2 Data management and graphics

2.1 Introduction

The starting point of an empirical investigation based on microeconomic data is the collection and preparation of a relevant dataset. The primary sources are often government surveys and administrative data. We assume the researcher has such a primary dataset and do not address issues of survey design and data collection. Even given primary data, it is rare that it will be in a form that is exactly what is required for ultimate analysis.

The process of transforming original data to a form that is suitable for econometric analysisis referred to as data management. This is typically a time-intensive task that has important implications for the quality and reliability of modeling carried out at the next stage.

This process usually begins with a data file or files containing basic information extracted from a census or a survey. They are often organized by data record for a sampled entity such as an individual, a household, or a firm. Each record or observation is a vector of data on the qualitative and quantitative attributes of each individual. Typically, the data need to be cleaned up and recoded, and data from multiple sources may need to be combined. The focus of the investigation might be a particular group or subpopulation, e.g., employed women, so that a series of criteria need to be used to determine whether a particular observation in the dataset is to be included in the analysis sample.

In this chapter, we present the tasks involved in data preparation and management. These include reading in and modifying data, transforming data, merging data, checking data, and selecting an analysis sample. The rest of the book focuses on analyzing a given sample, though special features of handling panel data and multinomial data are given in the relevant chapters.

2.2 Types of data

All data are ultimately stored in a computer as a sequence of 0s and 1s because computers operate on binary digits, or bits, that are either 0 or 1. There are several different ways to do this, with potential to cause confusion.

2.2.1 Text or ASCII data

A standard text format is ASCII, an acronym for American Standard Code for Information Interchange. Regular ASCII represents $2^7 = 128$ and extended ASCII represents $2^8 = 256$ different digits, letters (uppercase and lowercase), and common symbols and punctuation marks. In either case, eight bits (called a byte) are used. As examples, 1 is stored as 00110001, 2 is stored as 00110010, 3 is stored as 00110011, A is stored as 01010001, and a is stored as 00110001. A text file that is readable on a computer screen is stored in ASCII.

A leading text-file example is a spreadsheet file that has been stored as a "comma-separated values" file, usually a file with the `.csv` extension. Here a comma is used to separate each data value; however, more generally, other separators can be used.

Text-file data can also be stored as fixed-width data. Then no separator is needed provided we use the knowledge that, say, columns 1–7 have the first data entry, columns 8–9 have the second data entry, and so on.

Text data can be numeric or nonnumeric. The letter a is clearly nonnumeric, but depending on the context, the number 3 might be numeric or nonnumeric. For example, the number 3 might represent the number of doctor visits (numeric) or be part of a street address, such as 3 Main Street (nonnumeric).

2.2.2 Internal numeric data

When data are numeric, the computer stores them internally using a format different from text to enable application of arithmetic operations and to reduce storage. The two main types of numeric data are integer and floating point. Because computers work with 0s and 1s (a binary digit or bit), data are stored in base-2 approximations to their base-10 counterparts.

For integer data, the exact integer can be stored. The size of the integer stored depends on the number of bytes used, where a byte is eight bits. For example, if one byte is used, then in theory $2^8 = 256$ different integers could be stored, such as $-127, -126, \dots, 127, 128$.

Noninteger data, or often even integer data, are stored as floating-point data. Standard floating-point data are stored in four bytes, where the first bit may represent the sign, the next 8 bits may represent the exponent, and the remaining 23 bits may represent the digits. Although all integers have an exact base-2 representation, not all base-10 numbers do. For example, the base-10 number 0.1 is $0.0\overline{0011}$ in base 2. For this reason, the more bytes in the base-2 approximation, the more precisely it approximates the base-10 number. Double-precision floating-point data use eight bytes, have about 16 digits precision (in base 10), and are sufficiently accurate for statistical calculations.

Stata has the numeric storage types listed in table 2.1: three are integer and two are floating point.

Table 2.1. Stata's numeric storage types

Storage type	Bytes	Minimum	Maximum
`byte`	1	-127	100
`int`	2	$-32,767$	32,740
`long`	4	$-2,147,483,647$	2,147,483,620
`float`	4	$-1.70141173319 \times 10^{38}$	$1.70141173319 \times 10^{38}$
`double`	8	$-8.9984656743 \times 10^{307}$	$8.9984656743 \times 10^{307}$

These internal data types have the advantage of taking fewer bytes to store the same amount of data. For example, the integer 123456789 takes up 9 bytes if stored as text but only 4 bytes if stored as an integer (`long`) or floating point (`float`). For large or long numbers, the savings can clearly be much greater. The Stata default is for floating-point data to be stored as `float` and for computations to be stored as `double`.

Data read into Stata are stored using these various formats, and Stata data files (`.dta`) use these formats. One disadvantage is that numbers in internal-storage form cannot be read in the same way that text can; we need to first reconvert them to a text format. A second disadvantage is that it is not easy to transfer data in internal format across packages, such as transferring Excel's `.xls` to Stata's `.dta`, though commercial software is available that transfers data across leading packages.

It is much easier to transfer data that is stored as text data. Downsides, however, are an increase in the size of the dataset compared with the same dataset stored in internal numeric form, and possible loss of precision in converting floating-point data to text format.

2.2.3 String data

Nonnumeric data in Stata are recorded as strings, typically enclosed in double quotes, such as "3 Main Street". The format command `str20`, for example, states that the data should be stored as a string of length 20 characters.

In this book, we focus on numeric data and seldom use strings. Stata has many commands for working with strings. Two useful commands are `destring`, which converts string data to integer data, and `tostring`, which does the reverse.

2.2.4 Formats for displaying numeric data

Stata output and text files written by Stata format data for readability. The format is automatically chosen by Stata but can be overridden.

The most commonly used format is the f format, or the fixed format. An example is `%7.2f`, which means the number will be right-justified and fill 7 columns with 2 digits after the decimal point. For example, 123.321 is represented as `123.32`.

The format type always begins with `%`. The default of right-justification is replaced by left-justification if an optional `-` follows. Then follows an integer for the width (number of columns), a period (`.`), an integer for the number of digits following the decimal point, and an `e` or `f` or `g` for the format used. An optional `c` at the end leads to comma format.

The usual format is the f format, or fixed format, e.g., `123.32`. The e, or exponential, format (scientific notation) is used for very large or small numbers, e.g., `1.23321e+02`. The g, or general format, leads to `e` or `f` being chosen by Stata in a way that will work well regardless of whether the data are very large or very small. In particular, the format `%#.(#-1)g` will vary the number of columns after the decimal point optimally. For example, `%8.7g` will present a space followed by the first six digits of the number and the appropriately placed decimal point.

2.3 Inputting data

The starting point is the computer-readable file that contains the raw data. Where large datasets are involved, this is typically either a text file or the output of another computer program, such as Excel, SAS, or even Stata.

2.3.1 General principles

For a discussion of initial use of Stata, see chapter 1. We generally assume that Stata is used in batch mode.

To replace any existing dataset in memory, you need to first clear the current dataset.

```
. * Remove current dataset from memory
. clear
```

This removes data and any associated value labels from memory. If you are reading in data from a Stata dataset, you can instead use the `clear` option with the `use` command. Various arguments of `clear` lead to additional removal of Mata functions, saved results, and programs. The `clear all` command removes all these.

Some datasets are large. In that case, we need to assign more memory than the Stata default by using the `set memory` command. For example, if 100 megabytes are needed, then we type

```
 * Set memory to 100 mb
. set memory 100m
```

Various commands are used to read in data, depending on the format of the file being read. These commands, discussed in detail in the rest of this section, include the following:

- use to read a Stata dataset (with extension .dta)
- edit and input to enter data from the keyboard or the Data Editor
- insheet to read comma-separated or tab-separated text data created by a spreadsheet
- infile to read unformatted or fixed-format text data
- infix to read formatted data

As soon as data are inputted into Stata, you should save the data as a Stata dataset. For example,

```
. * Save data as a Stata dataset
. save mydata.dta, replace
  (output omitted)
```

The replace option will replace any existing dataset with the same name. If you do not want this to happen, then do not use the option.

To check that data are read in correctly, list the first few observations, use describe, and obtain the summary statistics.

```
. * Quick check that data are read in correctly
. list in 1/5        // list the first five observations
  (output omitted)
. describe           // describe the variables
  (output omitted)
. summarize          // descriptive statistics for the variables
  (output omitted)
```

Examples illustrating the output from describe and summarize are given in sections 2.4.1 and 3.2.

2.3.2 Inputting data already in Stata format

Data in the Stata format are stored with the .dta extension, e.g., mydata.dta. Then the data can be read in with the use command. For example,

```
. * Read in existing Stata dataset
. use c:\research\mydata.dta, clear
```

The clear option removes any data currently in memory, even if the current data have not been saved, enabling the new file to be read in to memory.

If Stata is initiated from the current directory, then we can more simply type

```
* Read in dataset in current directory
use mydata.dta, clear
```

The use command also works over the Internet, provided that your computer is connected. For example, you can obtain an extract from the 1980 U.S. Census by typing

```
. * Read in dataset from an Internet web site
. use http://www.stata-press.com/data/r10/census.dta, clear
(1980 Census data by state)
. clear
```

2.3.3 Inputting data from the keyboard

The `input` command enables data to be typed in from the keyboard. It assumes that data are numeric. If instead data are character, then `input` should additionally define the data as a string and give the string length. For example,

```
* Data input from keyboard
input str20 name age female income
                     name        age      female      income
1.   "Barry" 25 0 40.990
2.   "Carrie" 30 1 37.000
3.   "Gary" 31 0 48.000
4. end
```

The quotes here are not necessary; we could use `Barry` rather than `"Barry"`. If the name includes a space, such as `"Barry Jr"`, then double quotes are needed; otherwise, `Barry` would be read as a string, and then `Jr` would be read as a number, leading to a program error.

To check that the data are read in correctly, we use the `list` command. Here we add the `clean` option, which lists the data without divider and separator lines.

```
list, clean
         name   age   female   income
1.      Barry    25        0    40.99
2.     Carrie    30        1       37
3.       Gary    31        0       48
```

In interactive mode, you can instead use the Data Editor to type in data (and to edit existing data).

2.3.4 Inputting nontext data

By nontext data, we mean data that are stored in the internal code of a software package other than Stata. It is easy to establish whether a file is a nontext file by viewing the file using a text editor. If strange characters appear, then the file is a nontext file. An example is an Excel `.xls` file.

Stata supports several special formats. The `fdause` command reads SAS XPORT Transport format files; the `haver` command reads Haver Analytics database files; the `odbc` command reads Open Database Connectivity (ODBC) data files; and the `xmluse` command reads XML files.

Other formats such as an Excel `.xls` file cannot be read by Stata. One solution is to use the software that created the data to write the data out into one of the readable text format files discussed below, such as a comma-separated values text file. For example, just save an Excel worksheet as a `.csv` file. A second solution is to purchase software such as Stat/Transfer that will change data from one format to another. For conversion programs, see http://www.ats.ucla.edu/stat/Stata/faq/convert_pkg.htm.

2.3.5 Inputting text data from a spreadsheet

The `insheet` command reads data that are saved by a spreadsheet or database program as comma-separated or tab-separated text data. For example, `mus02file1.csv`, a file with comma-separated values, has the following data:

```
name,age,female,income
Barry,25,0,40.990
Carrie,30,1,37.000
Gary,31,0,48.000
```

To read these data, we use `insheet`. Thus

```
. * Read data from a csv file that includes variable names using insheet
. clear
. insheet using mus02file1.csv
(4 vars, 3 obs)
. list, clean

         name   age   female   income
  1.    Barry    25        0    40.99
  2.   Carrie    30        1       37
  3.     Gary    31        0       48
```

Stata automatically recognized the `name` variable to be a string variable, the `age` and `female` variables to be integer, and the `income` variable to be floating point.

A major advantage of `insheet` is that it can read in a text file that includes variable names as well as data, making mistakes less likely. There are some limitations, however. The `insheet` command is restricted to files with a single observation per line. And the data must be comma-separated or tab-separated, but not both. It cannot be space-separated, but other delimiters can be specified by using the `delimiter` option.

The first line with variable names is optional. Let `mus02file2.csv` be the same as the original file, except without the header line:

```
Barry,25,0,40.990
Carrie,30,1,37.000
Gary,31,0,48.000
```

The insheet command still works. By default, the variables read in are given the names v1, v2, v3, and v4. Alternatively, you can assign more meaningful names in insheet. For example,

```
. * Read data from a csv file without variable names and assign names
. clear
. insheet name age female income using mus02file2.csv
(4 vars, 3 obs)
```

2.3.6 Inputting text data in free format

The infile command reads free-format text data that are space-separated, tab-separated, or comma-separated.

We again consider mus02file2.csv, which has no header line. Then

```
. * Read data from free-format text file using infile
. clear
. infile str20 name age female income using mus02file2.csv
(3 observations read)
. list, clean

          name   age   female   income
  1.     Barry    25        0    40.99
  2.   Carrie     30        1       37
  3.      Gary    31        0       48
```

By default, infile reads in all data as numbers that are stored as floating point. This causes obvious problems if the original data are string. By inserting str20 before name, the first variable is instead a string that is stored as a string of at most 20 characters.

For infile, a single observation is allowed to span more than one line, or there can be more than one observation per line. Essentially every fourth entry after Barry will be read as a string entry for name, every fourth entry after 25 will be read as a numeric entry for age, and so on.

The infile command is the most flexible command to read in data and will also read in fixed-format data.

2.3.7 Inputting text data in fixed format

The infix command reads fixed-format text data that are in fixed-column format. For example, suppose mus02file3.txt contains the same data as before, except without the header line and with the following fixed format:

```
Barry     250 40.990
Carrie    301 37.000
Gary      310 48.000
```

Here columns 1–10 store the name variable, columns 11–12 store the age variable, column 13 stores the female variable, and columns 14–20 store the income variable.

Note that a special feature of fixed-format data is that there need be no separator between data entries. For example, for the first observation, the sequence 250 is not age of 250 but is instead two variables: age = 25 and female = 0. It is easy to make errors when reading fixed-format data.

To use infix, we need to define the columns in which each entry appears. There are a number of ways to do this. For example,

```
. * Read data from fixed-format text file using infix
. clear
. infix str20 name 1-10 age 11-12 female 13 income 14-20 using mus02file3.txt
(3 observations read)
. list, clean

          name   age   female   income
  1.     Barry    25        0    40.99
  2.   Carrie     30        1       37
  3.      Gary    31        0       48
```

Similarly to infile, we include str20 to indicate that name is a string rather than a number.

A single observation can appear on more than one line. Then we use the symbol / to skip a line or use the entry 2:, for example, to switch to line 2. For example, suppose mus02file4.txt is the same as mus02file3.txt, except that income appears on a separate second line for each observation in columns 1–7. Then

```
. * Read data using infix where an observation spans more than one line
. clear
. infix str20 name 1-10 age 11-12 female 13 2: income 1-7 using mus02file4.txt
(3 observations read)
```

2.3.8 Dictionary files

For more complicated text datasets, the format for the data being read in can be stored in a dictionary file, a text file created by a word processor, or editor. Details are provided in [D] **infile (fixed format)**. Suppose this file is called mus02dict.dct. Then we simply type

```
. * Read in data with dictionary file
. infile using mus02dict
```

where the dictionary file mus02dict.dct provides variable names and formats as well as the name of the file containing the data.

2.3.9 Common pitfalls

It can be surprisingly difficult to read in data. With fixed-format data, wrong column alignment leads to errors. Data can unexpectedly include string data, perhaps with embedded blanks. Missing values might be coded as NA, causing problems if a numeric value is expected. An observation can span several lines when a single line was erroneously assumed.

It is possible to read a dataset into Stata without Stata issuing an error message; no error message does not mean that the dataset has been successfully read in. For example, transferring data from one computer type to another, such as a file transfer using File Transfer Protocol (FTP), can lead to an additional carriage return, or *Enter*, being typed at the end of each line. Then `infix` reads the dataset as containing one line of data, followed by a blank line, then another line of data, and so on. The blank lines generate extraneous observations with missing values.

You should always perform checks, such as using `list` and `summarize`. Always view the data before beginning analysis.

2.4 Data management

Once the data are read in, there can be considerable work in cleaning up the data, transforming variables, and selecting the final sample. All data-management tasks should be recorded, dated, and saved. The existence of such a record makes it easier to track changes in definitions and eases the task of replication. By far, the easiest way to do this is to have the data-management manipulations stored in a do-file rather than to use commands interactively. We assume that a do-file is used.

2.4.1 PSID example

Data management is best illustrated using a real-data example. Typically, one needs to download the entire original dataset and an accompanying document describing the dataset. For some major commonly used datasets, however, there may be cleaned-up versions of the dataset, simple data extraction tools, or both.

Here we obtain a very small extract from the 1992 Individual-Level data from the Panel Study of Income Dynamics (PSID), a U.S. longitudinal survey conducted by the University of Michigan. The extract was downloaded from the Data Center at the web site http://psidonline.isr.umich.edu/, using interactive tools to select just a few variables. The extracted sample was restricted to men aged 30–50 years. The output conveniently included a Stata do-file in addition to the text data file. Additionally, a codebook describing the variables selected was provided. The data download included several additional variables that enable unique identifiers and provide sample weights. These should also be included in the final dataset but, for brevity, have been omitted below.

Reading the text dataset `mus02psid92m.txt` using a text editor reveals that the first two observations are

```
4^ 3^ 1^ 2^ 1^ 2482^ 1^ 10^ 40^ 9^ 22000^ 2340
4^ 170^ 1^ 2^ 1^ 6974^ 1^ 10^ 37^ 12^ 31468^ 2008
```

The data are text data delimited by the symbol ^.

Several methods could be used to read the data, but the simplest is to use `insheet`. This is especially simple here given the provided do-file. The `mus02psid92m.do` file contains the following information:

```
. * Commands to read in data from PSID extract
. type mus02psid92m.do
* mus02psid92m.do
clear
#delimit ;
*   PSID DATA CENTER *****************************************************
    JOBID             : 10654
    DATA_DOMAIN       : PSID
    USER_WHERE        : ER32000=1 and ER30736 ge 30 and ER
    FILE_TYPE         : All Individuals Data
    OUTPUT_DATA_TYPE  : ASCII Data File
    STATEMENTS        : STATA Statements
    CODEBOOK_TYPE     : PDF
    N_OF_VARIABLES    : 12
    N_OF_OBSERVATIONS: 4290
    MAX_REC_LENGTH    : 56
    DATE & TIME       : November 3, 2003 @ 0:28:35
*************************************************************************
;
insheet
   ER30001 ER30002 ER32000 ER32022 ER32049 ER30733 ER30734 ER30735 ER30736
    ER30748 ER30750 ER30754
using mus02psid92m.txt, delim("^") clear
;
destring, replace ;
label variable er30001  "1968 INTERVIEW NUMBER"  ;
label variable er30002  "PERSON NUMBER                68"  ;
label variable er32000  "SEX OF INDIVIDUAL"  ;
label variable er32022  "# LIVE BIRTHS TO THIS INDIVIDUAL"  ;
label variable er32049  "LAST KNOWN MARITAL STATUS"  ;
label variable er30733  "1992 INTERVIEW NUMBER"  ;
label variable er30734  "SEQUENCE NUMBER              92"  ;
label variable er30735  "RELATION TO HEAD             92"  ;
label variable er30736  "AGE OF INDIVIDUAL            92"  ;
label variable er30748  "COMPLETED EDUCATION          92"  ;
label variable er30750  "TOT LABOR INCOME             92"  ;
label variable er30754  "ANN WORK HRS                 92"  ;
#delimit cr;    //  Change delimiter to default cr
```

To read the data, only `insheet` is essential. The code separates commands using the delimiter `;` rather than the default `cr` (the *Enter* key or carriage return) to enable comments and commands that span several lines. The `destring` command, unnecessary here, converts any string data into numeric data. For example, $1,234 would become 1234. The `label variable` command provides a longer description of the data that will be reproduced by using `describe`.

Executing this code yields output that includes the following:

```
(12 vars, 4290 obs)
. destring, replace ;
er30001 already numeric; no replace
  (output omitted)
er30754 already numeric; no replace
```

The statement `already numeric` is output for all variables because all the data in `mus02psid92m.txt` are numeric.

The `describe` command provides a description of the data:

```
. * Data description
. describe

Contains data
  obs:         4,290
 vars:            12
 size:        98,670 (99.1% of memory free)
--------------------------------------------------------------------------------
              storage  display     value
variable name   type   format      label      variable label
--------------------------------------------------------------------------------
er30001         int    %8.0g                  1968 INTERVIEW NUMBER
er30002         int    %8.0g                  PERSON NUMBER 68
er32000         byte   %8.0g                  SEX OF INDIVIDUAL
er32022         byte   %8.0g                  # LIVE BIRTHS TO THIS INDIVIDUAL
er32049         byte   %8.0g                  LAST KNOWN MARITAL STATUS
er30733         int    %8.0g                  1992 INTERVIEW NUMBER
er30734         byte   %8.0g                  SEQUENCE NUMBER 92
er30735         byte   %8.0g                  RELATION TO HEAD 92
er30736         byte   %8.0g                  AGE OF INDIVIDUAL 92
er30748         byte   %8.0g                  COMPLETED EDUCATION 92
er30750         long   %12.0g                 TOT LABOR INCOME 92
er30754         int    %8.0g                  ANN WORK HRS 92
--------------------------------------------------------------------------------
Sorted by:
     Note:  dataset has changed since last saved
```

The `summarize` command provides descriptive statistics:

```
. * Data summary
. summarize

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
     er30001 |      4290      4559.2    2850.509          4       9308
     er30002 |      4290    60.66247    79.93979          1        227
     er32000 |      4290           1           0          1          1
     er32022 |      4290    21.35385    38.20765          1         99
     er32049 |      4290    1.699534    1.391921          1          9
-------------+--------------------------------------------------------
     er30733 |      4290    4911.015      2804.8          1       9829
     er30734 |      4290    3.179487     11.4933          1         81
     er30735 |      4290    13.33147    12.44482         10         98
     er30736 |      4290    38.37995    5.650311         30         50
     er30748 |      4290    14.87249    15.07546          0         99
-------------+--------------------------------------------------------
     er30750 |      4290    27832.68    31927.35          0     999999
     er30754 |      4290    1929.477    899.5496          0       5840
```

Satisfied that the original data have been read in carefully, we proceed with cleaning the data.

2.4.2 Naming and labeling variables

The first step is to give more meaningful names to variables by using the rename command. We do so just for the variables used in subsequent analysis.

```
* Rename variables
rename er32000 sex
rename er30736 age
rename er30748 education
rename er30750 earnings
rename er30754 hours
```

The renamed variables retain the descriptions that they were originally given. Some of these descriptions are unnecessarily long, so we use label variable to shorten output from commands, such as describe, that give the variable labels.

```
* Relabel some of the variables
label variable age "AGE OF INDIVIDUAL"
label variable education "COMPLETED EDUCATION"
label variable earnings "TOT LABOR INCOME"
label variable hours "ANN WORK HRS"
```

For categorical variables, it can be useful to explain the meanings of the variables. For example, from the codebook discussed in section 2.4.4, the er32000 variable takes on the value 1 if male and 2 if female. We may prefer that the output of variable values uses a label in place of the number. These labels are provided by using label define together with label values.

```
* Define the label gender for the values taken by variable sex
label define gender 1 male 2 female
label values sex gender
list sex in 1/2, clean

       sex
  1.  male
  2.  male
```

After renaming, we obtain

```
* Data summary of key variables after renaming
summarize sex age education earnings hours

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         sex |      4290           1           0          1          1
         age |      4290    38.37995    5.650311         30         50
   education |      4290    14.87249    15.07546          0         99
    earnings |      4290    27832.68    31927.35          0     999999
       hours |      4290    1929.477    899.5496          0       5840
```

Data exist for these variables for all 4,290 sample observations. The data have $30 \leq$ age ≤ 50 and sex = 1 (male) for all observations, as expected. The maximum value for earnings is $999,999, an unusual value that most likely indicates top-coding. The maximum value of hours is quite high and may also indicate top-coding ($365 \times 16 = 5840$). The maximum value of 99 for education is clearly erroneous; the most likely explanation is that this is a missing-value code, because numbers such as 99 or −99 are often used to denote a missing value.

2.4.3 Viewing data

The standard commands for viewing data are `summarize`, `list`, and `tabulate`.

We have already illustrated the `summarize` command. Additional statistics, including key percentiles and the five largest and smallest observations, can be obtained by using the `detail` option; see section 3.2.4.

The `list` command can list every observation, too many in practice. But you could list just a few observations:

```
. * List first 2 observations of two of the variables
. list age hours in 1/2, clean

      age   hours
  1.   40    2340
  2.   37    2008
```

The `list` command with no variable list provided will list all the variables. The `clean` option eliminates dividers and separators.

The `tabulate` command lists each distinct value of the data and the number of times it occurs. It is useful for data that do not have too many distinctive values. For `education`, we have

```
. * Tabulate all values taken by a single variable
. tabulate education

  COMPLETED |
  EDUCATION |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |         82        1.91        1.91
          1 |          7        0.16        2.07
          2 |         20        0.47        2.54
          3 |         32        0.75        3.29
          4 |         26        0.61        3.89
          5 |         30        0.70        4.59
          6 |        123        2.87        7.46
          7 |         35        0.82        8.28
          8 |         78        1.82       10.09
          9 |        117        2.73       12.82
         10 |        167        3.89       16.71
         11 |        217        5.06       21.77
         12 |      1,510       35.20       56.97
         13 |        263        6.13       63.10
         14 |        432       10.07       73.17
         15 |        172        4.01       77.18
         16 |        535       12.47       89.65
         17 |        317        7.39       97.04
         99 |        127        2.96      100.00
------------+-----------------------------------
      Total |      4,290      100.00
```

Note that the variable label rather than the variable name is used as a header. The values are generally plausible, with 35% of the sample having a highest grade completed of exactly 12 years (high school graduate). The 7% of observations with 17 years most likely indicates a postgraduate degree (a college degree is only 16 years). The value 99 for 3% of the sample most likely is a missing-data code. Surprisingly, 2% appear to have completed no years of schooling. As we explain next, these are also observations with missing data.

2.4.4 Using original documentation

At this stage, it is really necessary to go to the original documentation.

The mus02psid92mcb.pdf file, generated as part of the data extraction from the PSID web site, states that for the er30748 variable a value of 0 means "inappropriate" for various reasons given in the codebook; the values 1–16 are the highest grade or year of school completed; 17 is at least some graduate work; and 99 denotes not applicable (NA) or did not know (DK).

Clearly, the education values of both 0 and 99 denote missing values. Without using the codebook, we may have misinterpreted the value of 0 as meaning zero years of schooling.

2.4.5 Missing values

It is best at this stage to flag missing values and to keep all observations rather than to immediately drop observations with missing data. In later analysis, only those observations with data missing on variables essential to the analysis need to be dropped. The characteristics of individuals with missing data can be compared with those having complete data. Data with a missing value are recoded with a missing-value code.

For education, the missing-data values 0 or 99 are replaced by . (a period), which is the default Stata missing-value code. Rather than create a new variable, we modify the current variable by using replace, as follows:

```
. * Replace missing values with missing-data code
. replace education = . if education == 0 | education == 99
(209 real changes made, 209 to missing)
```

Using the double equality and the symbol | for the logical operator or is detailed in section 1.3.5. As an example of the results, we list observations 46–48:

```
  * Listing of variable including missing value
. list education in 46/48, clean

        educat~n
 46.          12
 47.           .
 48.          16
```

Evidently, the original data on education for the 47th observation equaled 0 or 99. This has been changed to missing.

Subsequent commands using the education variable will drop observations with missing values. For example,

```
. * Example of data analysis with some missing values
. summarize education age

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
   education |      4081     12.5533    2.963696          1         17
         age |      4290    38.37995    5.650311         30         50
```

For education, only the 4,081 nonmissing values are used, whereas for age, all 4,290 of the original observations are available.

If desired, you can use more than one missing-value code. This can be useful if you want to keep track of reasons why a variable is missing. The extended missing codes are .a, .b, ..., .z. For example, we could instead have typed

```
. * Assign more than one missing code
. replace education = .a if education == 0
. replace education = .b if education == 99
```

When we want to apply multiple missing codes to a variable, it is more convenient to use the mvdecode command, which is similar to the recode command (discussed in section 2.4.7), which changes variable values or ranges of values into missing-value codes. The reverse command, mvencode, changes missing values to numeric values.

Care is needed once missing values are used. In particular, missing values are treated as large numbers, higher than any other number. The ordering is that all numbers are less than ., which is less than .a, and so on. The command

```
. * This command will include missing values
. list education in 40/60 if education > 16, clean

       educat~n
 45.         17
 47.          .
 60.         17
```

lists the missing value for observation 47 in addition to the two values of 17. If this is not desired, we should instead use

```
. * This command will not include missing values
. list education in 40/60 if education > 16 & education < . , clean

       educat~n
 45.         17
 60.         17
```

Now observation 47 with the missing observation has been excluded.

The issue of missing values also arises for earnings and hours. From the codebook, we see that a zero value may mean missing for various reasons, or it may be a true zero if the person did not work. True zeros are indicated by er30749=0 or 2, but we did not extract this variable. For such reasons, it is not unusual to have to extract data several times. Rather than extract this additional variable, as a shortcut we note that earnings and hours are missing for the same reasons that education is missing. Thus

```
. * Replace missing values with missing-data code
. replace earnings = . if education >= .
(209 real changes made, 209 to missing)

. replace hours = . if education >= .
(209 real changes made, 209 to missing)
```

2.4.6 Imputing missing data

The standard approach in microeconometrics is to drop observations with missing values, called listwise deletion. The loss of observations generally leads to less precise estimation and inference. More importantly, it may lead to sample-selection bias in regression if the retained observations have unrepresentative values of the dependent variable conditional on regressors.

An alternative to dropping observations is to impute missing values. The impute command uses predictions from regression to impute. The ipolate command uses interpolation methods. We do not cover these commands because these imputation methods have limitations, and the norm in microeconometrics studies is to use only the original data.

A more promising approach, though one more advanced, is multiple imputation. This produces M different imputed datasets (e.g., $M = 20$), fits the model M times, and performs inference that allows for the uncertainty in both estimation and data imputation. For implementation, see the user-written ice and hotdeck commands. You can find more information in Cameron and Trivedi (2005) and from findit multiple imputation.

2.4.7 Transforming data (generate, replace, egen, recode)

After handling missing values, we have the following for the key variables:

```
* Summarize cleaned up data
summarize sex age education earnings

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         sex |      4290           1           0          1          1
         age |      4290    38.37995    5.650311         30         50
   education |      4081     12.5533    2.963696          1         17
    earnings |      4081    28706.65    32279.12          0     999999
```

We now turn to recoding existing variables and creating new variables. The basic commands are generate and replace. It can be more convenient, however, to use the additional commands recode, egen, and tabulate. These are often used in conjunction with the if qualifier and the by: prefix. We present many examples throughout the book.

The generate and replace commands

The `generate` command is used to create new variables, often using standard mathematical functions. The syntax of the command is

generate [*type*] *newvar* = *exp* [*if*] [*in*]

where for numeric data the default type is `float`, but this can be changed, for example, to `double`.

It is good practice to assign a unique identifier to each observation if one does not already exist. A natural choice is to use the current observation number stored as the system variable `_n`.

```
. * Create identifier using generate command
. generate id = _n
```

We use this identifier for simplicity, though for these data the `er30001` and `er30002` variables when combined provide a unique PSID identifier.

The following command creates a new variable for the natural logarithm of earnings:

```
. * Create new variable using generate command
. generate lnearns = ln(earnings)
(498 missing values generated)
```

Missing values for `ln(earnings)` are generated whenever `earnings` data are missing. Additionally, missing values arise when `earnings` ≤ 0 because it is then not possible to take on the logarithm.

The `replace` command is used to replace some or all values of an existing variable. We already illustrated this when we created missing-values codes.

The egen command

The `egen` command is an extension to `generate` that enables creation of variables that would be difficult to create using `generate`. For example, suppose we want to create a variable that for each observation equals sample average earnings provided that sample earnings are nonmissing. The command

```
. * Create new variable using egen command
. egen aveearnings = mean(earnings) if earnings < .
(209 missing values generated)
```

creates a variable equal to the average of earnings for those observations not missing data on `earnings`.

The recode command

The recode command is an extension to replace that recodes categorical variables and generates a new variable if the generate() option is used. The command

```
. * Replace existing data using the recode command
. recode education (1/11=1) (12=2) (13/15=3) (16/17=4), generate(edcat)
(4074 differences between education and edcat)
```

creates a new variable, edcat, that takes on a value of 1, 2, 3, or 4 corresponding to, respectively, less than high school graduate, high school graduate, some college, and college graduate or higher. The edcat variable is set to missing if education does not lie in any of the ranges given in the recode command.

The by prefix

The by *varlist*: prefix repeats a command for each group of observations for which the variables in *varlist* are the same. The data must first be sorted by *varlist*. This can be done by using the sort command, which orders the observations in ascending order according to the variable(s) given in the command.

The sort command and the by prefix are more compactly combined into the bysort prefix. For example, suppose we want to create for each individual a variable that equals the sample average earnings for all persons with that individual's years of education. Then we type

```
. * Create new variable using bysort: prefix
. bysort education: egen aveearnsbyed = mean(earnings)
(209 missing values generated)
. sort id
```

The final command, one that returns the ordering of the observation to the original ordering, is not required. But it could make a difference in subsequent analysis if, for example, we were to work with a subsample of the first 1,000 observations.

Indicator variables

Consider creating a variable indicating whether earnings are positive. While there are several ways to proceed, we only describe our recommended method.

The most direct way is to use generate with logical operators:

```
. * Create indicator variable using generate command with logical operators
. generate d1 = earnings > 0 if earnings < .
(209 missing values generated)
```

The expression d1 = earnings > 0 creates an indicator variable equal to 1 if the condition holds and 0 otherwise. Because missing values are treated as large numbers, we add the condition if earnings < . so that in those cases d1 is set equal to missing.

Using summarize,

```
summarize d1
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
          d1 |      4081     .929184    .2565486          C          1
```

we can see that about 93% of the individuals in this sample had some earnings in 1992. We can also see that we have 0.929184 × 4081 = 3792 observations with a value of 1, 289 observations with a value of 0, and 209 missing observations.

Set of indicator variables

A complete set of mutually exclusive categorical indicator dummy variables can be created in several ways.

For example, suppose we want to create mutually exclusive indicator variables for less than high school graduate, high school graduate, some college, and college graduate or more. The starting point is the edcat variable, created earlier, which takes on the values 1–4.

We can use tabulate with the generate() option.

```
. * Create a set of indicator variables using tabulate with generate() option
. quietly tabulate edcat, generate(eddummy)
. summarize eddummy*
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
    eddummy1 |      4081    .2087724    .4064812          0          1
    eddummy2 |      4081    .3700074    .4828655          0          1
    eddummy3 |      4081    .2124479    .4090902          0          1
    eddummy4 |      4081    .2087724    .4064812          0          1
```

The four means sum to one, as expected for four mutually exclusive categories. Note that if edcat had taken on values 4, 5, 7, and 9, rather than 1–4, it would still generate variables numbered eddummy1–eddummy4.

An alternative method is to use the xi command. For example,

```
. * Create a set of indicator variables using command xi
. xi i.edcat, noomit
. summarize _I*
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
   _Iedcat_1 |      4081    .2087724    .4064812          0          1
   _Iedcat_2 |      4081    .3700074    .4828655          0          1
   _Iedcat_3 |      4081    .2124479    .4090902          0          1
   _Iedcat_4 |      4081    .2087724    .4064812          0          1
```

The created categorical variables are given the name edcat with the prefix _I. The suffix numbering corresponds exactly to the distinct values taken by edcat, here 1–4. The noomit option is added because the default is to omit the lowest value category, so here _Iedcat_1 would have been dropped. The prefix option allows a prefix other than _I to be specified. This is necessary if xi will be used again.

More often, xi is used as a prefix to a command, in which case the variable list includes i.*varname*, where *varname* is a categorical variable that is to appear as a set of categorical indicators. For example,

```
. * Command with a variable list that includes indicators created using xi:
. xi: summarize i.edcat
i.edcat          _Iedcat_1-4         (naturally coded; _Iedcat_1 omitted)
    Variable         Obs        Mean    Std. Dev.       Min        Max

   _Iedcat_2        4081    .3700074    .4828655          0          1
   _Iedcat_3        4081    .2124479    .4090902          0          1
   _Iedcat_4        4081    .2087724    .4064812          0          1
```

This is especially convenient in regression commands. We can simply include i.edcat in the regressor list, so there is no need to first create the set of indicator variables; see chapter 8.5.4 for an example.

Interactions

Interactive variables can be created in the obvious manner. For example, to create an interaction between the binary earnings indicator d1 and the continuous variable education, type

```
. * Create interactive variable using generate commands
. generate d1education = d1*education
(209 missing values generated)
```

It can be much simpler to use the xi command, especially if the categorical variable takes on more than two values. For example, we can generate a complete set of interactions between the categorical variable edcat (with four categories) and earnings (continuous) by typing

```
. * Create set of interactions between cat variable and set of indicators
. drop _Iedcat_*
. xi i.edcat*earnings, noomit
i.edcat*earni~s   _IedcXearni_#       (coded as above)
. summarize _I*
    Variable         Obs        Mean    Std. Dev.       Min        Max

   _Iedcat_1        4081    .2087724    .4064812          0          1
   _Iedcat_2        4081    .3700074    .4828655          0          1
   _Iedcat_3        4081    .2124479    .4090902          0          1
   _Iedcat_4        4081    .2087724    .4064812          0          1
 _IedcXearn~1       4081    3146.368    8286.325          0      80000

 _IedcXearn~2       4081    8757.823    15710.76          0     215000
 _IedcXearn~3       4081    6419.347    16453.14          0     270000
 _IedcXearn~4       4081    10383.11    32316.32          0     999999
```

Another example is to interact a categorical variable with another set of indicators. For example, to interact variable d1 with edcat, type

```
  * Create a set of interactions between a categorical and a set of indicators
  drop _I*
  xi i.edcat*i.d1, noomit
i.edcat*i.d1       IedcXd1_#_#       (coded as above)
  summarize _I*
    Variable     Obs        Mean    Std. Dev.       Min        Max

    Iedcat_1     4081    .2087724    .4064812         0          1
   _Iedcat_2     4081    .3700074    .4828655         0          1
    Iedcat_3     4081    .2124479    .4090902         0          1
    Iedcat_4     4081    .2087724    .4064812         0          1
       Id1_0     4081     .070816    .2565486         0          1

       Id1_1     4081     .929184    .2565486         0          1
  IedcXd1 1 0 _ _   4081    .0316099    .1749806         0          1
  IedcXd1 1 1 _ _   4081    .1771625    .3818529         0          1
 _IedcXd1_2_0    4081    .0279343    .1648049         0          1
  IedcXd1 2 1 _ _   4081     .342073     .474462         0          1

  IedcXd1 3 0 _ _   4081    .0098015    .0985283         0          1
  IedcXd1 3 1 _ _   4081    .2026464    .4020205         0          1
 _IedcXd1 4 0 _ _   4081    .0014702      .03832         0          1
 _IedcXd1_4_1    4081    .2073021    .4054235         0          1
```

Again this is especially convenient in regression commands because it can obviate the need to first create the set of interactions.

Demeaning

Suppose we want to include a quadratic in age as a regressor. The marginal effect of age is much easier to interpret if we use the demeaned variables $(\text{age}-\overline{\text{age}})$ and $(\text{age}-\overline{\text{age}})^2$ as regressors.

```
  * Create demeaned variables
  egen double aveage = mean(age)
  generate double agedemean = age - aveage
  generate double agesqdemean = agedemean^2
  summarize agedemean agesqdemean
    Variable     Obs        Mean    Std. Dev.       Min        Max

   agedemean     4290    2.32e-15    5.650311  -8.379953   11.62005
 agesqdemean     4290    31.91857    32.53392   .1443646   135.0255
```

We expect the agedemean variable to have an average of zero. We specified double to obtain additional precision in the floating-point calculations. In the case at hand, the mean of agedemean is on the order of 10^{-15} instead of 10^{-6}, which is what single-precision calculations would yield.

2.4.8 Saving data

At this stage, the dataset may be ready for saving. The save command creates a Stata data file. For example,

```
. * Save as Stata data file
. save mus02psid92m.dta, replace
file mus02psid92m.dta saved
```

The replace option means that an existing dataset with the same name, if it exists, will be overwritten. The .dta extension is unnecessary because it is the default extension.

The related command saveold saves a data file that can be read by versions 8 and 9 of Stata.

The data can also be saved in another format that can be read by programs other than Stata. The outsheet command allows saving as a text file in a spreadsheet format. For example,

```
. * Save as comma-separated values spreadsheet
. outsheet age education eddummy* earnings d1 hours using mus02psid92m.csv,
> comma replace
```

Note the use of the wildcard * in eddummy. The outsheet command expands this to eddummy1–eddummy4 per the rules for wildcards, given in section 1.3.4. The comma option leads to a .csv file with comma-separated variable names in the first line. The first two lines in mus02psid92m.csv are then

```
age,education,eddummy1,eddummy2,eddummy3,eddummy4,earnings,d1,hours
40,9,1,0,0,0,22000,1,2340
```

A space-delimited formatted text file can also be created by using the outfile command:

```
. * Save as formatted text (ascii) file
. outfile age education eddummy* earnings d1 hours using mus02psid92m.asc,
> replace
```

The first line in mus02psid92m.asc is then

```
  40        9        1        0        0        0        22000
 1        2340
```

This file will take up a lot of space; less space is taken if the comma option is used. The format of the file can be specified using Stata's dictionary format.

2.4.9 Selecting the sample

Most commands will automatically drop missing values in implementing a given command. We may want to drop additional observations, for example, to restrict analysis to a particular age group.

This can be done by adding an appropriate if qualifier after the command. For example, if we want to summarize data for only those individuals 35–44 years old, then

```
. * Select the sample used in a single command using the if qualifier
. summarize earnings lnearns if age >= 35 & age <= 44

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
    earnings |      2114    30131.05    37660.11          0     999999
     lnearns |      1983    10.04658    .9001594   4.787492   13.81551
```

Different samples are being used here for the two variables, because for the 131 observations with zero earnings, we have data on earnings but not on lnearns. The if qualifier uses logical operators, defined in section 1.3.5.

However, for most purposes, we would want to use a consistent sample. For example, if separate earnings regressions were run in levels and in logs, we would usually want to use the same sample in the two regressions.

The drop and keep commands allow sample selection for the rest of the analysis. The keep command explicitly selects the subsample to be retained. Alternatively, we can use the drop command, in which case the subsample retained is the portion not dropped. The sample dropped or kept can be determined by using an if qualifier, a variable list, or by defining a range of observations.

For the current example, we use

```
. * Select the sample using command keep
. keep if (lnearns != .) & (age >= 35 & age <= 44)
(2307 observations deleted)

. summarize earnings lnearns

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
    earnings |      1983    32121.55    38053.31        120     999999
     lnearns |      1983    10.04658    .9001594   4.787492   13.81551
```

This command keeps the data provided: lnearns is nonmissing and $35 \leq$ age ≤ 44. Note that now earnings and lnearns are summarized for the same 1,983 observations.

As a second example, the commands

```
. * Select the sample using keep and drop commands
. use mus02psid92m.dta, clear

. keep lnearns age

. drop in 1/1000
(1000 observations deleted)
```

will lead to a sample that contains data on all but the first one thousand observations for just the two variables lnearns and age. The use mus02psid92m.dta command is added because the previous example had already dropped some of the data.

2.5 Manipulating datasets

Useful manipulations of datasets include reordering observations or variables, temporarily changing the dataset but then returning to the original dataset, breaking one observation into several observations (and vice versa), and combining more than one dataset.

2.5.1 Ordering observations and variables

Some commands, such as those using the by prefix, require sorted observations. The sort command orders observations in ascending order according to the variable(s) in the command. The gsort command allows ordering to be in descending order.

You can also reorder the variables by using the order command. This can be useful if, for example, you want to distribute a dataset to others with the most important variables appearing as the first variables in the dataset.

2.5.2 Preserving and restoring a dataset

In some cases, it is desirable to temporarily change the dataset, perform some calculation, and then return the dataset to its original form. An example involving the computation of marginal effects is presented in section 10.5.4. The preserve command preserves the data, and the restore command restores the data to the form it had immediately before preserve.

```
 * Commands preserve and restore illustrated
 use mus02psid92m.dta, clear
 list age in 1/1

      +-----+
      | age |
      |-----|
  1.  |  40 |
      +-----+

. preserve
. replace age = age + 1000
age was byte now int
(4290 real changes made)
 list age in 1/1

      +------+
      |  age |
      |------|
  1.  | 1040 |
      +------+

 restore
 list age in 1/1

      +-----+
      | age |
      |-----|
  1.  |  40 |
      +-----+
```

As desired, the data have been returned to original values.

2.5.3 Wide and long forms for a dataset

Some datasets may combine several observations into a single observation. For example, a single household observation may contain data for several household members, or a single individual observation may have data for each of several years. This format for data is called wide form. If instead these data are broken out so that an observation is for a distinct household member, or for a distinct individual–year pair, the data are said to be in long form.

The reshape command is detailed in section 8.11. It converts data from wide form to long form and vice versa. This is necessary if an estimation command requires data to be in long form, say, but the original dataset is in wide form. The distinction is important especially for analysis of panel data and multinomial data.

2.5.4 Merging datasets

The merge command combines two datasets to create a wider dataset, i.e., new variables from the second dataset are added to existing variables of the first dataset. Common examples are data on the same individuals obtained from two separate sources that then need to be combined, and data on supplementary variables or additional years of data.

Merging two datasets involves adding information from a dataset on disk to a dataset in memory. The dataset in memory is known as the master dataset.

Merging two datasets is straightforward if the datasets have the same number of observations and the merge is a line-to-line merge. Then line 10, for example, of one dataset is combined with line 10 of the other dataset to create a longer line 10. We consider instead a match-merge, where observations in the two datasets are combined if they have the same values for one or more identifying variables that are used to determine the match. In either case, when a match is made if a variable appears in both datasets, then the master dataset value is retained unless it is missing, in which case it is replaced by the value in the second dataset. If a variable exists only in the second dataset, then it is added as a variable to the master dataset.

To demonstrate a match-merge, we create two datasets from the dataset used in this chapter. The first dataset comprises every third observation with data on id, education, and earnings:

```
  * Create first dataset with every third observation
. use mus02psid92m.dta, clear
. keep if mod(_n,3) == 0
(2860 observations deleted)
. keep id education earnings
. list in 1/4, clean

       educat~n   earnings   id
  1.         16      38708    3
  2.         12       3265    6
  3.         11      19426    9
  4.         11      30000   12

. quietly save merge1.dta, replace
```

The keep if mod(_n,3) == 0 command keeps an observation if the observation number (_n) is exactly divisible by 3, so every third observation is kept. Because id=_n for these data, by saving every third observation we are saving observations with id equal to 3, 6, 9, ...

The second dataset comprises every second observation with data on id, education, and hours:

```
. * Create second dataset with every second observation
. use mus02psid92m.dta, clear
. keep if mod(_n,2) == 0
(2145 observations deleted)
. keep id education hours
. list in 1/4, clean

       educat~n   hours   id
  1.         12    2008    2
  2.         12    2200    4
  3.         12     552    6
  4.         17    3750    8

. quietly save merge2.dta, replace
```

Now we are saving observations with id equal to 2, 4, 6, ...

Now we merge the two datasets by using the merge command.

In our case, the datasets differ in both the observations included and the variables included, though there is considerable overlap. We perform a match-merge on id to obtain

```
. * Merge two datasets with some observations and variables different
. clear
. use merge1.dta
. sort id
. merge id using merge2.dta
. sort id
. list in 1/4, clean

       educat~n   earnings   id   hours   _merge
  1.         12          .    2    2008        2
  2.         16      38708    3       .        1
  3.         12          .    4    2200        2
  4.         12       3265    6     552        3
```

Recall that observations from the master dataset have id equal to 3, 6, 9, ..., and observations from the second dataset have id equal to 2, 4, 6, ... Data for education and earnings are always available because they are in the master dataset. But observations for hours come from the second dataset; they are available when id is 2, 4, 6, ... and are missing otherwise.

merge creates a variable, _merge, that takes on a value of 1 if the variables for an observation all come from the master dataset, a value of 2 if they all come from only the second dataset, and a value of 3 if for an observation some variables come from the master and some from the second dataset. After using merge, you should tabulate _merge and check that the results match your expectations. For the example, we obtain the expected results:

```
. tab _merge

     _merge |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |        715       25.00       25.00
          2 |      1,430       50.00       75.00
          3 |        715       25.00      100.00
------------+-----------------------------------
      Total |      2,860      100.00
```

There are several options when using merge. The update option varies the action merge takes when an observation is matched. By default, the master dataset is held inviolate—if update is specified, values from the master dataset are retained if the same variables are found in both datasets. However, the values from the merging dataset are used in cases where the variable is missing in the master dataset. The replace option, allowed only with the update option, specifies that even if the master dataset contains nonmissing values, they are to be replaced with corresponding values from the merging dataset when corresponding values are not equal. A nonmissing value, however, will never be replaced with a missing value.

2.5.5 Appending datasets

The append command creates a longer dataset, with the observations from the second dataset appended after all the observations from the first dataset. If the same variable has different names in the two datasets, the variable name in one of the datasets should be changed by using the rename command so that the names match.

```
* Append two datasets with some observations and variables different
clear
use merge1.dta
append using merge2.dta
sort id
list in 1/4, clean

     educat~n   earnings   id   hours
  1.       12          .    2    2008
  2.       16      38708    3       .
  3.       12          .    4    2200
  4.       12       3265    6       .
```

Now merge2.dta is appended to the end of merge1.dta. The combined dataset has observations 3, 6, 9, ..., 4290 followed by observations 2, 4, 6, ..., 4290. We then sort on id. Now both every second and every third observation is included, so after sorting we have observations 2, 3, 4, 6, 8, 9, ... Note, however, that no attempt has been made to merge the datasets. In particular, for the observation with id = 6, the hours data are missing. This is because this observation comes from the master dataset, which did not include hours as a variable, and there is no attempt to merge the data.

In this example, to take full advantage of the data, we would need to merge the two datasets using the first dataset as the master, merge the two datasets using the second dataset as the master, and then append the two datasets.

2.6 Graphical display of data

Graphs visually demonstrate important features of the data. Different types of data require distinct graph formats to bring out these features. We emphasize methods for numerical data taking many values, particularly, nonparametric methods.

2.6.1 Stata graph commands

The Stata graph commands begin with the word graph (in some cases, this is optional) followed by the graph plottype, usually twoway. We cover several leading examples but ignore the plottypes bar and pie for categorical data.

Example graph commands

The basic graph commands are very short and simple to use. For example,

```
. use mus02psid92m.dta, clear
. twoway scatter lnearns hours
```

produces a scatterplot of lnearns on hours, shown in figure 2.1. Most graph commands support the if and in qualifiers, and some support weights.

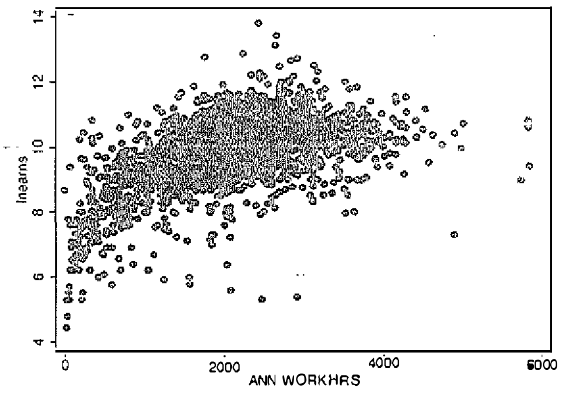

Figure 2.1. A basic scatterplot of log earnings on hours

In practice, however, customizing is often desirable. For example, we may want to display the relationship between lnearns and hours by showing both the data scatterplot and the ordinary least-squares (OLS) fitted line on the same graph. Additionally, we may want to change the size of the scatterplot data points, change the width of the regression line, and provide a title for the graph. We type

```
. * More advanced graphics command with two plots and with several options
. graph twoway (scatter lnearns hours, msize(small))
>    (lfit lnearns hours, lwidth(medthick)),
>    title("Scatterplot and OLS fitted line")
```

The two separate components scatter and lfit are specified separately within parentheses. Each of these commands is given with one option, after the comma but within the relevant parentheses. The msize(small) option makes the scatterplot dots smaller than the default, and the lwidth(medthick) option makes the OLS fitted line thicker than the default. The title() option for twoway appears after the last comma. The graph produced is shown in figure 2.2.

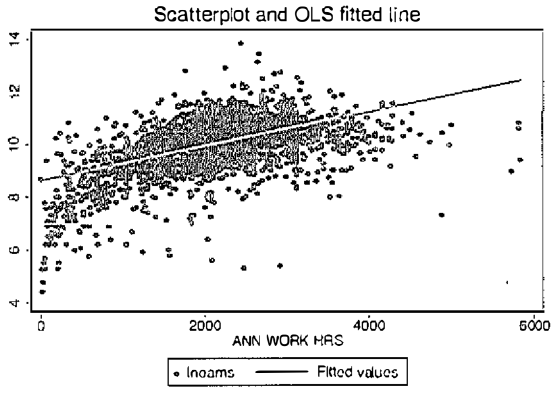

Figure 2.2. A more elaborate scatterplot of log earnings on hours

We often use lengthy graph commands that span multiple lines to produce template graphs that are better looking than those produced with default settings. In particular, these commands add titles and rescale the points, lines, and axes to a suitable size because the graphs printed in this book are printed in a much smaller space than a full-page graph in landscape mode. These templates can be modified for other applications by changing variable names and title text.

Saving and exporting graphs

Once a graph is created, it can be saved. Stata uses the term save to mean saving the graph in Stata's internal graph format, as a file with the .gph extension. This can be done by using the saving() option in a graph command or by typing graph save after the graph is created. When saved in this way, the graphs can be reaccessed and further manipulated at a later date.

Two or more Stata graphs can be combined into a single figure by using the graph combine command. For example, we save the first graph as graph1.gph, save the second graph as graph2.gph, and type the command

```
. * Combine graphs saved as graph1.gph and graph2.gph
. graph combine graph1 graph2
  (output omitted)
```

Section 3.2.7 provides an example.

The Stata internal graph format (.gph) is not recognized by other programs, such as word processors. To save a graph in an external format, you would use the graph export command. For example,

```
. * Save graph as a Windows meta-file
. graph export mygraph.wmf
  (output omitted)
```

Various formats are available, including PostScript (.ps), Encapsulated PostScript (.eps), Windows Metafile (.wmf), PDF (.pdf), and Portable Network Graphics (.png). The best format to select depends in part on what word processor is used; some trial and error may be needed.

Learning how to use graph commands

The Stata graph commands are extremely rich and provide an exceptional range of user control through a multitude of options.

A good way to learn the possibilities is to create a graph interactively in Stata. For example, from the menus, select **Graphics > Twoway graph (scatter, line, etc.)**. In the **Plots** tab of the resulting dialog box, select **Create...**, choose **Scatter**, provide a *Y variable* and an *X variable*, and then click on **Marker properties**. From the **Symbol** drop-down list, change the default to, say, **Triangle**. Similarly, cycle through the other options and change the default settings to something else.

Once an initial graph is created, the point-and-click Stata Graph Editor allows further customizing of the graph, such as adding text and arrows wherever desired. This is an exceptionally powerful tool that we do not pursue here; for a summary, see [G] **graph editor**. The Graph Recorder can even save sequences of changes to apply to similar graphs created from different samples.

Even given familiarity with Stata's graph commands, you may need to tweak a graph considerably to make it useful. For example, any graph that analyzes the earnings variable using all observations will run into problems because one observation has a large outlying value of $999,999. Possibilities in that case are to drop outliers, plot with the yscale(log) option, or use log earnings instead.

2.6.2 Box-and-whisker plot

The graph box command produces a box-and-whisker plot that is a graphical way to display data on a single series. The boxes cover the interquartile range, from the lower quartile to the upper quartile. The whiskers, denoted by horizontal lines, extend to cover most or all the range of the data. Stata places the upper whisker at the upper quartile plus 1.5 times the interquartile range, or at the maximum of the data if this is smaller. Similarly, the lower whisker is the lower quartile minus 1.5 times the interquartile range, or the minimum should this be larger. Any data values outside the whiskers are represented with dots. Box-and-whisker plots can be especially useful for identifying outliers.

The essential command for a box-and-whisker plot of the hours variable is

```
* Simple box-and-whisker plot
graph box hours
(output omitted)
```

We want to present separate box plots of hours for each of four education groups by using the over() option. To make the plot more intelligible, we first provide labels for the four education categories as follows:

```
. use mus02psid92m.dta, clear
. label define edtype 1 "< High School" 2 "High School" 3 "Some College"
> 4 "College Degree"
. label values edcat edtype
```

The scale(1.2) graph option is added for readability; it increases the size of text, markers, and line widths (by a multiple 1.2). The marker() option is added to reduce the size of quantities within the box; the ytitle() option is used to present the title; and the yscale(titlegap(*5)) option is added to increase the gap between the y-axis title and the tick labels. We have

```
. * Box and whisker plot of single variable over several categories
. graph box hours, over(edcat) scale(1.2) marker(1,msize(vsmall))
> ytitle("Annual hours worked by education") yscale(titlegap(*5))
```

The result is given in figure 2.3. The labels for edcat, rather than the values, are automatically given, making the graph much more readable. The filled-in boxes present the interquartile range, the intermediate line denotes the median, and data outside the whiskers appear as dots. For these data, annual hours are clearly lower for the lowest schooling group, and there are quite a few outliers. About 30 individuals appear to work in excess of 4,000 hours per year.

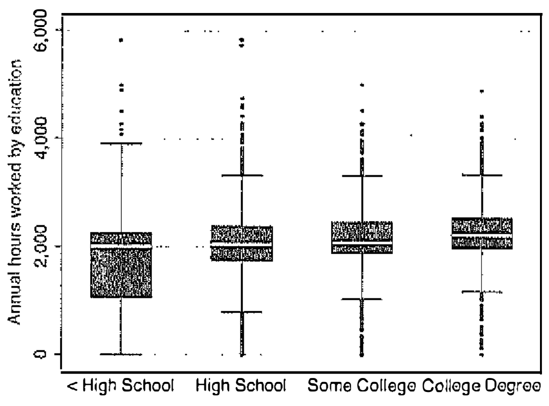

Figure 2.3. Box-and-whisker plots of annual hours for four categories of educational attainment

2.6.3 Histogram

The probability mass function or density function can be estimated using a histogram produced by the `histogram` command. The command can be used with `if` and `in` qualifiers and with weights. The key options are `width(#)` to set the bin width, `bin(#)` to set the number of bins, `start(#)` to set the lower limit of the first bin, and `discrete` to indicate that the data are discrete. The default number of bins is $\min(\sqrt{N}, 10 \ln N / \ln 10)$. Other options overlay a fitted normal density (the `normal` option) or a kernel density estimate (the `kdensity` option).

For discrete data taking relatively few values, there is usually no need to use the options.

For continuous data or for discrete data taking many values, it can be necessary to use options because the Stata defaults set bin widths that are not nicely rounded numbers and the number of bins might also not be desirable. For example, the output from `histogram lnearns` states that there are 35 bins, a bin width of 0.268, and a start value of 4.43. A better choice may be

```
. * Histogram with bin width and start value set
. histogram lnearns, width(0.25) start(4.0)
(bin=40, start=4, width=.25)
```

(Continued on next page)

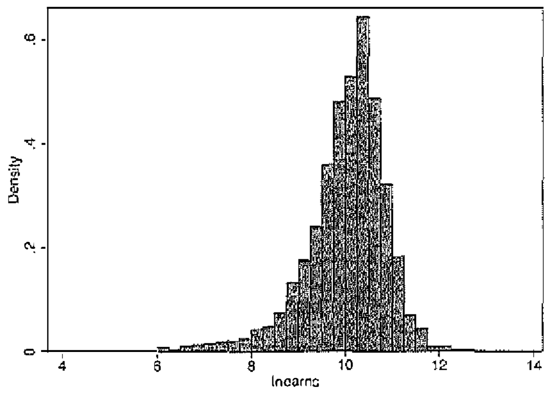

Figure 2.4. A histogram for log earnings

2.6.4 Kernel density plot

For continuous data taking many values, a better alternative to the histogram is a kernel density plot. This provides a smoother version of the histogram in two ways: First, it directly connects the midpoints of the histogram rather than forming the histogram step function. Second, rather than giving each entry in a bin equal weight, it gives more weight to data that are closest to the point of evaluation.

Let $f(x)$ denote the density. The kernel density estimate of $f(x)$ at $x = x_0$ is

$$\widehat{f}(x_0) = \frac{1}{Nh}\sum_{i=1}^{N} K\left(\frac{x_i - x_0}{h}\right) \tag{2.1}$$

where $K(\cdot)$ is a kernel function that places greater weight on points x_i close to x_0. More precisely, $K(z)$ is symmetric around zero, integrates to one, and either $K(z) = 0$ if $|z| \geq z_0$ (for some z_0) or $z \to 0$ as $z \to \infty$. A histogram with a bin width of $2h$ evaluated at x_0 can be shown to be the special case $K(z) = 1/2$ if $|z| < 1$, and $K(z) = 0$ otherwise.

A kernel density plot is obtained by choosing a kernel function, $K(\cdot)$; choosing a width, h; evaluating $\widehat{f}(x_0)$ at a range of values of x_0; and plotting $\widehat{f}(x_0)$ against these x_0 values.

The `kdensity` command produces a kernel density estimate. The command can be used with `if` and `in` qualifiers and with weights. The default kernel function is the Epanechnikov, which sets $K(z) = (3/4)(1 - z^2/5)/\sqrt{5}$ if $|z| < \sqrt{5}$, and $K(z) = 0$ otherwise. The `kernel()` option allows other kernels to be chosen, but unless the width is relatively small, the choice of kernel makes little difference. The default window width or bandwidth is $h = 0.9m/n^{1/5}$, where $m = \min(s_x,\ iqr_x/1.349)$ and iqr_x is the interquartile range of x. The `bwidth(#)` option allows a different width (h) to be specified, with larger choices of h leading to smoother density plots. The n(#) option changes the number of evaluation points, x_0, from the default of min(N, 50). Other options overlay a fitted normal density (the normal option) or a fitted t density (the student(#) option).

The output from kdensity lnearns states that the Epanechnikov kernel is used and the bandwidth equals 0.1227. If we desire a smoother density estimate with a bandwidth of 0.2, one overlaid by a fitted normal density, we type the command

```
.   * Kernel density plot with bandwidth set and fitted normal density overlaid
.   kdensity lnearns, bwidth(0.20) normal n(4000)
```

which produces the graph in figure 2.5. This graph shows that the kernel density is more peaked than the normal and is somewhat skewed.

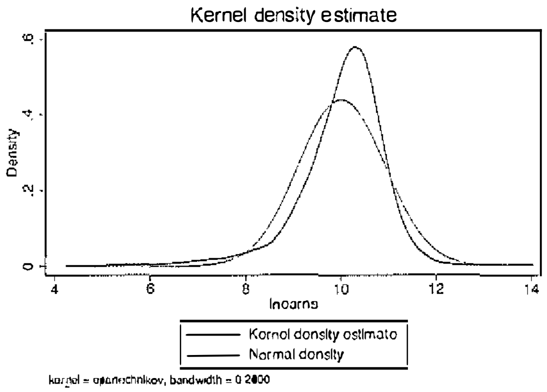

Figure 2.5. The estimated density of log earnings

The following code instead presents a histogram overlaid by a kernel density estimate. The histogram bin width is set to 0.25, the kernel density bandwidth is set to 0.2 using the kdenopts() option, and the kernel density plot line thickness is increased using the lwidth(medthick) option. Other options used here were explained in section 2.6.2. We have

```
.   * Histogram and nonparametric kernel density estimate
.   histogram lnearns if lnearns > 0, width(0.25) kdensity
>    kdenopts(bwidth(0.2) lwidth(medthick))
>    plotregion(style(none)) scale(1.2)
>    title("Histogram and density for log earnings")
>    xtitle("Log annual earnings", size(medlarge)) xscale(titlegap(*5))
>    ytitle("Histogram and density", size(medlarge)) yscale(titlegap(*5))
(bin=38, start=4.4308167, width=.25)
```

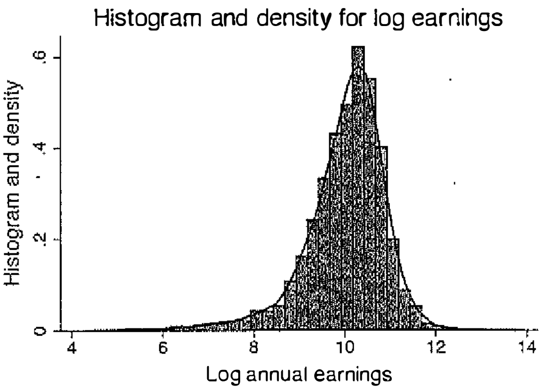

Figure 2.6. Histogram and kernel density plot for natural logarithm of earnings

The result is given in figure 2.6. Both the histogram and the kernel density estimate indicate that the natural logarithm of earnings has a density that is mildly left-skewed. A similar figure for the level of earnings is very right-skewed.

2.6.5 Twoway scatterplots and fitted lines

As we saw in figure 2.1, scatterplots provide a quick look at the relationship between two variables.

For scatterplots with discrete data that take on few values, it can be necessary to use the `jitter()` option. This option adds random noise so that points are not plotted on top of one another; see section 14.6.4 for an example.

It can be useful to additionally provide a fitted curve. Stata provides several possibilities for estimating a global relationship between y against x, where by global we mean that a single relationship is estimated for all observations, and then for plotting the fitted values of y against x.

The `twoway lfit` command does so for a fitted OLS regression line, the `twoway qfit` command does so for a fitted quadratic regression curve, and the `twoway fpfit` command does so for a curve fitted by fractional polynomial regression. The related `twoway` commands `lfitci`, `qfitci`, and `fpfitci` additionally provide confidence bands for predicting the conditional mean $E(y|x)$ (by using the `stdp` option) or for forecasting of the actual value of $y|x$ (by using the `stdf` option).

For example, we may want to provide a scatterplot and fitted quadratic with confidence bands for the forecast value of $y|x$ (the result is shown in figure 2.7):

```
. * Two-way scatterplot and quadratic regression curve with 95% ci for y|x
. twoway (qfitci lnearns hours, stdf) (scatter lnearns hours, msize(small))
```

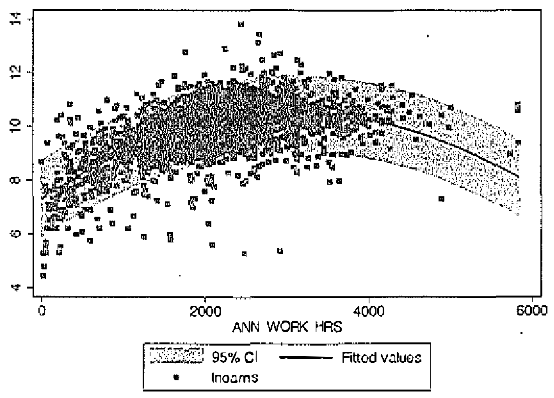

Figure 2.7. Twoway scatterplot and fitted quadratic with confidence bands

2.6.6 Lowess, kernel, local linear, and nearest-neighbor regression

An alternative curve-fitting approach is to use nonparametric methods that fit a local relationship between y and x, where by local we mean that separate fitted relationships are obtained at different values of x. There are several methods. All depend on a bandwidth parameter or smoothing parameter. There are well-established methods to automatically select the bandwidth parameter, but these choices in practice can undersmooth or oversmooth the data so that the bandwidth then needs to be set by using the `bwidth()` option.

An easily understood example is a median-band plot. The range of x is broken into, say, 20 intervals; the medians of y and x in each interval are obtained; and the 20 medians of y are plotted against the 20 medians of x, with connecting lines between the points. The `twoway mband` command does this, and the related `twoway mspline` command uses a cubic spline to obtain a smoother version of the median-band plot.

Most nonparametric methods instead use variants of local regression. Consider the regression model $y = m(x) + u$, where x is a scalar and the conditional mean function $m(\cdot)$ is not specified. A local regression estimate of $m(x)$ at $x = x_0$ is a local weighted average of y_i, $i = 1, \ldots, N$, that places great weight on observations for which x_i is close to x_0 and little or no weight on observations for which x_i is far from x_0. Formally,

$$\widehat{m}(x_0) = \sum\nolimits_{i=1}^{N} w(x_i, x_0, h) y_i$$

where the weights $w(x_i, x_0, h)$ sum over i to one and decrease as the distance between x_i and x_0 increases. As the bandwidth parameter h increases, more weight is placed on observations for which x_i is close to x_0.

A plot is obtained by choosing a weighting function, $w(x_i, x_0, h)$; choosing a bandwidth, h; evaluating $\widehat{m}(x_0)$ at a range of values of x_0; and plotting $\widehat{m}(x_0)$ against these x_0 values.

The kth-nearest-neighbor estimator uses just the k observations for which x_i is closest to x_0 and equally weights these k closest values. This estimator can be obtained by using the user-written knnreg command (Salgado-Ugarte, Shimizu, and Taniuchi 1996).

Kernel regression uses the weight $w(x_i, x_0, h) = K\{(x_i - x_0)/h\}/\sum_{i=1}^{N} K\{(x_i - x_0)/h\}$, where $K(\cdot)$ is a kernel function defined after (2.1). This estimator can be obtained by using the user-written kernreg command (Salgado-Ugarte, Shimizu, and Taniuchi 1996). It can also be obtained by using the lpoly command, which we present next.

The kernel regression estimate at $x = x_0$ can equivalently be obtained by minimizing $\sum_i K\{(x_i - x_0)/h\}(y_i - \alpha_0)^2$, which is weighted regression on a constant where the kernel weights are largest for observations with x_i close to x_0. The local linear estimator additionally includes a slope coefficient and at $x = x_0$ minimizes

$$\sum\nolimits_{i=1}^{N} K\left(\frac{x_i - x_0}{h}\right)\{y_i - \alpha_0 - \beta_0(x_i - x_0)\}^2 \tag{2.2}$$

The local polynomial estimator of degree p more generally uses a polynomial of degree p in $(x_i - x_0)$ in (2.2). This estimator is obtained by using lpoly. The degree(#) option specifies the degree p, the kernel() option specifies the kernel, the bwidth(#) option specifies the kernel bandwidth h, and the generate() option saves the evaluation points x_0 and the estimates $\widehat{m}(x_0)$. The local linear estimator with $p \geq 1$ does much better than the preceding methods at estimating $m(x_0)$ at values of x_0 near the endpoints of the range of x, as it allows for any trends near the endpoints.

The locally weighted scatterplot smoothing estimator (lowess) is a variation of the local linear estimator that uses a variable bandwidth, a tricubic kernel, and downweights observations with large residuals (using a method that greatly increases the computational burden). This estimator is obtained by using the lowess command. The bandwidth gives the fraction of the observations used to calculate $\widehat{m}(x_0)$ in the middle of the data, with a smaller fraction used towards the endpoints. The default value of 0.8 can be changed by using the bwidth(#) option, so unlike the other methods, a smoother plot is obtained by increasing the bandwidth.

The following example illustrates the relationship between log earnings and hours worked. The one graph includes a scatterplot (scatter), a fitted lowess curve (lowess), and a local linear curve (lpoly). The command is lengthy because of the detailed formatting commands used to produce a nicely labeled and formatted graph. The msize(tiny) option is used to decrease the size of the dots in the scatterplot. The lwidth(medthick) option is used to increase the thickness of lines, and the clstyle(p1) option changes the style of the line for lowess. The title() option provides the overall title for the graph. The xtitle() and ytitle() options provide titles for the x axis and y axis, and the size(medlarge) option defines the size of the text for these titles.

The legend() options place the graph legend at four o'clock (pos(4)) with text size small and provide the legend labels. We have

```
. * Scatterplot with lowess and local linear nonparametric regression
. graph twoway (scatter lnearns hours, msize(tiny))
>   (lowess lnearns hours, clstyle(p1) lwidth(medthick))
>   (lpoly lnearns hours, kernel(epan2) degree(1) lwidth(medthick)
>   bwidth(500)), plotregion(style(none))
>   title("Scatterplot, lowess, and local linear regression")
>   xtitle("Annual hours", size(medlarge))
>   ytitle("Natural logarithm of annual earnings", size(medlarge))
>   legend(pos(4) ring(0) col(1)) legend(size(small))
>   legend(label(1 "Actual Data") label(2 "Lowess") label(3 "Local linear"))
```

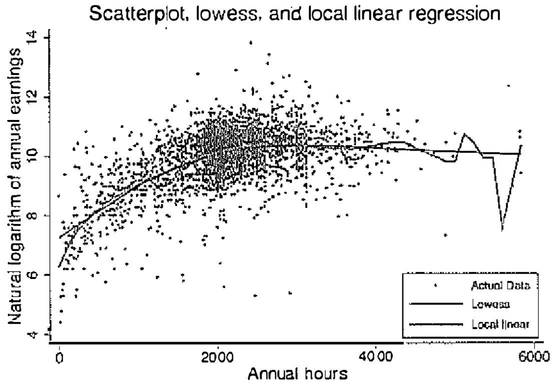

Figure 2.8. Scatterplot, lowess, and local linear curves for natural logarithm of earnings plotted against hours

From figure 2.8, the scatterplot, fitted OLS line, and nonparametric regression all indicate that log earnings increase with hours until about 2,500 hours and that a quadratic relationship may be appropriate. The graph uses the default bandwidth setting for lowess and greatly increases the lpoly bandwidth from its automatically selected value of 84.17 to 500. Even so, the local linear curve is too variable at high hours where the data are sparse. At low hours, however, the lowess estimator overpredicts while the local linear estimator does not.

2.6.7 Multiple scatterplots

The graph matrix command provides separate bivariate scatterplots between several variables. Here we produce bivariate scatterplots (shown in figure 2.9) of lnearns, hours, and age for each of the four education categories:

```
. * Multiple scatterplots
. label variable age "Age"
. label variable lnearns "Log earnings"
. label variable hours "Annual hours"
. graph matrix lnearns hours age, by(edcat) msize(small)
```

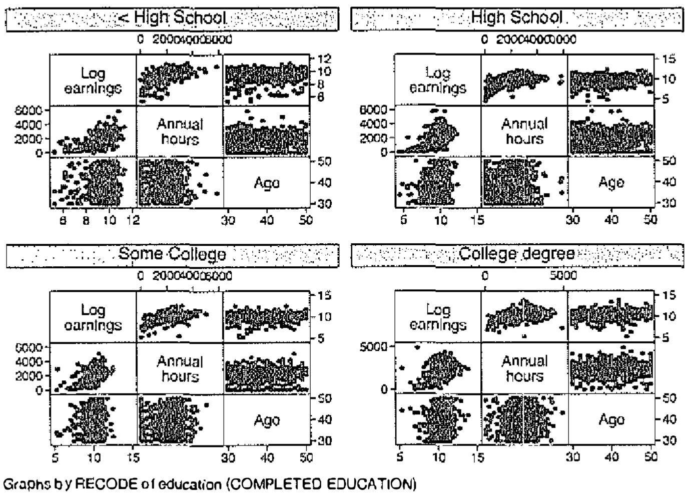

Figure 2.9. Multiple scatterplots for each level of education

Stata does not provide three-dimensional graphs, such as that for a nonparametric bivariate density estimate or for nonparametric regression of one variable on two other variables.

2.7 Stata resources

The key data-management references are [U] *Users Guide* and [D] *Data Management Reference Manual*. Useful online help categories include 1) `double`, `string`, and `format` for data types; 2) `clear`, `use`, `insheet`, `infile`, and `outsheet` for data input; 3) `summarize`, `list`, `label`, `tabulate`, `generate`, `egen`, `keep`, `drop`, `recode`, `by`, `sort`, `merge`, `append`, and `collapse` for data management; and 4) `graph`, `graph box`, `histogram`, `kdensity`, `twoway`, `lowess`, and `graph matrix` for graphical analysis.

The Stata graphics commands were greatly enhanced in version 8 and are still relatively underutilized. The Stata Graph Editor is new to version 10; see [G] **graph editor**. *A Visual Guide to Stata Graphics* by Mitchell (2008) provides many hundreds of template graphs with the underlying Stata code and an explanation for each.

2.8 Exercises

1. Type the command `display %10.5f 123.321`. Compare the results with those you obtain when you change the format `%10.5f` to, respectively, `%10.5e`, `%10.5g`, `%-10.5f`, `%10,5f`, and when you do not specify a format.

2. Consider the example of section 2.3 except with the variables reordered. Specifically, the variables are in the order age, `name`, `income`, and `female`. The three observations are 29 `"Barry" 40.990 0`; 30 `"Carrie" 37.000 1`; and 31 `"Gary" 48.000 0`. Use `input` to read these data, along with names, into Stata and list the results. Use a text editor to create a comma-separated values file that includes variable names in the first line, read this file into Stata by using `insheet`, and list the results. Then drop the first line in the text file, read in the data by using `insheet` with variable names assigned, and list the results. Finally, replace the commas in the text file with blanks, read the data in by using `infix`, and list the results.

3. Consider the dataset in section 2.4. The `er32049` variable is the last known marital status. Rename this variable as `marstatus`, give the variable the label "marital status", and tabulate `marstatus`. From the codebook, marital status is married (1), never married (2), widowed (3), divorced or annulment (4), separated (5), not answered or do not know (8), and no marital history collected (9). Set `marstatus` to missing where appropriate. Use `label define` and `label values` to provide descriptions for the remaining categories, and tabulate `marstatus`. Create a binary indicator variable equal to 1 if the last known marital status is married, and equal to 0 otherwise, with appropriate handling of any missing data. Provide a summary of earnings by marital status. Create a set of indicator variables for marital status based on `marstatus`. Create a set of variables that interact these marital status indicators with earnings.

4. Consider the dataset in section 2.6. Create a box-and-whisker plot of `earnings` (in levels) for all the data and for each year of educational attainment (use variable `education`). Create a histogram of `earnings` (in levels) using 100 bins and a kernel density estimate. Do earnings in levels appear to be right-skewed? Create a scatterplot of `earnings` against `education`. Provide a single figure that uses `scatterplot`, `lfit`, and `lowess` of `earnings` against `education`. Add titles for the axes and graph heading.

5. Consider the dataset in section 2.6. Create kernel density plots for `lnearns` using the `kernel(epan2)` option with kernel $K(z) = (3/4)(1 - z^2/5)$ for $|z| < 1$, and using the `kernel(epan2)` option with kernel $K(z) = 1/2$ for $|z| < 1$. Repeat with the bandwidth increased from the default to 0.3. What makes a bigger difference, choice of kernel or choice of bandwidth? The comparison is easier if the four graphs are saved using the `saving()` option and then combined using the `graph combine` command.

6. Consider the dataset in section 2.6. Perform lowess regression of `lnearns` on `hours` using the default bandwidth and using bandwidth of 0.01. Does the bandwidth make a difference? A moving average of y after data are sorted by x is a simple case of nonparametric regression of y on x. Sort the data by `hours`. Create a centered 15-period moving average of `lnearns` with ith observation $yma_i = 1/25 \sum_{j=-12}^{j=12} y_{i+j}$. This is easiest using `forvalues`. Plot this moving average against `hours` using the `twoway connected` graph command. Compare to the lowess plot.