.3000 0.2000
    4.6000 3.1000 1.5000 0.2000
    5.0000 3.6000 1.4000 0.2000
    5.4000 3.9000 1.7000 0.4000
    4.6000 3.4000 1.4000 0.3000
    5.0000 3.4000 1.5000 0.2000
    4.4000 2.9000 1.4000 0.2000
    4.9000 3.1000 1.5000 0.1000
```

outputs the first 10 observations, which belong to the *setosa* species.

1.1.1 Nominal arrays

We will use the species cell array from the fisheriris data to construct a nominal array. Nominal arrays are constructed with the `nominal` function. Type:

```
>> help nominal
```

to obtain detailed help on the function (see also section 4.1 in the Annex). We will use here the simplest constructor:

```
>> ndata = nominal( species );
>> getlabels( ndata )

ans =
    'setosa' 'versicolor' 'virginica'
```

The labels are assigned in the same sequence as they were found in the `species` cell array, but are not given any numerical or lexicographic order. One can override the name of the labels and their matching to the entries in the `species` array by specifying two additional input parameters for the nominal constructor:

```
>> ndata = nominal( species, ...
       { 'species1', 'species2', 'species3' }, ...
       {'setosa', 'versicolor', 'virginica' } );
>> getlabels( ndata )

ans =
    'species1' 'species2' 'species3'
```

The second parameter defines the names that the nominal array will use to categorize the data. They do not need to match the ones from the data. In our case, 'species1' is an alias for 'setosa', 'species2' is an alias for 'versicolor', and 'species3' is an alias for 'virginica'. If the second input is set as `{ }`, the label names will be inherited from the `species` cell array. The third argument allows to define a mapping from the labels in the cell array and those in the nominal array. Imagine that two names are swapped in the `species` array: you could fix the problem with following call:

```
>> ndata = nominal( species, ...
       { 'setosa', 'versicolor', 'virginica' }, ...
       { 'versicolor', 'setosa', 'virginica' } );
```

You can get a complete list of methods (functions) applicable to a nominal object by typing:

```
>> methods( ndata )

Methods for class nominal:
      addlevels      display       flipud        intersect
      isundefined    nominal       setdiff       subsasgn
      uint64         cat           double        getlabels
      ipermute       isvector      numel         setlabels
      subsref        uint8         cellstr       droplevels
    ...
```

and the corresponding help by typing:

```
help nominal/functionname
```

1.1.2 Ordinal arrays

An `ordinal` array is constructed exactly as a `nominal` array, with the only exception that the created levels are ordered.

```
>> odata = ordinal( species );
```

Type:

```
>> help ordinal
```

to obtain detailed help on the function (see also section 4.2 in the Annex). The optional third input parameter of the ordinal constructor overrides the default order of the labels (as found in the input data). If we wanted to give an alternative order to the labels, we could create an ordinal array as follows:

```
>> odata = ordinal( species, { }, ...
       { 'virginica', 'setosa', 'versicolor' } );
>> getlabels( odata )

ans =
    'virginica' 'setosa' 'versicolor'
```

Here, `odata` encodes an ordering of levels with `virginica` < `setosa` < `versicolor`. `ordinal` data can indeed be sorted by the order of their labels:

```
>> sortedOdata = sort( odata );
```

Exercise

while `nominal` data cannot. **Exercise**: try.

Since the fisher Iris data is not the best example for discussing ordinal data arrays, let's move to a better example.

Example

Imagine we have the following scores in an examination (maximum is 30).

```
>> scores = ...
    [ 25 27 4 27 20 7 12 18 28 28 9 28 28 17 24 8 15 27 24 28 ];
```

One cool feature of the categorical data arrays is the possibility to assign levels and labels automatically from the data as follows.

```
>> edges = 0:10:30; % We set scores between 0 and 10 to be 'poor',
                    % scores between 10 and 20 to be 'satisfactory',
                    % and scores between 20 and 30 to be 'outstanding'
>> labels = { 'poor', 'satisfactory', 'outstanding' };
>> result = ordinal( scores, labels, [ ] , edges )
```

```
result =
Columns 1 through 6
   outstanding outstanding poor outstanding outstanding poor
Columns 7 through 12
   satisfactory satisfactory outstanding outstanding poor outstanding
Columns 13 through 18
   outstanding satisfactory outstanding poor satisfactory outstanding
Columns 19 through 20
   outstanding outstanding
```

If we want to extract the outstanding scores, we can do it easily:

```
>> scores( result == 'outstanding' )

ans =
   25 27 27 20 28 28 28 28 24 27 24 28
```

Imagine now that you want to add another label for scores that are under 5.

```
>> result( scores < 5 ) = 'abysmal';
Warning: Categorical level 'abysmal' being added.
```

Notice how MATLAB automatically added the new level *abysmal*. Now let's check:

```
>> getlabels( result )

ans =
    'poor' 'satisfactory' 'outstanding' 'abysmal'

>> scores( result == 'abysmal' )

ans =
    4
```

The newly added label gets the highest order. Since this is not the desired behavior, we might want to reorder the labels as follows:

```
>> newLabels = { 'abysmal', 'poor', 'satisfactory', 'outstanding' };
>> result = reorderlevels( result, newLabels );
>> getlabels( result )

ans =
    'abysmal' 'poor' 'satisfactory' 'outstanding'
```

1.2 Dataset arrays

Dataset arrays can be viewed as tables of values, with rows representing different observations or cases and columns representing different measured variables. In this sense, dataset arrays are analogous to standard MATLAB numerical arrays. Basic methods for creating and manipulating dataset arrays parallel the syntax of corresponding methods for numerical arrays.

While each column of a dataset array must be a variable of a single type, each row may contain an observation consisting of measurements of different types. In this sense, dataset arrays lie somewhere between variables that enforce complete homogeneity on the data and those that enforce nothing. Because of the potentially heterogeneous nature of the data, dataset arrays have indexing methods with syntax that parallels corresponding methods for cell and structure arrays.

> In contrast to categorical arrays, a dataset array contains the measurement data along with the categorical data that describes it. Categorical arrays are indeed accessory objects that are used by the dataset array to categorize its measurements.

1.2.1 Constructing dataset arrays

Complete help for creating a dataset can be obtained by typing:

```
>> help dataset
```

(see also section 4.3 in the Annex). The dataset class is quite powerful. We will discuss only one way to create a dataset array here (and leave the following as an **exercise**: create a dataset from the Excel file iris.xls and the text file iris.csv that can be downloaded from the course website[1]):

Exercise

```
dataset({var1,name(s)},{var2,name(s)},...,'ObsNames',obsNames};
```

All names are optional, but help organize the data in the dataset object. You can copy the following code to a script to simplify the (re-)creation of the dataset:

```
% Fisher's Iris data (1936)
load fisheriris

% Create a nominal array called 'species' from species to
% label the measurements
n = { nominal( species ), 'species' };

% The variable names associated to the measurement
% matrix meas are 'SL', 'SW', 'PL', 'PW' (for Sepal and Petal Length
% and Width)
m = { meas, 'SL', 'SW', 'PL', 'PW' };

% Create the observation names
NumObs = size( meas, 1 );
NameObs = strcat( { 'Obs' }, num2str( ( 1 : NumObs )', '%d' ) );

% Create the dataset itself
iris = dataset( n, m, 'ObsNames', NameObs );
```

[1] http://www.fmi.ch/html/technical_resources/microscopy/homepage2/training_material.html

Run the script to create the dataset and then inspect it:

```
>> iris

iris =
              species       SL     SW     PL     PW
    Obs  1    setosa        5.1    3.5    1.4    0.2
    Obs  2    setosa        4.9      3    1.4    0.2
    Obs  3    setosa        4.7    3.2    1.3    0.2
    Obs  4    setosa        4.6    3.1    1.5    0.2
    Obs  5    setosa          5    3.6    1.4    0.2
    ...
```

The content of iris is displayed in tabular form with all rows and columns labeled by the names we specified.

The methods associated to a dataset object can be obtained as follows:

```
>> methods( iris );

Methods for class dataset:
    cat          display      get          length
    set          subsasgn     vertcat      dataset
    double       horzcat      ndims        single
    subsref      datasetfun   end          isempty
    numel        size         summary      disp
    export       join         replacedata  sortrows
    unique

Static methods:
empty
```

Help for the various commands can be obtained as follows:

```
>> help dataset.functionname
```

With the *get* and *set* modes one can access and modify the properties of the array. The dataset properties that can be accessed are 'Description', 'Units', 'DimNames', 'UserData', 'ObsNames', 'VarNames'. **Exercise**: set 'Fisher''s iris data (1936)' as the dataset description. **Exercise**

1.2.2 Accessing dataset arrays

Dataset arrays support multiple types of indexing. Like MATLAB's numerical matrices, parenthesis () indexing is used to access data subsets. Like MATLAB's cell and structure arrays, dot . indexing is used to access data variables and curly brace {} indexing is used to access data elements.

Use parenthesis indexing to assign a subset of the data in iris to a new dataset array iris1:

```
iris1 = iris(1:5,2:3)
```

```
iris1 =
         SL  SW
    Obs1 5.1 3.5
    Obs2 4.9 3
    Obs3 4.7 3.2
    Obs4 4.6 3.1
    Obs5 5 3.6
```

Similarly, use parenthesis indexing to assign new data to the first variable in iris1:

```
iris1(:,1) = dataset([5.2;4.9;4.6;4.6;5])

iris1 =
         SL  SW
    Obs1 5.2 3.5
    Obs2 4.9 3
    Obs3 4.6 3.2
    Obs4 4.6 3.1
    Obs5 5 3.6
```

Variable and observation names can also be used to access data:

```
SepalObs = iris1({'Obs1','Obs3','Obs5'},'SL')

SepalObs =
         SL
    Obs1 5.1
    Obs3 4.7
    Obs5 5
```

The following code extracts the sepal lengths in iris1 corresponding to sepal widths greater than 3:

```
BigSWLengths = iris1.SL(iris1.SW > 3)

BigSWLengths =
    5.2000
    4.6000
    4.6000
    5.0000
```

Dot indexing also allows entire variables to be deleted from a dataset array:

```
iris1.SL = []

iris1 =
         SW
    Obs1 3.5
    Obs2 3
    Obs3 3.2
    Obs4 3.1
    Obs5 3.6
```

Curly brace indexing is used to access individual data elements. The following are equivalent:

```
iris1{1,1}

ans =
    3.5

iris1{'Obs1','SW'}

ans =
    3.5
```

1.2.3 Computing with dataset arrays

The *summary*(dataset) function provides summary statistics for the component variables of a dataset array. Let's use the original Iris dataset as created in section 1.2.1.

```
>> summary( iris )

species: [150x1 nominal]
     setosa      versicolor      virginica
         50              50             50

SL: [150x1 double]
     min      1st Q       median      3rd Q       max
     4.3000   5.1000      5.8000      6.4000      7.9000

SW: [150x1 double]
     min      1st Q       median      3rd Q       max
     2        2.8000      3           3.3000      4.4000

PL: [150x1 double]
     min      1st Q       median      3rd Q       max
     1        1.6000      4.3500      5.1000      6.9000

PW: [150x1 double]
     min      1st Q       median      3rd Q       max
     0.1000   0.3000      1.3000      1.8000      2.5000
```

Notice how the summaries use **all** measurements (i.e. 150) for each of the variables, without discriminating between 'setosa', 'versicolor', or 'virginica' measurements.

1.2.4 Grouping data

Grouping variables are utility variables used to indicate which elements in a data set are to be considered together when computing statistics and creating visualizations. They may be numeric vectors, string arrays, cell arrays of strings, or categorical arrays. Logical vectors can be used to indicate membership (or not) in a single group.

Grouping variables have the same length as the variables (columns) in a data set. Observations (rows) *i* and *j* are considered to be in the same group if the values of the corresponding grouping variable are identical at those indices. Grouping variables with multiple columns are used to specify different groups within multiple variables.

To group the observations by species, the following are all acceptable (and equivalent) grouping variables:

```
>> group1 = species;          % Cell array of strings
>> group2 = grp2idx(species); % Numeric vector
>> group3 = char(species);    % Character array
>> group4 = nominal(species); % Categorical array (see section 1.2.1)
```

The following is a short list of functions that take groups as input parameter (for the complete list please see the Statistics Toolbox manual, Chapter 2: Section Grouped Data): *anova1*, *anovan*, *boxplot*, *grp2idx*, *grpstats*, *gscatter*, *kruskalwallis*, *manova1*, *tabulate*.

1.2.5 Using grouping variables

We will use groups to play with the Fisher's Iris dataset created in section 1.2.1. We will use the nominal array that we added to the iris dataset (**iris.species**) as a grouping variable. While **species**, as a cell array of strings, is itself a grouping variable, the categorical array has the advantage that it can be easily manipulated with methods of the categorical class.

Let's see for instance how many measurements for each species we have:

```
>> tabulate( iris.species )
     Value Count Percent
    setosa    50  33.33%
versicolor    50  33.33%
 virginica    50  33.33%
```

Compute some basic statistics for the data (median and interquartile range), by group, using the *grpstats* function:

```
>> [order,number,group_median,group_iqr] = ...
grpstats([iris.SL iris.SW iris.PL iris.PW], ...
iris.species,{'gname','numel',@median,@iqr})[2]

order =
    'setosa'
    'versicolor'
    'virginica'
number =
    50    50    50    50
    50    50    50    50
    50    50    50    50
group_median =
    5.0000    3.4000    1.5000    0.2000
    5.9000    2.8000    4.3500    1.3000
```

[2]Either function handles (@median) or function names in character arrays ('numel') can be used.

```
    6.5000    3.0000    5.5500    2.0000
group_iqr =
    0.4000    0.5000    0.2000    0.1000
    0.7000    0.5000    0.6000    0.3000
    0.7000    0.4000    0.8000    0.5000
```

The statistics appear in 3-by-4 arrays, corresponding to the 3 groups ('setosa', 'versicolor', 'virginica') and 4 variables ('SL', 'SW', 'PL', 'PW') in the data. The order of the groups (not encoded in the nominal array group) is indicated by the group names in order.

To improve the labeling of the data, one can use the *grpstats* function **on the dataset object directly**:

```
>> stats = grpstats(iris,'species',{@median,@iqr})

stats =
                  species       GroupCount
    setosa        setosa        50
    versicolor    versicolor    50
    virginica     virginica     50

                  median_SL     iqr_SL
    setosa          5           0.4
    versicolor    5.9           0.7
    virginica     6.5           0.7

                  median_SW     iqr_SW
    setosa        3.4           0.5
    versicolor    2.8           0.5
    virginica       3           0.4

                  median_PL     iqr_PL
    setosa         1.5          0.2
    versicolor    4.35          0.6
    virginica     5.55          0.8

                  median_PW     iqr_PW
    setosa        0.2           0.1
    versicolor    1.3           0.3
    virginica       2           0.5
```

When you call *grpstats* with a dataset array as an argument, you invoke the *grpstats* method of the *dataset* class, rather than the *grpstats* function. The method has a slightly different syntax than the function, but it returns the same results, with better labeling.

The statistics calculated by the *summary* function for the sepal length of the species setosa could be called explicitly as follows:

```
% Minimun
>> min( iris.SL( iris.species == 'setosa' ) )

ans =
    4.3000

% First quartile
>> quantile( iris.SL( iris.species == 'setosa' ), 0.25 )

ans =
    4.8000

% Median
median( iris.SL( iris.species == 'setosa' ) )

ans =
     5

% Third quartile
quantile( iris.SL( iris.species == 'setosa' ), 0.75 )

ans =
    5.2000

% Maximum
max( iris.SL( iris.species == 'setosa' ) )

ans =
    5.8000
```

2 Statistical visualization

Statistics Toolbox data visualization functions add to the extensive graphics capabilities already in MATLAB.

- *Scatter plots* are a basic visualization tool for multivariate data. They are used to identify relationships among variables. Grouped versions of these plots use different plotting symbols to indicate group membership.
- *Box plots* display a five-number summary of a set of data: the median, the two ends of the interquartile range (the box), and two extreme values (the whiskers) above and below the box. Because they show less detail than histograms, box plots are most useful for side-by-side comparisons of two distributions.
- *Distribution plots* help you identify an appropriate distribution family for your data.

2.1 Scatter plots

A scatter plot is a simple plot of one variable against another that is helpful for investigating relationship among variables. MATLAB offers the functions *scatter* and *plotmatrix* to produce simple scatter plots and scatter plot matrices, where all variables are plotted against each other in pairs. The statistics toolbox adds the two functions *gscatter* and *gplotmatrix* that implement support for grouped data.

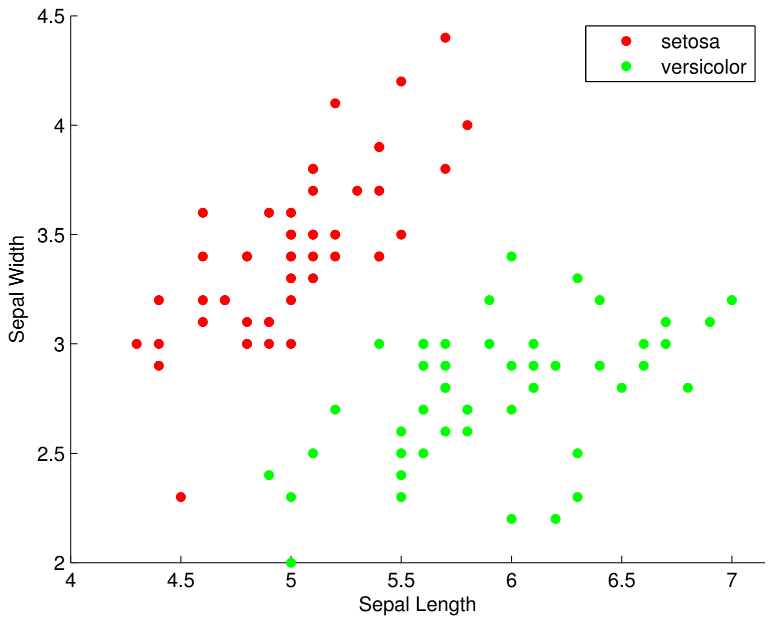

Figure 2: Scatter plot

The scatter plot in Figure 2, created with the *gscatter* function, shows the correlation between sepal length and sepal width in two species of Iris (setosa and versicolor). Use *ismember* to subset the two species from group:

```
subset = ismember(species,{'setosa','versicolor'});[3]
scattergroup = species(subset);
gscatter(iris.SL(subset),iris.SW(subset),scattergroup)
xlabel('Sepal Length')
ylabel('Sepal Width')
```

From this plot, one can say that (i) setosa sepals tend to be shorter and wider than versicolor sepals and (ii) sepal lengths and widths appear to correlate with each other (the larger the sepal length of a flower, the larger its sepal width tends to be).

If we wanted to plot all possible pairs of variables in the Iris dataset against each other, we could use the *gplotmatrix* function like this:

```
>> gplotmatrix( [ iris.SL iris.SW iris.PL iris.PW ], ...
   [ iris.SL iris.SW iris.PL iris.PW ], iris.species );
```

From figure 3 it is clear that versicolor and virginica flowers are more similar to each other in terms of sepal and petal witdhs and lenghts that they are to setosa flowers. Moreover, setosa variables tend to correlate less with each other. The points in the graphs on the diagonal lie on a straight line since they are plotted against themselves. If one sets the second matrix of measurements to [], gplotmatrix plots the first matrix

[3]Alternatively: subset = iris.species == 'setosa' | iris.species == 'versicolor';

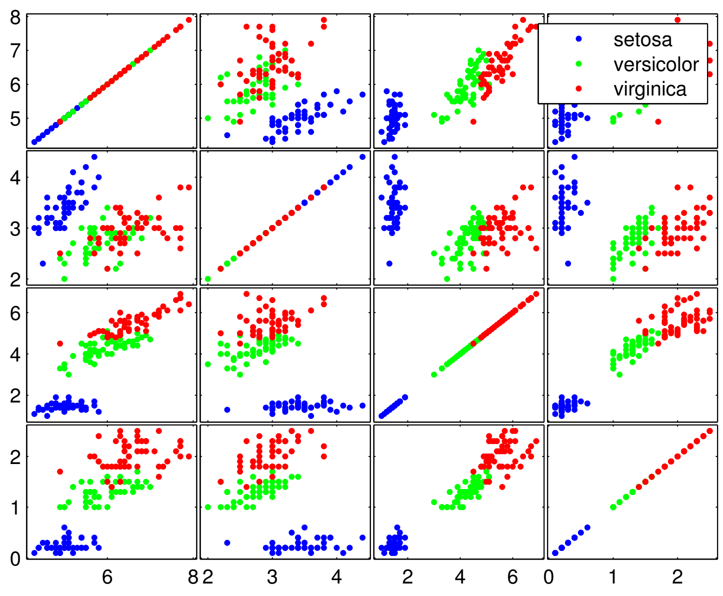

Figure 3: Scatter plot matrix

against itself, and replaces the plots on the diagonals with histograms of the various measurements (Figure 4).

```
>> gplotmatrix( [ iris.SL iris.PL ], [ ], iris.species );
```

2.2 Box plots

The graph in Figure 5, created with the *boxplot* command, compares petal lengths in samples from the three species of iris.

```
>> boxplot([iris.PL(iris.species=='setosa'), ...
    iris.PL(iris.species=='versicolor'), ...
    iris.PL(iris.species=='virginica')], 'notch', 'on', ...
    'labels', {'setosa','versicolor','virginica'});
```

This plot has the following features:

- The tops and bottoms of each "box" are the 25th and 75th percentiles of the samples (also called, as we saw, the first and third quartile), respectively. The distances between the tops and bottoms are the interquartile ranges.
- The line in the middle of each box is the sample median. If the median is not centered in the box, it shows sample skewness.
- The whiskers are lines extending above and below each box. Whiskers are drawn from the ends of the interquartile ranges to the furthest observations within the whisker length (the adjacent values).

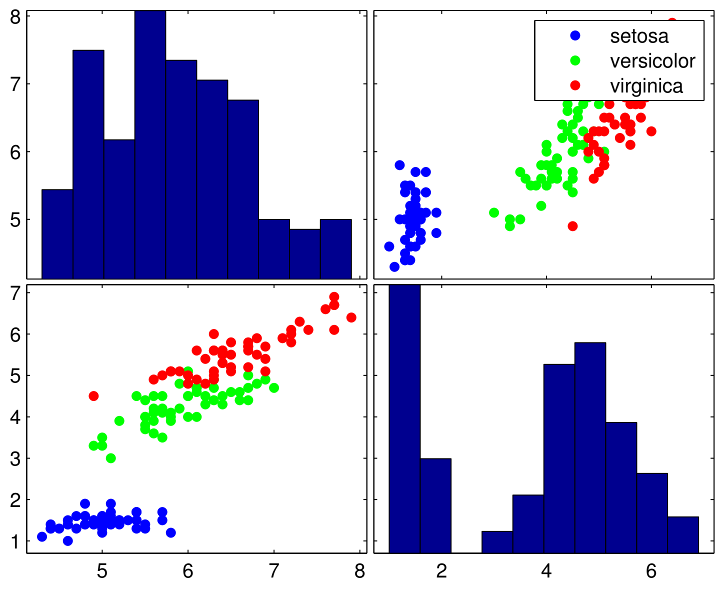

Figure 4: Scatter plot matrix with histograms

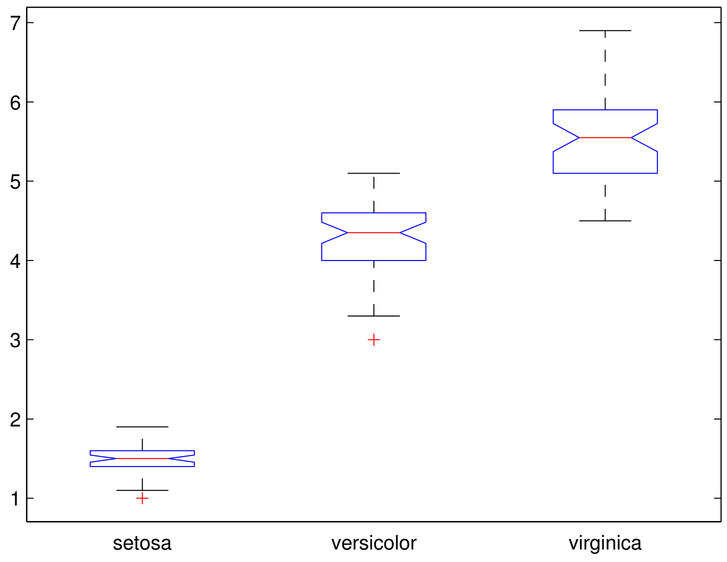

Figure 5: Box plot

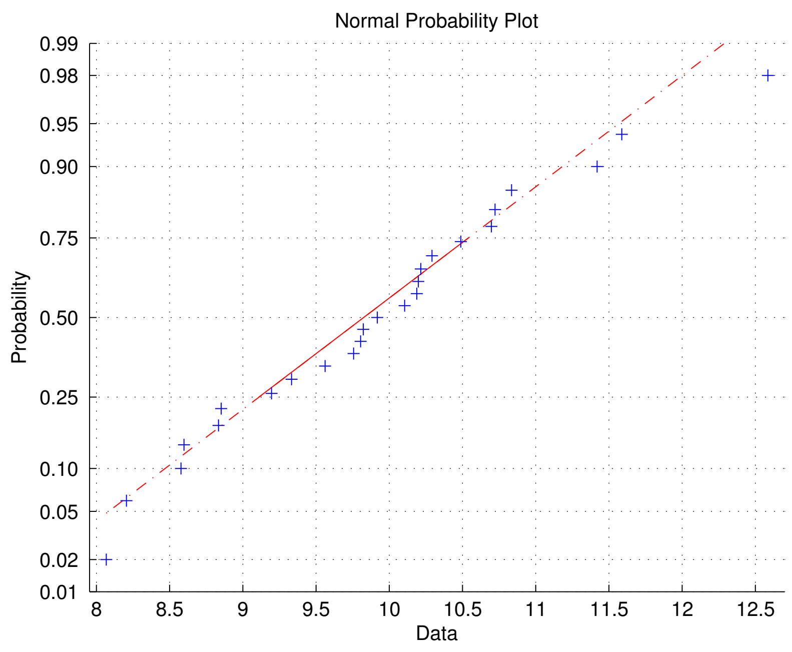

Figure 6: Normal probability plot of a normally-distributed sample

- Observations beyond the whisker length are marked as outliers. By default, an outlier is a value that is more than 1.5 times the interquartile range away from the top or bottom of the box, but this value can be adjusted with additional input arguments. Outliers are displayed with a red + sign.

- Notches display the variability of the median between samples. The width of a notch is computed so that box plots whose notches do not overlap (as in Figure 5) have different medians at the 5% significance level. The significance level is based on a normal distribution assumption, but comparisons of medians are reasonably robust for other distributions. Comparing box-plot medians is like a visual hypothesis test, analogous to the *t*-test used for means.

2.3 Distribution plots

The only probability plot we will discuss here is the **normal probability plot**. Normal probability plots are used to assess whether data comes from a normal distribution. Many statistical procedures make the assumption that an underlying distribution is normal, so normal probability plots can provide some assurance that the assumption is justified, or else provide a warning of problems with the assumption.

First we plot data sampled from a normal distribution to see what we should expect.

```
>> x = normrnd(10,1,25,1);
>> normplot(x);
```

Exercise

It is left as an **exercise** to figure how how the *normrnd* function works (*help normrnd*).

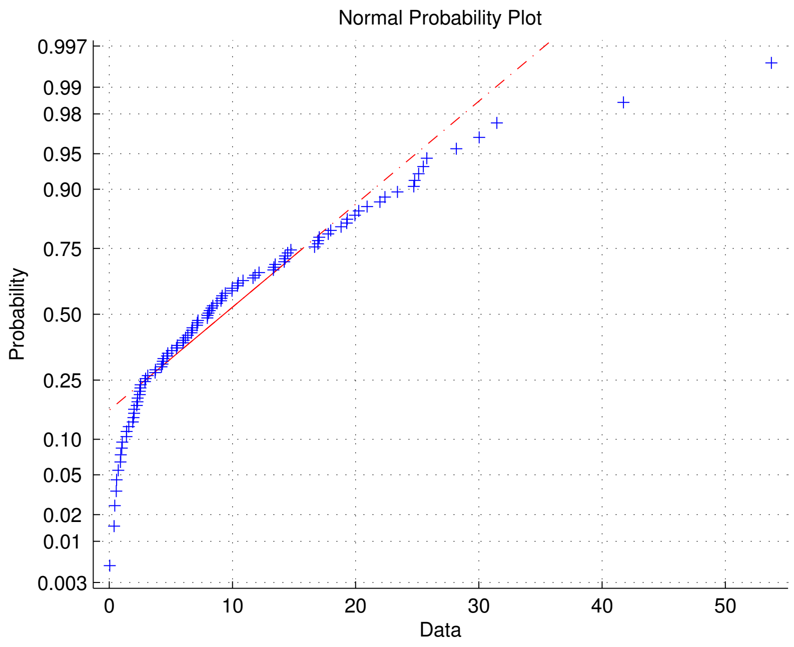

Figure 7: Normal probability plot of an exponentially-distributed sample

The plus signs in Figure 6 plot the empirical probability versus the data value for each point in the data. A solid line connects the 25th and 75th percentiles in the data, and a dashed line extends it to the ends of the data. The y-axis values are probabilities from zero to one, but the scale is not linear. The distance between tick marks on the y-axis matches the distance between the quantiles of a normal distribution. The quantiles are close together near the median (probability = 0.5) and stretch out symmetrically as you move away from the median.

In a normal probability plot, if all the data points fall near the line, an assumption of normality is reasonable. Otherwise, the points will curve away from the line, and an assumption of normality is not justified.

For example:

```
>> x = exprnd(10,100,1);
>> normplot(x);
```

The plot in Figure 7 is strong evidence that the underlying distribution is not normal.

Exercise: plot the sepal lengths for the three iris species in a normal probability plot and discuss the outcome. Does the normality assumption hold? **Exercise**

3 References

1. The official Statistics Toolbox documentation:
   http://www.mathworks.com/access/helpdesk/help/toolbox/stats/ (html).
   http://www.mathworks.com/access/helpdesk/help/pdf_doc/stats/stats.pdf (pdf)

4 Annex

The help information for the classes nominal, ordinal and dataset are printed here for reference.

4.1 help nominal

```
NOMINAL Create a nominal array.
    B = NOMINAL(A) creates a nominal array from A.  A is a numeric, logical,
    character, or categorical array, or a cell array of strings. NOMINAL
    creates levels of B from the sorted unique values in A, and creates
    default labels for them.

    B = NOMINAL(A,LABELS) creates a nominal array from A, labelling the levels
    in B using LABELS.  LABELS is a character array or cell array of strings.
    NOMINAL assigns the labels to levels in B in order according to the sorted
    unique values in A.

    B = NOMINAL(A,LABELS,LEVELS) creates a nominal array from A, with possible
    levels defined by LEVELS.  LEVELS is a vector whose values can be compared
    to those in A using the equality operator.  NOMINAL assigns labels to each
    level from the corresponding elements of LABELS.  If A contains any values
    not present in LEVELS, the levels of the corresponding elements of B are
    undefined.  Pass in [] for LABELS to allow NOMINAL to create default labels.

    B = NOMINAL(A,LABELS,[],EDGES) creates a nominal array by binning the
    numeric array A, with bin edges given by the numeric vector EDGES.  The
    uppermost bin includes values equal to the rightmost edge.  NOMINAL
    assigns labels to each level in B from the corresponding elements of
    LABELS.  EDGES must have one more element than LABELS.

    By default, an element of B is undefined if the corresponding element of A
    is NaN (when A is numeric), an empty string (when A is character), or
    undefined (when A is categorical).  NOMINAL treats such elements as
    "undefined" or "missing" and does not include entries for them among the
    possible levels for B.  To create an explicit level for those elements
    instead of treating them as undefined, you must use the LEVELS input, and
    include NaN, the empty string, or an undefined element.

    You may include duplicate labels in LABELS in order to merge multiple
    values in A into a single level in B.

    See also ordinal, histc.

    Reference page in Help browser
       doc nominal
```

4.2 help ordinal

```
ORDINAL Create an ordinal array.

    B = ORDINAL(A) creates an ordinal array from A.  A is a numeric, logical,
    character, or categorical array, or a cell array of strings. ORDINAL
    creates levels of B from the sorted unique values in A, and creates
    default labels for them.

    B = ORDINAL(A,LABELS) creates an ordinal array from A, labelling the levels
    in B using LABELS.  LABELS is a character array or cell array of strings.
    ORDINAL assigns the labels to levels in B in order according to the sorted
    unique values in A.

    B = ORDINAL(A,LABELS,LEVELS) creates an ordinal array from A, with
    possible levels and their order defined by LEVELS.  LEVELS is a vector
    whose values can be compared to those in A using the equality operator.
    ORDINAL assigns labels to each level from the corresponding elements of
    LABELS.  If A contains any values not present in LEVELS, the levels of the
    corresponding elements of B are undefined.  Pass in [] for LABELS to allow
    ORDINAL to create default labels.

    B = ORDINAL(A,LABELS,[],EDGES) creates an ordinal array by binning the
    numeric array A, with bin edges given by the numeric vector EDGES.  The
    uppermost bin includes values equal to the rightmost edge.  ORDINAL
    assigns labels to each level in B from the corresponding elements of
    LABELS.  EDGES must have one more element than LABELS.

    By default, an element of B is undefined if the corresponding element of A
    is NaN (when A is numeric), an empty string (when A is character), or
    undefined (when A is categorical).  ORDINAL treats such elements as
    "undefined" or "missing" and does not include entries for them among the
    possible levels for B.  To create an explicit level for those elements
    instead of treating them as undefined, you must use the LEVELS input, and
    include NaN, the empty string, or an undefined element.

    You may include duplicate labels in LABELS in order to merge multiple
    values in A into a single level in B.

    See also nominal, histc.

    Reference page in Help browser
       doc ordinal
```

4.3 help dataset

```
DATASET Create a dataset array.

    DS = DATASET(VAR1, VAR2, ...) creates a dataset array DS from the
    workspace variables VAR1, VAR2, ... .  All variables must have the same
    number of rows.

    DS = DATASET(..., {VAR,'name'}, ...) creates a dataset variable named
    'name' in DS.  Dataset variable names must be valid MATLAB identifiers,
    and unique.

    DS = DATASET(..., {VAR,'name1',...,'name_M'}, ...), where VAR is an
    N-by-M-by-P-by-... array, creates M dataset variables in DS, each of size
    N-by-P-by-..., with names 'name1', ..., 'name_M'.

    DS = DATASET(..., 'VarNames', {'name1', ..., 'name_M'}) creates dataset
    variables that have the specified variable names.  The names must be valid
    MATLAB identifiers, and unique.  You may not provide both the 'VarNames'
    parameter and names for individual variables.

    DS = DATASET(..., 'ObsNames', {'name1', ..., 'name_N'}) creates a dataset
    array that has the specified observation names.  The names need not be
    valid MATLAB identifiers, but must be unique.

    Dataset arrays can contain variables that are built-in types, or objects that
    are arrays and support standard MATLAB parenthesis indexing of the form
    var(i,...), where i is a numeric or logical vector that corresponds to
    rows of the variable.  In addition, the array must implement a SIZE method
    with a DIM argument, and a VERTCAT method.

    You can also create a dataset array by reading from a text or spreadsheet
    file, as described below.  This creates scalar-valued dataset variables,
    i.e., one variable corresponding to each column in the file.  Variable
    names are taken from the first row of the file.

    DS = DATASET('File',FILENAME, ...) creates a dataset array by reading
    column-oriented data in a tab-delimited text file.  The dataset variables
    that are created are either double-valued, if the entire column is
    numeric, or string-valued, i.e. a cell array of strings, if any element in
    a column is not numeric.  Fields that are empty are converted to either
    NaN (for a numeric variable) or the empty string (for a string-valued
    variable).  Insignificant whitespace in the file is ignored.

    Specify a delimiter character using the 'Delimiter' parameter name/value
    pair.  The delimiter can be any of ' ', '\t', ',', ';', '|' or their
    corresponding string names 'space', 'tab', 'comma', 'semi', or 'bar'.
    Specify strings to be treated as the empty string in a numeric column
    using the 'TreatAsEmpty' parameter name/value pair.  This may be a
    character string, or a cell array of strings.  'TreatAsEmpty' only applies
    to numeric columns in the file, and numeric literals such as '-99' are not
    accepted.
```

```
DS = DATASET('File',FILENAME,'Format',FORMAT, ...)  creates a dataset
array using the TEXTSCAN function to read column-oriented data in a text
file.  FORMAT is a format string as accepted by the TEXTSCAN function.
You may also specify any of the parameter name/value pairs accepted by
the TEXTSCAN function.

DS = DATASET('XLSFile',XLSFILENAME, ...) creates a dataset array from
column-oriented data in an Excel spreadsheet file.  You may also specify
the 'Sheet' and 'Range' parameter name/value pairs, with parameter values
as accepted by the XLSREAD function.  Variable names are taken from the
first row of the spreadsheet.  If the spreadsheet contains figures or
other non-tabular information, you should use the 'Range' parameter to
read only the tabular data.  By default, the 'XLSFile' option reads data
from the spreadsheet contiguously out to the right-most column that
contains data, including any empty columns that precede it.  If the
spreadsheet contains one or more empty columns between columns of data,
use the 'Range' parameter to specify a rectangular range of cells from
which to read variable names and data.

When reading from a text or spreadsheet file, the 'ReadVarNames' parameter
name/value pair determines whether or not the first row of the file is
treated as variable names.  Specify as a logical value (default true).
When reading from a text or spreadsheet file, the 'ReadObsNames' parameter
name/value pair determines whether or not the first column of the file is
treated as observation names.  Specify as a logical value (default false).
If the 'ReadVarNames' and 'ReadObsNames' parameter values are both true,
the name in the first column of the first row of the file is saved as the
first dimension name for the dataset.
```

See also **dataset/set**, **dataset/get**, **genvarname**, **tdfread**, **textscan**, **xlsread**.

Reference page in Help browser

```
   doc dataset
```