

Contents

1	Refinements	3
1.1	Graphs	3
1.1.1	A simple scatterplot	3
1.1.2	Labelling Graphs	4
1.1.3	Overlaying Graphs	4
1.1.4	Graph Schemes	5
1.1.5	Saving Graphs	7
1.1.6	Naming Graphs	8
1.1.7	Other Graph Types	8
1.2	Summarizing Data	9
1.2.1	<code>describe</code>	9
1.2.2	<code>codebook</code>	9
1.2.3	<code>summarize</code>	9
1.2.4	<code>tabulate</code>	10
1.3	More Command Syntax	10
1.3.1	The <code>if</code> clause	10
1.3.2	The <code>by:</code> construct	11
1.3.3	Subscripting	12
1.4	Looping	13
1.5	Wide Form vs. Long Form	14
1.5.1	Converting from long to wide	14
1.5.2	Converting from wide to long	15
1.6	Other Useful Commands	16
1.6.1	<code>display</code>	16
1.6.2	<code>expand</code>	16
1.6.3	<code>cmdlog</code>	16
2	Practical on Refinements of Stata	19
2.1	Graphs	19
2.2	Summarizing Data	19
2.3	Further Syntax	20
2.3.1	<code>if</code>	20
2.3.2	<code>by</code>	20
2.4	Looping	21
2.5	Reshaping Data	22
2.5.1	Long to Wide	22
2.5.2	Wide to Long	22

Contents

1 Refinements

1.1 Graphs

The graphical capabilities of stata were massively improved for version 8.0. However, the additional power means that the graphics system is slightly more complicated to use than previously, and also slower: it can take several seconds for a graph to appear.

All graph commands start with the word `graph`. In some cases, this can be omitted, but not always.

1.1.1 A simple scatterplot

Very commonly, you will want to plot one variable against another. This is achieved using the command `graph twoway`, or simply `twoway`. There are a number of subcommands to `twoway`, to give different types of plot. We will meet many of them later, but the simplest is `scatter`, to give a scatter plot. The effect of `twoway scatter mpg weight` using the `auto` data is given in Figure 1.1. Note that the horizontal axis (x -axis) is the second variable, and the first variable is plotted on the vertical (y) axis.

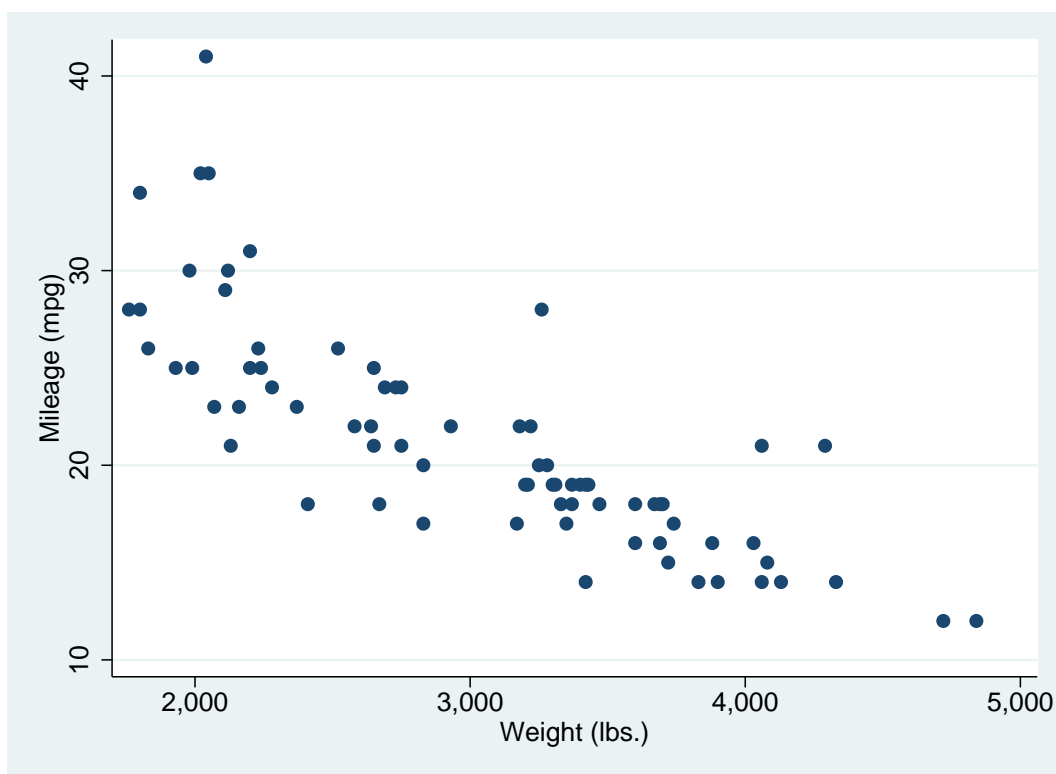


Figure 1.1: Simple Scatterplot

1.1.2 Labelling Graphs

Stata makes sensible guesses as to what titles and labels you want on your graph, but they can be changed easily if required. The options most commonly used relate to giving the graph a title^a or labelling and setting tick-marks on the axes.

The title options most commonly used are:

title The overall title, which by default appears centred at the top of the graph

subtitle A subtitle appearing centred above the graph below the overall title by default

note A note about the graph appearing left-justified below the graph

caption A caption for the graph appearing left-justified below the note, if any.

Each of these options needs to be given one or more strings (i.e. contained in inverted commas). If more than one string is given, the two strings appear on separate lines. So, for example, `title("Life Expectancy" "1900-2000")` would create the title

Life Expectancy
1900-2000

The titles for the axes are generally taken from the corresponding variable's label, or if it has no label, its name. Hence the labels "Weight (lbs)" and "Mileage (mpg)" in the Figure 1.1. However, they can be changed using the options `xtitle()` and `ytitle()`.

There are a vast number of options to control the scale, tick-marks and values labelled on the axes: see `help axis_scale_options` and `help axis_label_options`. The most important ones are `[x|y]scale(log)` to produce a log-scale for the x or y -axis, and `[x|y]label` to define the values of the axis that are to be labelled. You can use a `numlist` (which we met in session 1) to define the values, or you can use the form `xlabel(#n)`, where n is an integer, which means "create about n labels at reasonable values for me".

1.1.3 Overlaying Graphs

It is very common to want to overlay graphs. For example, you may wish to add a regression line to a scatter plot. Producing graphs of parameter estimates with confidence intervals also requires overlaying graphs.

There are two different syntaxes for combining graphs: you can either list each plot within parentheses, or separate them with two "pipe" symbols `||`. For example, suppose we want to plot both `mpg` and `length` against `weight`: we could use either

```
twoway (scatter mpg weight) (scatter length weight, yaxis(2))
```

or

```
twoway scatter mpg weight || scatter length weight, yaxis(2)
```

Both would produce the graph in Figure 1.2. The `yaxis(2)` option tell stata to use a different Y-axis for the two variables: as you can see, they cover very different ranges.

Although overlaying graphs in that way is possible, it is more useful to be able to overlay graphs of different types. For example, to overlay a regression line on the scatter-plot of `mpg` against `weight`, together with a 95% confidence region, the command would be

^aor several: there is scope for 12 differently placed titles altogether, although many of them are used very rarely

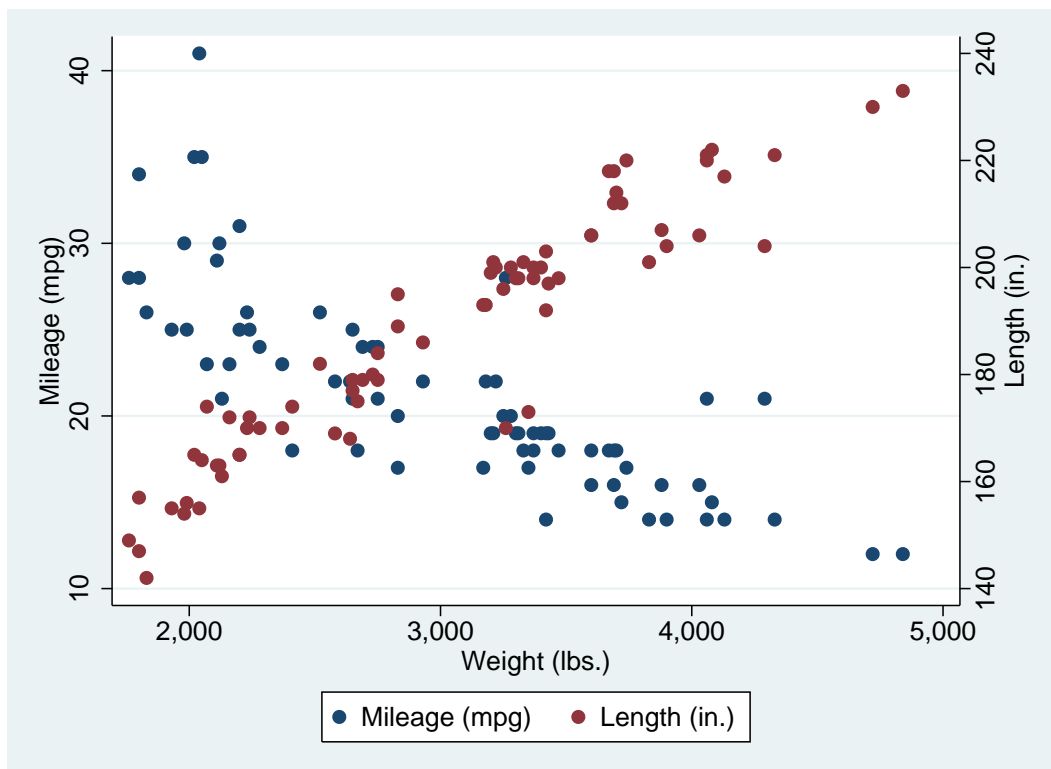


Figure 1.2: Overlaid Scatterplots

```
twoway lfitci mpg weight || scatter mpg weight
```

The results are shown in Figure 1.3. Note that plots are overlaid from left to right, so we give the `lfitci` (Linear FIT with Confidence Interval) plot first. If it were given second, the points lying within the confidence region would be hidden by the shading of the confidence region.

1.1.4 Graph Schemes

You often want to present exactly the same data in exactly the same way, but with slight differences in presentation. For example, a simple black graph on a white background would be suitable for a journal, but if you were presenting the same data in a Powerpoint presentation, you may wish to use a coloured background, make the lines bolder, increase the size of the lettering relative to the overall size of the graph etc. This can be achieved very simply with schemes.

There are 9 schemes delivered with stata, although it is also possible to define your own^b. The available schemes are listed in Table 1.1. In general, the `s1` family are simpler than the `s2` family.

To set a scheme for the rest of your stata session, the command is

```
set scheme scheme_name
```

^bBasically, copy an existing scheme to your own do-file directory, save it with a new name, and edit it to achieve the effects you want.

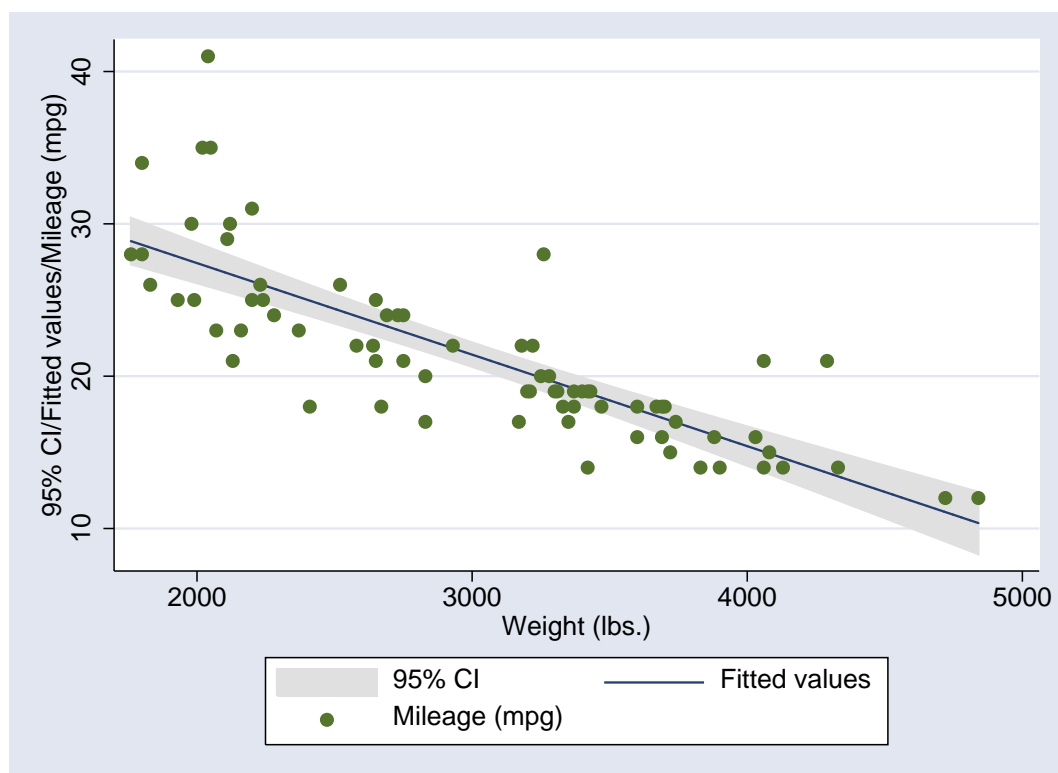


Figure 1.3: Scatterplot with overlaid regression line and confidence interval

Family	Scheme	Foreground	Background	Description
s1	s1color	coloured	white	colour on white
	s1rcolor	coloured	black	colour on black
	s1mono	monochrome	white	grey on white
	s1manual	monochrome	white	s1mono , but smaller
s2	s2color	coloured	white	default
	s2mono	monochrome	white	grey on white
	s2manual	monochrome	white	s2mono , but smaller
others	sj	monochrome	white	Used in Stata Journal
	economist	colour	white	copied from <i>The Economist</i>

Table 1.1: Available Graphics Schemes

If you want that scheme as your default scheme for all your stata sessions, use

```
set scheme scheme_name, permanently
```

Alternatively, you set set a scheme for a single graph by adding the scheme name as the `scheme` option, e.g.

```
twoway scatter mpg weight, schemesj
```

To give an example of how the scheme can affect the appearance of a graph, Figure 1.4 was created using the commands

```
set scheme economist
twoway lfitci mpg weight || scatter mpg weight
```

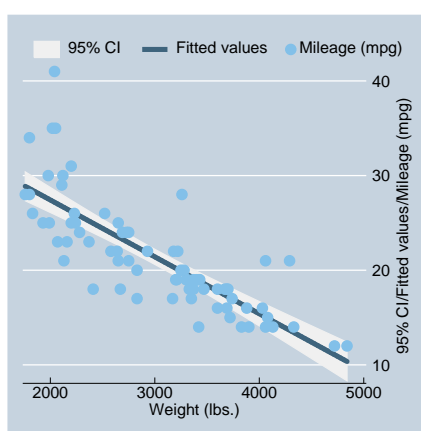


Figure 1.4: Simple scatterplot in *Economist* scheme

1.1.5 Saving Graphs

Graphs can be saved in stata’s own graphical format with the command `graph save`. Graphs can be saved either “live”, which is the default, or “as-is” using the option `asis`. A live graph can be recalled to stata and its appearance altered by using a different scheme, whilst an as-is graph will always look exactly as it did when it was saved.

It is also possible to save graphs in other formats using the command `graph export`. The format of the graph is determined by the file suffix, but this can be overridden by using an option to the command if an unusual file suffix is required. The file formats available are listed in Table 1.2.

Output format	file suffix	option
PostScript	.ps	as(ps)
Encapsulated PostScript	.eps	as(eps)
Windows Metafile	.wmf	as(wmf)
Windows Enhanced Metafile	.emf	as(emf)
Macintosh PICT Format	.pict	as(pict)

Table 1.2: Available `graph export` options

1 Refinements

For inclusion in any sensible software, the format of choice is Encapsulated PostScript. However, Powerpoint can't handle PostScript, so you need to use Metafiles or Enhanced Metafiles.

1.1.6 Naming Graphs

As well as storing a graph as a file, it is possible to store it in memory, so that stata can recall it quickly, rather than having to replot it. This is done by giving the graph a name. This can be done by using the `name` option in the graph command:

```
twoway scatter mpg weight, name(scat)
```

The graph can then be reviewed using the command

```
graph display scat
```

By default, every graph is given the name `Graph`. It is also possible to change the name of a graph using the `graph rename` command. So

```
graph rename Graph scat
```

will change the name of the currently displayed graph to `scat`, so that it is not overwritten by the next graph to be displayed.

1.1.7 Other Graph Types

This is not meant to be a complete list of all graph types in stata, but an introduction to the most commonly used ones. See `help graph` for (lots) more information

```
graph bar Bar charts
```

```
graph box Box and whisker plots
```

```
graph matrix Given  $n$  variables, creates an  $n$  by  $n$  matrix of scatterplots, plotting every variable against every other variable.
```

```
twoway histogram Histograms
```

```
twoway lfit Linear regression fit to a scatter plot
```

```
twoway qfit Quadratic regression fit to a scatter plot
```

```
twoway fffit Fractional polynomial fit to a scatter plot
```

```
twoway lowess Non-parametric smoothed fit to a scatter plot
```

```
twoway rcap Given two  $y$ -values for each  $x$ -value, plots a line between the two  $y$ -values, with "caps" at each end. Useful for showing confidence intervals if overlaid.
```

```
kdensity Can be thought of as a smoothed histogram. Very useful for comparing 2 or more distributions: see Figure 1.5
```

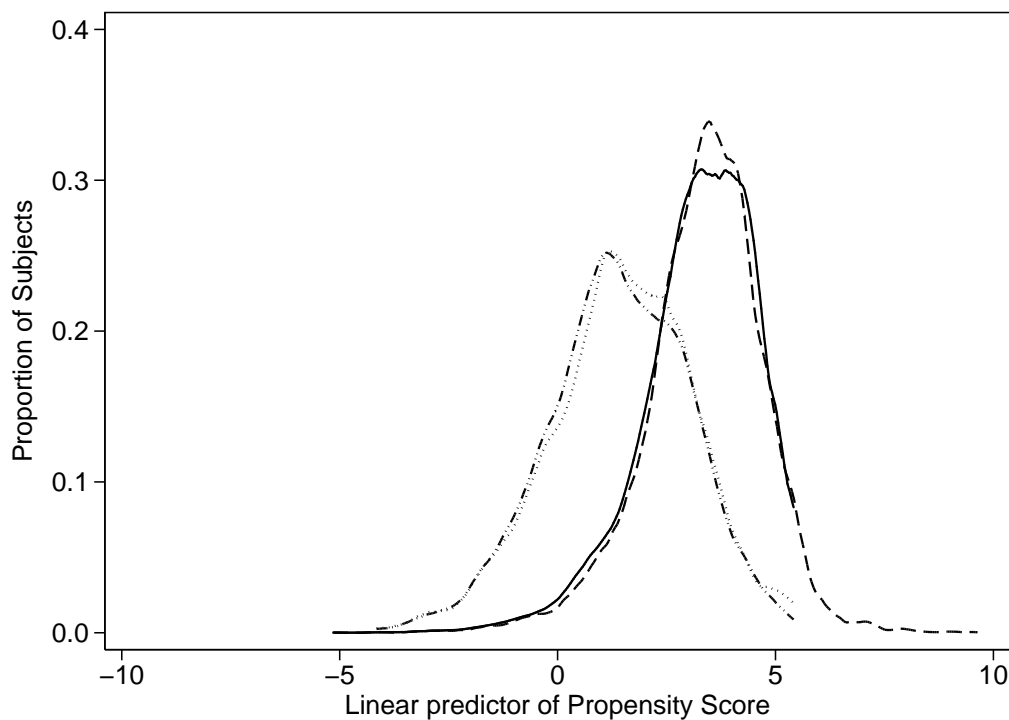


Figure 1.5: Example kernel density plot

1.2 Summarizing Data

1.2.1 *describe*

To find out what is in a dataset, you can use the command `describe`. This gives some information as to the size of the dataset, and the number of variables and observations. It also gives, for every variable, the name, type, display format^c and any labels assigned to the variable. If you are only interested in a subset of the variables, you can use `describe varlist`.

1.2.2 *codebook*

The `codebook` command gives a little more detail about each variable. This gives, for each variable, the type, range, number of unique values, number of missing values and the units used (the minimum distance between values of the variable). In addition, it gives the mean, standard deviation and various percentiles for continuous variables, and a frequency table for categorical variables (or typical values if there are too many for a full table). This command is particularly useful when data-cleaning: it makes implausible values obvious.

1.2.3 *summarize*

Another command that can be useful when inspecting data is `summarize`. In its simple form, `summarize varlist` gives the mean, standard deviation, minimum, maximum and number of non-missing values for each variable in `varlist`, in a very compact form. The option `detail`

^cthis controls how many decimal places are shown for each variable, for details type `help format`

1 Refinements

gives a fuller summary, including various percentiles, the 5 largest and 5 smallest values, the skewness and kurtosis.

1.2.4 *tabulate*

Whilst `summarize` is useful for numerical variables, the command for finding out about categorical variables is `tabulate`. The command `tabulate varname`, with a single variable, produces a frequency table for that variable. Beware: if the variable is continuous, the frequency table can be extremely long !

Another use for `tabulate` is to produce cross-tabulations of two variables. The syntax for this is

```
tabulate varname1 varname2
```

1.3 More Command Syntax

In session 1 we saw a simplified version of the syntax that (almost) all stata commands follow. However, there are some additional optional parts that we will consider this week. The full syntax we will consider is

```
[by varlist]: command varlist [if expression][, options]
```

1.3.1 *The if clause*

It is possible to restrict a command to run on only a subset of your data using an “if” clause. For example, in the `auto` data

```
summarize weight if foreign == 1
```

will produce a summary of weight only for the foreign cars, and

```
summarize weight if foreign == 0
```

will produce a summary only for the domestic cars.

Note that *two* equals signs are required: this is a hangover from the “C” programming language (used by the programmers who wrote stata), in which “=” is used to assign a value and “==” is used when comparing two values. This is probably the most irritating “feature” of stata, and one that causes a huge amount of wasted time. If you put a single equals sign by mistake, the error message is simply `invalid syntax` which is not a great help.

The operators that can be used in logical expressions are listed in Table 1.3

For example

```
if (foreign==0) | (rep78 != 3)
```

will select any cars that are either U.S. made or have a repair history score other than 3.

A very important thing to remember when using logical expressions is that the missing value is larger than any non-missing value. Hence the expression

```
if price > 15000
```

Operator	Meaning
&	and
	or
==	equal
~=	not equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

Table 1.3: Operators for logical expressions

would pick out any cars which had missing prices, as well as cars with price over \$15000. If you only want to select cars with prices known to be over \$15000, you must use

```
if (price > 15000) & (price != .)
```

This odd behaviour of the missing value is particularly important when dividing a continuous variable into categories. For example, in the first practical, you divided cars into long and short using the code

```
generate short = 0
replace short = 1 if length < 190
```

This was fine because there was no missing data in `length`. However, any cars who did have missing lengths would be given a 0 for `short`, which is not what we want. It would be better to use the code

```
generate short = 0 if length = .
replace short = 1 if length < 190
```

1.3.2 The *by:* construct

You may want to process different subgroups of your data differently. For example, you may want to calculate the mean value of a variable in men and women separately, or in cases and controls separately. This can be done using the `by:` part of the command.

The important thing to remember with the `by:` construct is that the data needs to be sorted before it is processed. Suppose that we wish to know the mean weight of foreign and domestic cars separately. The commands to use would be

```
sort foreign
by foreign: summarize weight
```

However, there is a command `bysort` which will, if used in place of `by` both sort the data and produce output for each subgroup separately.

These commands should produce the same results as

```
summarize weight if foreign == 0
summarize weight if foreign == 1
```

1 Refinements

Which method is preferable depends on

1. How many groups there are: `by` is better than `if` if there are large numbers of groups
2. Whether you need results for all groups or not
3. Whether the command is “byable”: although `by` can be used with most official stata commands, there are a few, and some user-written add-ons, which do not accept it.
4. If you need a complex expression to identify the group you are interested in, only an `if` clause will do.

1.3.3 Subscripting

A finer subdivision of your data is possible using subscripting, which effectively treats each observation as a separate group. Subscripting is achieved by putting the number of the observation that you want in square brackets after the variable name: e.g. `weight[7]` means the weight of the 7th car in the dataset.

Two macros that are commonly used in subscripting are `"_n"` and `"_N"`. The macro `"_n"` represents the number of the current observation, whilst `"_N"` represents the total number of observations.

Two particular uses of subscripting are to create “lagged” differences and to find the number of subjects in a particular subgroup.

Lagged Differences

You may wish to calculate the difference between one measurement and the next. For example, consider the blood pressure data we have seen previously. With the data in wide form (one observation and two variables per subject, see section 1.5) this can be done easily. However, if there are a large number of observations per subject, it may be easier and more efficient to leave the data in long form (several observations per subject). In this case, you want to subtract from each observation, the value of the previous observation.

patient	sex	agegroup	when	bp
1	0	1	1	143
1	0	1	2	153
2	0	1	1	163
2	0	1	2	170
3	0	1	1	153
3	0	1	2	168

Table 1.4: Extract from `bplong`

Consider the data in Table 1.4. The change in blood pressure is given by subtracting the `bp` when `when` is 1 from the value of `bp` when `when` is 2. This can be calculated as follows:

```
sort patient when
by patient: gen bpdiff = bp - bp[_n-1]
```

Subgroup size

The size of a subgroup is given by `_N` directly, with `by:` being used to define the subgroups. So, if we consider the `bpwide` dataset, we can generate a variable containing the the number of men and women with the code

```
sort sex
by sex: gen count = _N
```

1.4 Looping

You often want to perform the same command repeatedly, with only slight changes. The command `foreach` can help to do this. In its simplest form, the syntax is

```
foreach macname in list {
list of stata commands
}
```

For example, using the `bplong`

```
foreach visit in 1 2 {
summarize bp if when == 'visit'
}
```

will produce

Variable	Obs	Mean	Std. Dev.	Min	Max
bp	120	156.45	11.38985	138	185

Variable	Obs	Mean	Std. Dev.	Min	Max
bp	120	151.3583	14.17762	125	185

There are more specialised forms of `foreach`, but they are all very similar. For example,

```
foreach varname of varlist list
```

will check that each *varname* in *list* is the name of an existing variable before performing the commands. It can also expand a *varlist*, which `foreach ... in ...` cannot. Notice that all of the specialised forms use `foreach ...of listtype ...`, rather than `foreach ...in ...`.

The *varlist* form of `foreach` can be particularly useful when labelling variables. Suppose that there are a series of variables, all called `site_pain`, and all taking the value 0 for no and 1 for yes. Rather than label each variable individually, you can simplify with the following code:

```
label define yesno 0 "No" 1 "Yes"
foreach x of varlist *_pain {
label values 'x' yesno
}
```

Thus four lines of code can label as many variables as necessary.

1.5 Wide Form vs. Long Form

We saw in session 1 that when the data consists of several observations on each subject, the data can be stored either as several observations per subject (long form) or as a single observation with several variables per subject (wide form). Either form may be needed for analysis, so it is important to be able to change between them freely. This can be done with the `reshape` command.

There are two forms of the `reshape` command: `reshape wide` converts from long form to wide form, and `reshape long` converts from wide form to long form.

1.5.1 Converting from long to wide

First, we will see how to convert from long form to wide form. We will use the data listed in Table 1.5. This data consists of ID number and gender for each of three subjects, and their x-ray score at 1 year, 2 years and possibly 5 years after registration in the study.

ID	Gender	Anniversary	Score
900108	1	1	7
900108	1	2	15
900108	1	5	19
900113	2	1	0
900113	2	2	18
900114	1	1	0
900114	1	2	0

Table 1.5: X-Ray data in long form

In order to be able to reshape the data, stata needs to know:

- How to recognise which observations belong to the same subject
- How to recognise which visit a particular observation relates to
- Which variables should change between visits

In this case, we have `ID` to show which observations belong together, and `anniversary` to show which visit the observation belongs to. The only variable that should change between anniversaries is the x-ray score.

The basic syntax for the `reshape wide` command is

```
reshape wide changing-vars, i(ID-var(s)) j(visit-vars)
```

Both of the options `i()` and `j()` must be given. So in our case, we would type

```
reshape wide score, i(id) j(anniversary)
```

and the resulting dataset would look like Table 1.6

Notice that there are now 3 variables for `score`, since it changes between visits, but only one variable for `sex`, because it does not. If there is a variable that changes between visits, but it is not included in the list of variables to reshape, that is an error and stata will refuse to continue. Notice also that there are now missing values for the score at 5 years for the second and third subjects, since this data was not in the original (long) dataset.

ID	Gender	Score1	Score2	Score5
900108	1	7	15	19
900113	2	0	18	.
900114	1	0	0	.

Table 1.6: X-Ray data converted to wide form

1.5.2 Converting from wide to long

Now suppose that we have the data in Table 1.6, and need to get to long form (Table 1.5). We need to tell stata

- Which variable is a unique identifier for our subjects
- The name of a variable for it to create to contain the visit number (**anniversary** in Table 1.5)
- Which of the variables are in wide form, and need to be converted. Strictly, we give the name that the variable will take *when it is in long form*

So, the command we need will start

```
reshape long score
```

Stata will then look for all variables that begin with **score**, and therefore know that we want **score1**, **score2** and **score5** converted to **score**. Again, our identifier is **id** and the option **j** will can be given the name **anniversary**, to match that in Table 1.5. So the entire command is

```
reshape long score, i(id) j(anniversary)
```

and the result is shown in Table 1.7. This is almost identical to that in Table 1.5, but contains two extra records for 5th anniversary visits for subjects 900113 and 900114. These visits did not take place, so the data is missing.

ID	Gender	Anniversary	Score
900108	1	1	7
900108	1	2	15
900108	1	5	19
900113	2	1	0
900113	2	2	18
900113	2	5	.
900114	1	1	0
900114	1	2	0
900114	1	5	.

Table 1.7: X-Ray data converted to long form

It is important to notice that in the above example, the variables being reshaped all consisted of the prefix (**score**) followed by a numerical suffix (1, 2 or 5). The variable **anniversary** is therefore numeric. If any of the suffixes are non-numeric, you *must* use the option **string** so that stata uses a string variable for **j**. We will see an example in the practical.

1.6 Other Useful Commands

1.6.1 *display*

The command `display` simply causes its argument to appear in the results window. This can have a number of uses. For example, it can function as a simple calculator:

```
display 2+2
```

It can also be useful to check the contents of a macro, if you are not sure what it contains:

```
display "$mydir"
```

will show what the macro `$mydir` contains, or simply a blank line if `$mydir` is not defined.

1.6.2 *expand*

The command `expand` creates additional observations in your dataset. The command itself is followed by an expression which says how many identical copies of each record should be created. For example,

```
expand 2
```

will create 1 new identical copy of each existing record in the dataset.

This can be useful if you wish to recreate a dataset from a summary table in a paper. Suppose the paper contained the following table:

Exposed	Cases	Controls
No	20	40
Yes	30	10

Table 1.8: `expand` example data

Then you can enter the following data into stata:

exposed	case	frequency
0	0	40
0	1	20
1	0	10
1	1	30

If you then type

```
expand frequency
```

you will end up with a dataset containing 100 observations, and the command

```
tab exposed case
```

will recreate Table 1.8.

1.6.3 *cmdlog*

As well as being able to start a log file for your output, it is possible to start a command-log file, which contains only the commands that you have entered. This can be done at the same

time as a log file, and I recommend doing so everytime that you start stata. I have the following commands in my `profile.do` to achieve this:

```
local logname "$S_DATE $S_TIME"  
local newname : subinstr local logname " " "_", all  
local newname : subinstr local newname ":" "_", all  
local logname "C:/cmdlogs/'newname'.txt"  
cmdlog using "'logname'"
```

The first line gets the current date and time to use as the name for the log-file. The second line changes spaces to underscores, and the third line changes colons to underscores. The fourth line defines a macro to use as the name of the logfile, and the last line starts the log-file.

1 Refinements

2 Practical on Refinements of Stata

Start stata.

2.1 Graphs

Type

```
sysuse uslifeexp
```

to load up a dataset concerning life expectancy in various subgroups in the U.S. from 1900 - 2000. Produce a simple scatterplot of life-expectancy against year with the command

```
twoway scatter le year
```

You should see life expectancy increasing steadily, with a blip of very low life expectancy in 1918. Now we will add a title to this graph. Press **PageUp** to recall the previous command, and add the `title()` option:

```
twoway scatter le year, title("U.S. Life Expectancy")
```

Now we are going to extend the *y*-axis back to 0, rather than starting at 40. The option to do this is `ylabel(0(20)80)`: this will produce labels every 20 years from 0 to 80.

Now we will practice overlaying graphs, using the same dataset. We can compare male and female life-expectancy with the command

```
twoway scatter le_male year || scatter le_female year
```

You should see that life-expectancy is increasing over time in both sexes, but consistently higher in females. You can add regression lines to the plot with:

```
twoway scatter le_male year || scatter le_female year || lfit le_male year || lfit le_female year
```

(The above command must all be entered on a single line)

You should notice that the label on the *y*-axis is silly, consisting simply of the names of the four variables being plotted. A more sensible label would be given by the option `ytitle("Life Expectancy")`.

2.2 Summarizing Data

Read the cancer dataset into stata with the command

```
sysuse cancer, clear
```

Find out what the dataset is about with the command `describe`

2 Practical on Refinements of Stata

- 2.1 How many observations are there in the dataset
Now use the command `codebook` to get some idea of the values taken by the different variables.
- 2.2 What was the longest follow-up time ?
- 2.3 How many different treatments were in the study ?
- 2.4 How old were the oldest and youngest subjects in the study ?
Use the command `summarize age studytime, det` to obtain details about the ages and lengths of follow-up in the study.
- 2.5 What was the mean age at the start of the study ?
- 2.6 What was the standard deviation of the follow-up time ?
Use the command `tab drug died, row` to produce a cross-tabulation of the number, and percentage, of subjects who died on each treatment.
- 2.7 How many subjects on placebo died ?
- 2.8 What percentage of subjects on treatment 2 died ?

2.3 Further Syntax

2.3.1 *if*

- 3.1 What is the mean age of subjects in the cancer study who died ?
The command you need is `summarize age if died == 1`
- 3.2 Again using an `if` clause, find the mean followup time among subjects on placebo.....
- 3.3 What was the mean age among subjects who died after being treated on placebo ?.....

2.3.2 *by*

Use `by` clauses to verify the answers to the previous three questions. *Hint: for the last one, you need to use* `by drug died:`

Now load the `cancer` dataset with
`sysuse cancer`

Create a variable called `agegrp`, dividing subjects into 2 more-or-less equal sized age groups:
`egen agegrp = cut(age), group(2)`

To find out at what age the groups were split, enter

```
bysort agegrp: summarize age
```

3.4 How could you have found out the age at which the split would have occurred before making `agegrp`

Create a label for `agegrp` so that you know the actual age-range in each age-group appears in any printout. Assign this label to `agegrp`. Check that you were successful with

```
tabulate agegrp died
```

Create a new variable containing the number of subjects in each age-group using the command

```
bysort agegrp: gen group_size = _N
```

You can check that this has worked correctly with

```
tab group_size
```

Save this dataset as `mycancer`, as we will need to use it later.

2.4 Looping

For a simple illustration of looping, type

```
foreach x in one two three {
display " 'x' "
}
```

(The opening single inverted comma is at the top left of the keyboard, the closing one at the right hand end of the “asdf” row).

You should see the words “one”, “two” and “three” appear, one on each line.

You should still have the dataset called `mycancer` in `stata`. If not, load it using

```
use mycancer
```

You can now enter the following code:

```
foreach x in drug agegrp {
tab 'x' died, row
}
```

This should produce two cross-tabulations, one for `drug` against `died`, and the other for `agegrp` against `died`.

For a more complex example of using `foreach`, load the life-expectancy data with

```
sysuse uslifeexp
```

We will create graphs for all of the variables with the code

```
foreach x of varlist le* {  
  twoway scatter 'x' year, name("'x'")  
}
```

In the name option, you must put inverted commas around the local macro 'x' so that stata knows that it is a string. This will produce a series of scatterplots called `le`, `le_male` etc. You can recall one of these graphs with, for example

```
graph display le_female
```

2.5 Reshaping Data

2.5.1 Long to Wide

Read the `bplong` dataset into stata with the command

```
sysuse bplong
```

Use a `by` clause to get the mean and standard deviation of the blood pressure when `when == 1` and `when == 2` separately. Record these you that you can check later that your reshaping was successful.

To reshape this data into wide form, the unique identifier is `patient` and `j` is `when`. Therefore, the command you need is

```
reshape wide bp, i(patient) j(when)
```

Now use the command

```
summarize bp1 bp2
```

to ensure that the data has been transformed correctly.

2.5.2 Wide to Long

Load the life-expectancy data into stata, using the command

```
sysuse uslifeexp, clear
```

There are series of variables giving the life-expectancy in different subgroups of the U.S. population over the years, and a variable `le` containing the overall life-expectancy. We will change this to have several observations for each year, a single variable `le` containing the life-expectancy and a variable `group` saying which subgroup the life-expectancy applies to.

First, we need to change the name of the variable `le` to `le_total`, so that there is something to put in the `group` variable for the overall life-expectancy. The command to do this is

```
rename le le_total
```

Now, unique observations are identified by the variable `year`, the variable we want to have in the long data is `le`, and the variable we want to identify which variable in wide form corresponds to an observation in long form is called `group`, so the reshape command we need is

```
reshape long le, i(year) j(group) string
```

The string part is because the parts that follow `le` in the variable names are not numbers, but strings, so `group` must be a string variable.

If you now enter

```
twoway scatter le year if group == "_male" || scatter le year if group == "_female"
```

you should get the same graph as we have seen earlier.