Introduction to Stata

Marco Rosso[1]

This version: November 23, 2025

[1]Department of Economics, University of Bologna (email: marco.rosso4@unibo.it)

Contents

Preface **1**

1 Introduction **3**

1.1 What is Stata 3

1.2 The Reproducible Stata Workflow 5

2 Data Handling **10**

2.1 Importing External Data 10

2.2 Dataset Management 11

2.3 Memory Allocation 12

2.4 Variables and Labels 12

2.5 Generating and Modifying Variables 13

3 Essential Commands **14**

3.1 Describe 15

3.2 Display 16

3.3 List 17

3.4 Format 18

3.5 Drop and Keep 19

3.6 Classes of Operators 22

3.7 Functions 23

3.8 Sort and Gsort 25

3.9 The Power of Egen 27

3.10 Group Commands 28
3.11 Generate Dummy 29
Exercises 32
4 Descriptive Statistics 33
4.1 Count 33
4.2 Summarize 34
4.3 Means 35
4.4 Centile 37
4.5 Cumul 39
4.6 Correlate 40
4.7 Tables 42
4.8 Loops 45
Exercises 48
5 Data Manipulation 49
5.1 Append 49
5.2 Merge 50
5.3 Collapse 51
5.4 Contract 52
5.5 Expand 52
5.6 Fillin 53
Exercises 55
6 Estimation 56
6.1 OLS regression 57
6.2 Predicted values and residuals 58
6.3 Hypothesis testing 59
6.4 Instrumental Variables 61
6.5 Logit Model 62
6.6 Probit Model 64
6.7 Post-Estimation: Publication-Ready Tables 66
Exercises 68

7 Graphs **69**

7.1 Histograms: Distribution of a Single Variable 69

7.2 Twoway Graphs: Relationships Between Variables 70

7.3 Bar Charts: Summarizing Categorical Data 70

7.4 Saving and Exporting Graphs 71

Exercises 72

References **73**

Preface

These notes, which are continuously updated[1], provide a comprehensive introduction to the fundamental principles of using Stata for data analysis and statistical computing. They are designed for the Econometrics course in the Bachelor program, as Stata is the reference software for this course and is widely used in various research fields, including economics, sociology, psychology, and statistics.

The course emphasizes the specification of an econometric model, the interpretation of model parameters, and the evaluation of model performance, rather than focusing on programming in Stata. However, code to obtain estimates and details on how to run the software will be provided, enabling students to write their own code, under their own responsibility, for applications that may differ from those considered here.

In the dynamic world of data-driven decision-making, the ability to utilize statistical software is an invaluable skill that empowers individuals across various fields. Whether you are a student, researcher, or professional, understanding how to navigate and leverage data analysis tools is crucial. Among the array of statistical software available, Stata stands out as a versatile and robust platform that caters to both novices and experienced statisticians alike.

These notes are tailored for those who may be taking their first steps into the realm of statistical analysis or those looking to deepen their understanding of Stata's

[1]check always to use the last version.

capabilities. Whether you aim to conduct econometric analyses, social science research, or any data-driven project, these notes will equip you with the skills and confidence needed to navigate the software with proficiency.

CHAPTER 1

Introduction

1.1 What is Stata

Stata is a comprehensive statistical software package designed for analyzing large datasets. It operates in both a graphical (windowed) environment and a command-line-driven interface, ensuring enhanced reproducibility. Starting with the 2003 release, Stata also introduced a graphical user interface (GUI) for command entry, catering to users with varying preferences.

Stata handles datasets organized as matrices, with variables represented in columns and observations in rows. It supports a diverse range of data types, encompassing numerical, categorical, and string data. This adaptability makes Stata a versatile tool for researchers across various disciplines. Stata's syntax is intuitive and easy to learn, providing a robust environment for conducting reproducible research.

As a result, Stata stands out as an excellent tool for data manipulation, statistical analysis, and graphical data visualization. It offers a complete set of standard univariate, bivariate, and multivariate statistical tools for descriptive statistics, statistical tests, analysis of variance, and regressions. Additionally, Stata provides specialized environments and tools for time-series econometrics, model simulation, bootstrapping, maximum likelihood estimation, nonlinear least squares, and various other advanced statistical techniques.

Stata is widely used in academia, government institutions, and industry for data analysis and interpretation. It is available in different versions with varying levels of features and capacity:

- Stata/MP: The most powerful version of Stata, optimized for dual-core and multicore/multiprocessor computers;
- Stata/SE: Stata designed for handling large datasets[1];
- Stata/IC: Stata suitable for analyzing moderate-sized datasets;
- Numerics by Stata: Stata tailored for embedded and web applications.

Stata runs seamlessly across various platforms, including Windows, Mac, Unix, and Sun Solaris.

Upon starting a new Stata session, four default windows are typically opened:

- **command window**: Commands can be directly typed into this window for execution;
- **results window**: This window displays the output generated by executed commands;
- **review window**: This window maintains a history of all typed commands, providing a convenient reference;
- **variables window**: This window lists the names and labels of all variables currently loaded in memory.

Additional windows can be opened using the corresponding menu options. Two frequently used additional windows are:

- **do-file editor window**: This window is used to create and edit do-files, which contain sequences of Stata commands that can be executed as a batch;

[1]The University of Bologna provides a license-free version of Stata/SE for its students.

- **viewer window**: This window displays help information on Stata syntax and commands.

At start, the default directory is considered, usually `"C:/data"`. This is the Stata working directory, and if no path is specified data and files are read/written from/in this directory. It can be changed using the dos command `cd "new_path"`.

1.2 The Reproducible Stata Workflow

To ensure all analysis is reproducible, Stata users must adopt a rigorous workflow centered on two key files: the **Do-File** and the **Log-File**.

Organizing Your Project and Setting the Directory

Every project should start by setting a root working directory. This directory acts as the central hub for your data, do-files, and output. Using **relative paths** is a best practice, as it ensures your code runs correctly even when the project folder is moved or shared with others. The current working directory can be checked and changed using the commands:

- `pwd`: Displays the current working directory (Print Working Directory).
- `cd "path_to_my_project"`: Changes the current working directory.

A crucial step for portability is to use a **global macro** to define the root directory at the start of your main do-file. For example, after defining `global root "folder_path"`, you can simplify all subsequent file references to `use "$root/data/data.dta; clear`.

The Do-File: Your Command Script

The **Do-File** (`.do` extension) is a script that contains all commands necessary to reproduce your analysis, from loading data to generating final estimates.

- Commands are executed by opening the do-file editor (or command: `doedit`) and pressing the "Do" button or highlighting and executing specific lines.

- It is best practice to begin every do-file with commands to clear memory and set preferences:

  ```
  clear all
  set more off
  ```

The latter command is useful when running lengthy codes, eliminating the need to press `"more"` in the `results window` while the code is executing, as the entire output is displayed automatically.

The Log-File: Capturing Output

The **Log-File** (`.log` or `.smcl` extension) captures all the output generated by Stata commands, including tables and estimation results. It is essential for documenting your results.

- `log using name.log, replace`: Starts recording the output to a file, replacing it if it already exists.

- `log close`: Stops recording the output.

- `log using name.log, append`: Continues writing to an existing log file.

Always start your do-file by opening a log and end it by closing the log.

Moreover, you can also open several **log-files** at the same time, the key condition is to name the.

- `log using name_1.log, replace name(log_name_1)`: Starts recording the output to a file, replacing it if it already exists.

- `log using name_2.log, replace name(log_name_2)`: Starts recording the output to a new file, replacing it if it already exists, keeping the previous open.

- `log close log_name_1`: Stops recording the output in the defined **log-file**.

- `log close _all`: Stops recording all the **log-files**.

Stata's Help System and Documentation

Stata's documentation is excellent and is your most reliable resource.

- `help command_name`: Displays detailed syntax, options, and examples for a specific command.
- `search keyword`: Searches the official Stata documentation for commands and topics related to a keyword.
- `findit keyword`: Searches official documentation and user-contributed packages (like the SSC archive) for broader, web-based results.

It is also important to organize the project folders well. A recommended practice is to create a main project folder, which serves as the central repository for project-related files. Within this main folder, establish sub-directories to organize specific file types:

- Dataset: Store the project's dataset within a dedicated folder;
- Output: Place the generated output files in a separate folder for easy identification and retrieval;
- Graphs: Allocate a folder for storing any generated graphs or visualizations;
- Log Files: Create a folder to store log files generated throughout the project;
- Do-Files (Optional): Establish a folder for storing do-files if desired[2].

By adopting this organized file structure, you enhance the portability of your code, enabling seamless execution on different computers. The clear separation of file types facilitates navigation and comprehension of the project's structure and workflow. Below you can find an example:

[2]They can be stored in the main directory as well.

```
*=================================
* Example of how to organize a new do-file
*=================================
clear all
capture log close
log close _all
set more off
version 19
*=================================
// Set paths to your file and directories
global root ".../main_directory" // Path to main folder
global Dataset "$root/Dataset" // Path to dataset folder
global Output "$root/Output" // Path to output folder
global Graphs "$root/Graphs" // Path to graphs folder
global Logs "$root/Logs" // Path to logs folder
*=================================
cd "$root"
use "$Data/dataset"
*=================================
log using "$Log/log_name.txt",text replace name(log_name)
```

By modifying only the line referencing `root`, you can ensure that the entire code runs seamlessly on any computer, provided the main folder and subfolders have been created. This approach promotes code portability and simplifies the process of sharing and executing code across different environments.

The key to this strategy lies in isolating the system-specific path information to a single location, allowing for easy modification without affecting the core functionality of the code. This technique streamlines code execution and enhances its adaptability to diverse computing environments.

tips: to insert comments in the do-file there are several ways:

- Adding * at the beginning of a line allows to insert notes;
- Adding // at the beginning of a line allows to insert notes. Moreover, it can also be used after commands on the same line;
- Everything is written inside /* ...*/ is considered as a comment, therefore it allows for multiple line notes.

CHAPTER 2

Data Handling

2.1 Importing External Data

While the `use` command loads existing Stata datasets (`.dta`), much of your work will involve importing data from other formats. The two most common types are delimited text files (like CSV) and spreadsheets (like Excel).

Importing Delimited Text Files (CSV)

The `import delimited` command is the robust way to import data where columns are separated by a delimiter (e.g., a comma, tab, or space).

```
import delimited "data/external.csv", clear
import delimited "data/data_tab.txt", delimiter("tab") clear
```

The `clear` option is necessary to remove any current dataset from memory. If the data is separated by a comma, the delimiter does not need to be specified.

Importing Excel Files

The `import excel` command provides options specific to spreadsheet formats, such as specifying the sheet or range of cells.

```
import excel "data/survey.xlsx", sheet("Sheet1") firstrow clear
```

The `firstrow` option is crucial, as it tells Stata to treat the first row of the sheet as **variable names** rather than data. Without it, Stata would assign default names like `A`, `B`, `C`, etc.

2.2 Dataset Management

Data files in Stata are commonly saved with the extension `.dta`. To load a dataset, such as `dataset.dta`, the command is:

```
use "directory/dataset.dta", clear
```

where `directory` refers to the specific folder on your computer where the dataset is saved and `clear` is an optional command that is necessary if there is an existing dataset currently loaded in Stata. Additionally, Stata includes several built-in datasets that can be easily accessed using the `sysuse` command. These datasets are useful for learning and experimenting with various Stata commands and features. To access these, the appropriate command is:

```
sysuse dataset_name, clear
```

For example, to load the auto dataset, which contains information about different car models, you would use:

```
sysuse auto, clear
```

This command loads the auto dataset into your current Stata session, allowing you to perform various operations and analyses on it.

To streamline the process of accessing data, it is recommended to set the directory where all data files are stored:

```
cd "directory"
use "dataset.dta"
save "dataset.dta"
```

It is important always to remember to save your files. To overwrite an existing file, use:

```
save "dataset.dta", replace
```

2.3 Memory Allocation

By default, Stata allocates 10.10 MB of memory. To increase this allocation, enter the following command:

```
set memory 50m
```

If the allocated memory is insufficient, Stata will indicate "‘no room to add more observations,” prompting you to allocate more memory as needed.

2.4 Variables and Labels

In Stata, variables correspond to columns in a dataset, while observations correspond to rows. Variables can be either alphanumeric (strings) or numeric. Variable names must not contain spaces. To rename a variable:

```
rename x y
```

Assigning Value Labels

While a **variable label** describes the variable itself (e.g., "Gender"), **value labels** map numeric codes to meaningful text (e.g., 1 to "Male", 2 to "Female"). This is vital for making descriptive statistics and regression output interpretable. This process involves two steps:

1. **Define** the map (the label set) using `label define`.
2. **Assign** the map to a variable using `label values`.

```
label define gender_lbl 1 "Male" 2 "Female"
label values gender_var gender_lbl
tab gender_var // Tabulate to check the result
```

You can view all defined value labels using the `label list` command.

2.5 Generating and Modifying Variables

New variables can be created using the `generate` or `egen` commands. For example:

```
generate x = 1
```

generates a variable `x` that takes value 1 (a column of 1) for each observation (row). We can also generate variables using functions (that will be explained later). An example is:

```
generate percentage = 100*(old-new)/old if old > 0
```

Finally, we can also modify existing variables by using the `replace` command:

```
replace oldvar = exp [if exp] [in range]
```

CHAPTER 3

Essential Commands

Stata commands can be categorized broadly into two types:

1. Commands that provide data-related information, such as describe, list, summarize, and tabulate.
2. Commands that modify data, including keep, drop, and generate.

Many commands follow a standardized syntax format, typically expressed as:

```
[by varlist:]  command varlist [if exp] [in range]
```

The components of this syntax are defined as follows:

- The prefix `by varlist` ensures that the command operates on each unique set of values of the specified variable(s), contingent upon the data being pre-sorted according to the variables listed in *"varlist"*.
- `command` denotes the specific operation that Stata is instructed to perform (e.g., summarize, list, save).

- `varlist` refers to the variables upon which the command is executed, often being optional in various commands.
- The expression `if exp` signifies a conditional statement, where the command applies only to observations meeting the specified condition. If not specified, the command acts on all observations.
- `in range` defines a subset of observations based on specified criteria (i.e., an interval or range).

note: commands between [ ] are optional

3.1 Describe

The `describe` command provides a summary of the data currently in memory or present in a dataset. Its syntax is structured as follows:

```
describe [varlist | using filename] , [short detail]
```

Here is a breakdown of its components:

- `[varlist | using filename]`: specifies variables or a dataset file to describe. This part is optional.
  - `varlist`: lists specific variables to describe.
  - `using filename`: describes a dataset stored in the file named filename.
- `[, short detail]`: optional parameters that modify the output:
  - `short`: provides concise information about each variable.
  - `detail`: includes details such as the width of a single observation and the maximum size of the dataset.

These examples illustrate how the describe command can be used in different contexts to obtain varying levels of detail about the dataset within Stata:

- `describe`: displays a standard summary of the dataset in memory
- `describe, short`: provides a brief summary for each variable in the dataset, omitting detailed information.
- `describe, detail`: offers a comprehensive summary, including details about each variable's properties, the width of a single observation, and the maximum size of the dataset.

3.2 Display

The `display` command serves multiple purposes, including displaying strings and the values of scalar expressions. It can also function as a calculator when used directly in the Command window. This is the syntax of its usage:

```
display [subcommand [subcommand]...]
```

where `subcommand` can be a string enclosed in double quotes or a scalar expression. When you input a string enclosed in double quotes (`" "`), `display` outputs that string directly. While, when you input a mathematical expression or a function, `display` evaluates it and shows the result. You can also use `display` directly in the Command window to quickly evaluate expressions or display specific strings. In summary, `display` is versatile, serving both as a means to output strings and as a tool for performing calculations and displaying their results directly within the Stata environment.

For example,

- `display "this is a sentence"` simply displays the string *"this is a sentence"* in the output window.
- `display 5 + ln(70)` calculates the natural logarithm of 70 and adds 5 to it, then displays the result.

3.3 List

The `list` command in Stata is used to display the values of variables within a dataset. Its syntax and options allow for flexibility in how data is presented. Here's an explanation of its syntax and examples:

```
[by varlist:]  list [varlist] [if exp] [in range] ///
[, [no]display nolabel noobs]
```

where:

- `[by varlist:]` specifies that the list command is to be performed by each unique set of values of the variables listed in varlist. The dataset must be sorted accordingly.
- `list [varlist]` specifies which variables to display. If varlist is not specified, all variables in the dataset are listed.
- `[if exp]` specifies a condition (expression) that restricts the observations displayed.
- `[in range]` specifies a range of observations to be displayed.
- `[, [no]display nolabel noobs]` are options that modify the display:
  - `display` forces the format to display or table format (`nodisplay` suppresses this).
  - `nolabel` displays numeric codes rather than variable labels.
  - `noobs` suppresses the appearance of the number of observations.

For example:

- `list in 1/5` displays values for all variables from observations 1 to 5 in the dataset.

- `list age education` displays values for the variables 'age' and 'education' for all observations in the dataset.
- `list if age > 18` displays values for all variables only for observations where 'age' is greater than 18.
- `list education if age < 3` cisplays values specifically for the variable 'education' for observations where 'age' is less than 3.

Additionally, the `list` command is useful for examining specific subsets of data based on conditions or ranges. By default, list displays values in a tabular format unless overridden by the `nodisplay` option. It's important to ensure variables are correctly specified and conditions are clear to retrieve the desired subset of data.

This command is essential for inspecting and reviewing specific data subsets within a dataset in Stata, providing flexibility in data exploration and analysis.

3.4 Format

The `format` command allows you to control how data is presented without altering the underlying values. This is particularly useful for making output more readable and ensuring that numeric and string data are displayed in a consistent and meaningful way. It specifies how numeric values and strings are displayed in the output. This does not change the actual values stored in the dataset but only how they are presented.

```
format varname %fmt
```

where

- `varname` is the variable whose display format you want to change.
- `%fmt` is the format specification that dictates how the variable will be displayed.

3.5 Drop and Keep

The `drop` and `keep` commands are essential for data management, allowing you to refine your dataset by removing unnecessary variables or observations (using `drop`) or retaining only those that are relevant (using `keep`). Proper use of these commands helps streamline your data analysis workflow.

The `drop` command is used to remove variables or observations from the dataset currently in memory.

Drop Variables

```
drop varlist
```

where

- `varlist` is the list of variables to be dropped from the dataset.

For example,

- `drop age` drop the variable `age` from the dataset.

Drop observations based on a condition

```
[by varlist:]  drop if exp
```

where

- `by varlist:` specifies that the `drop` operation should be performed within groups defined by `varlist`. The data must be sorted by `varlist` first.
- `if exp` drops observations that meet the specified condition `exp`.

For example,

- `drop if income > 50000` drop observations where the variable `income` is greater than 50000.
- `sort gender // by gender:  drop if age < 18` drop observations where `age` is less than 18 within each group defined by `gender`.

Drop observations in a specific range

```
drop in range [if exp]
```

where

- `in range` specifies the range of observations to drop.
- `if exp` is an optional condition to drop observations within the specified range that meet the condition `exp`.

For example,

- `drop in 1/5` drop the first 5 observations.

The `keep` command works similarly to the `drop` command, but instead of specifying the variables or observations to remove, you specify the ones to retain.

Keep Variables

```
keep varlist
```

where

- `varlist` is the list of variables to be dropped from the dataset.

For example,

- `keep age income` keep only the variables `age` and `income` in the dataset.

Keep observations based on a condition

```
[by varlist:]  keep if exp
```

where

- `by varlist:` specifies that the `drop` operation should be performed within groups defined by `varlist`. The data must be sorted by `varlist` first.
- `if exp` drops observations that meet the specified condition `exp`.

For example,

- `keep if income > 50000` keep observations where the variable `income` is greater than 50000.
- `sort gender // by gender:  keep if age < 18` keep observations where `age` is less than 18 within each group defined by `gender`.

Keep observations in a specific range

```
keep in range [if exp]
```

where

- `in range` specifies the range of observations to drop.
- `if exp` is an optional condition to drop observations within the specified range that meet the condition `exp`.

For example,

- `keep in 1/5` keep the first 5 observations.

3.6 Classes of Operators

Stata distinguishes between several classes of operators, each serving different purposes in algebraic and logical expressions.

Arithmetic Operators

These operators perform basic mathematical operations.

- `+` adds two numbers.
- `-` subtracts the second number from the first.
- `*` multiplies two numbers.
- `/` divides the first number by the second.
- `^` raises the first number to the power of the second.

note: any arithmetic operation involving a missing value or an impossible operation (e.g., division by zero) results in a missing value.

String Operators

These operators are used for manipulating strings.

- `+` combines two strings into one, e.g., "variance-" + "covariance matrix" results in "variance-covariance matrix"

Relational Operators

These operators compare two values and return a logical value: true or false.

- `<` is less then.
- `>` is greater then.
- `<=` is less then or equal to.

- `=>` is greater than or equal to.
- `==` is equal to.
- `!=` or $\sim$ `=` are not equal to.

Logical Operators

These operators combine or negate logical expressions and return a value of 1 if the expression is true and 0 if it is false.

- `&` (logical and) returns true if both expressions are true.
- `|` (logical or) returns true if at least one of the expressions is true.
- `!` (logical not) returns true if the expression is false.

Understanding these operators allows for constructing complex expressions, whether performing arithmetic calculations, manipulating strings, comparing values, or evaluating logical conditions. Here's a quick example combining these elements:

```
generate example_var = (age > 18) & (income < 50000)
```

This creates a new variable `example_var` that is 1 if `age` is greater than 18 and `income` is less than 50000, and 0 otherwise.

3.7 Functions

Functions are used within expressions and can accept various types of arguments, including other functions. They serve different purposes, from mathematical calculations to statistical operations and random number generation.

Mathematical Functions

- `exp(x)` calculates the exponential of `x`.
- `log(x)` or `ln(x)` computes the natural logarithm (`ln`) or logarithm base 10 (`log`) of `x`.
- `sqrt(x)` calculates the square root of `x`.
- `abs(x)` calculates the absolute value of `x`.

Trigonometric Functions

Standard trigonometric functions like `sin(x)`, `cos(x)`, `tan(x)`, are also available.

Statistical Functions

- `chiprob(df, x)` computes the upper tail probability of the chi-square cumulative distribution function with `df` degrees of freedom.
- `fprob(df1, df2, f)` calculates the upper tail probability of the `F` cumulative distribution function with `df1` and `df2` degrees of freedom.
- `invnorm(p)` computes the quantile function (inverse of the cumulative distribution function) of the standard normal distribution for probability `p`.
- `normd(z)` computes the density of the standard normal distribution at `z`.
- `normprob(z)` computes the cumulative distribution function of the standard normal distribution at `z`.
- `tprob(df, t)` calculates the two-tailed probability of the Student's t distribution with `df` degrees of freedom.

Random Number Generation

- `uniform()` generates pseudo-random numbers uniformly distributed on the interval [0, 1).

- `invnorm(uniform())` generates pseudo-random numbers following a standard normal distribution using the `invnorm` function applied to `uniform()`.

Special Functions

- `float(x)` converts `x` to a floating-point representation.
- `int(x)` converts `x` to an integer.
- `max(x1, x2, ..., xn)` and `min(x1, x2, ..., xn)` returns the maximum or minimum value among the arguments, ignoring missing values.
- `sign(x)` returns `-1` if `x < 0`, `0` if `x = 0`, `1` if `x > 0`, and `.` if `x = .`.
- `sum(x)` computes the sum of `x`, treating missing values as zeros.

Let's see some examples of usage of these functions:

- `generate loginc = ln(income)` calculates the logarithm of income and generates a new variable.
- `generate rand_normal = invnorm(uniform())` generates pseudo-random numbers following a standard normal distribution.
- `generate tota_score = sum(score1, score2, score3)` calculates the sum of a variable.

These functions provide powerful capabilities for manipulating and analyzing data, enabling a wide range of statistical and mathematical operations directly within the software environment.

3.8 Sort and Gsort

The commands `sort` and `gsort` are fundamental for organizing data in a specified order, which is often essential for conducting analyses and generating meaningful summaries or reports.

The `sort` command arranges data in ascending order based on the values of the variables specified in `varlist`. Missing values are treated as the largest values and placed at the bottom. The syntax is:

```
sort varlist [in range]
```

where:

- `varlist` specifies the variables based on which the data should be sorted.
- `[in range]` optionally specifies a range of observations to sort.

For example, if we want to sort data by the variable age, `sort age` arranges the dataset in ascending order of the variable age.

The `gsort` command is used to sort observations in either ascending (+) or descending (-) order of the variables specified in `varlist`. The syntax is:

```
gsort [+|-] varname [+|-] varname [...]  [, generate(newvar) mfirst]
```

where:

- `+` or `-` before `varname` specifies ascending or descending order for each variable.
- `generate(newvar)` is optional and creates a new variable `newvar` that assigns group numbers based on the sorted order.
- `mfirst` is optional and places missing values (.) first in descending order instead of last.

For example:

- `gsort +y` arranges observations in ascending order of the variable y.

- `gsort -y` sorts observations in descending order of the variable y.
- `gsort -y age` orders observations first by `y` in descending order and then by `age`.
- `gsort -y, gen(revy)` creates a new variable `revy` where observations are numbered according to their order in descending `y`.

3.9 The Power of Egen

While the `generate` command creates a new variable based on existing variables, the `egen` command (short for *extended generate*) is designed for advanced, group-level operations and functions that are often tedious to perform with standard `generate` and `replace` commands.

```
egen newvar = function(arguments) [if] [in] [, options]
```

Its strength lies in its ability to compute summary statistics (like means, standard deviations, and ranks) and assign them to **every observation** in the data or within specified groups, all in a single line.

- `egen mean_price = mean(price)` // Assigns the overall mean price to every observation.
- `bysort foreign: egen group_mean_price = mean(price)` // Calculates the mean price for domestic and foreign cars separately, and assigns the group-specific mean to all observations in that group.
- `egen rank_mpg = rank(mpg)` // Creates a variable with the rank of each car's mileage.
- `egen count_obs = count(make), by(foreign)` // Counts the number of observations within each group defined by `foreign`.

The use of `egen` with `bysort` is a cornerstone of efficient Stata programming and is highly recommended for creating group-level metrics, which are often used as control variables in estimation.

3.10 Group Commands

The `by` command allows you to perform operations on data grouped by the values of a specified variable. Before using `by`, it is necessary to sort the data based on that variable. The syntax is:

```
sort varname
by varname:  command
```

where:

- `sort varname` arranges the dataset in ascending order based on the variable `varname`.
- `by varname:  command` executes `command` separately for each group defined by unique values of `varname`.

For example, consider the following commands:

- `sort gender`
  `by gender:  sum age` sorts the data by `gender` and then computes the summary statistics (`sum`) of age separately for each group defined by gender.
- `sort gender`
  `by gender:  gen lag2 = age[_n-1]` sorts the data by `gender` and then creates a new variable `lag2` that contains the lagged value of `age` within each group defined by `gender`.

We can also use the `quietly` command to suppresses output from appearing on the screen while still logging the output to the log file.

```
quietly command
```

where:

- `command` represents any Stata command you wish to execute quietly.

For example:

- `quietly by gender:`
  `gen lag2 = age[_n-1]` calculates the lagged variable `lag2` within each group defined by `gender` quietly, meaning the output will not be shown on the screen but will still be logged in the `log file` (if open).

These commands, `by` and `quietly`, are useful for conducting group-specific analyses and managing output visibility, enhancing efficiency and organization in data analysis workflows.

3.11 Generate Dummy

Creating dummy variables in Stata is straightforward and can be done using the `tab` or `gen` command, depending on whether you want dummies for all categories or a specific category.

Generating Dummy Variables for All Categories

To generate a dummy variable for each unique value of a categorical variable, you can use the `tab` command with the `gen` option.

```
tab varname, gen(newvar)
```

where:

- `varname` is the categorical variable for which you want to create dummies.

- `gen(newvar)` specifies the prefix for the new dummy variables.

For example, if you have a variable `education` representing educational attainment, you can create dummy variables for each category in `education` as follows:

- `tab education, gen(dummy)`

This command creates dummy variables `dummy1`, `dummy2`, `dummy3`, etc., corresponding to each unique value in `education`. For instance:

- `dummy1 = 1` if `education` indicates elementary school and 0 otherwise.
- `dummy2 = 1` if `education` indicates middle school and 0 otherwise.
- And so on for all categories present in `education`

Generating a Dummy Variable for a Specific Category

If you want to create a dummy variable for a specific category of a variable, you can use the `gen` command with a logical condition (or more).

```
gen newvar = (varname == value)
```

where:

- `newvar` is the name of the new dummy variable.
- `varname == value` is the logical condition specifying the category for which the dummy should be 1.

For example, to create a dummy variable `d_elem` that is 1 if `education` equals 1 (assuming 1 represents elementary school) and 0 otherwise:

- `gen d_elem = (education == 1)`

This command generates a dummy variable `d_elem` with:

- `d_elem = 1` if `education` is 1 (elementary school).

- `d_elem = 0` otherwise.

We can also combine multiple logical operators at the same time. For example, For example, to create a dummy variable `d_elem_f` that is 1 if `education` equals 1 (assuming 1 represents elementary school) and `gender` equals 1 (assuming 1 represents female) and 0 otherwise:

- `gen d_elem_f = (education == 1 & gender==1)`

This command generates a dummy variable `d_elem_f` with:

- `d_elem_f = 1` if `education` is 1 (elementary school) and `gender` is 1 (female).
- `d_elem_f = 0` otherwise.

By using these commands, you can efficiently generate dummy variables to represent categorical data, which can then be used in various statistical analyses and modeling.

Exercises

1. Use the `describe` command to display summary information about the variables in the dataset `auto.dta`.
2. List the first 5 observations for the variables `make`, `price`, and `mpg`.
3. Create a new variable called `price_mpg` that is the product of `price` and `mpg`.
4. Sort the dataset by the variable `mpg` and list the first 5 observations.
5. Drop the variable `rep78` from the dataset.

CHAPTER 4

Descriptive Statistics

Descriptive statistics provide simple summaries about the sample and the measures. They form the basis of virtually every quantitative analysis of data. Stata offers a variety of commands to obtain these statistics, which include measures of central tendency, variability, and distribution shape.

4.1 Count

The `count` command is a versatile tool for counting observations based on various criteria. By using conditions and grouping options, you can easily obtain counts specific to your analysis needs. This helps in summarizing and understanding the structure of your dataset. It is used to count the number of observations in the dataset, with the option to apply various conditions or ranges as restrictions.

```
[by varlist:]  count [if exp] [in range]
```

where

- `by varlist:` applies the count operation within groups defined by `varlist`. The dataset must be sorted by `varlist` beforehand.

- `if exp` counts observations that meet the specified condition `exp`.
- `in range` counts observations within the specified `range`.

If no conditions are specified, `count` will count all observations in the dataset. For example,

- `count` counts the total number of observations in the dataset.
- `count if y > 20000` counts the number of observations where the variable `y` is greater than 20000.
- `sort age // by age:  count if y < 0` first, the data is sorted by the variable `age`. Then, within each group defined by `age`, the command counts the number of observations where the variable `y` is less than 0.

4.2 Summarize

The `summarize` command (or its abbreviation, `sum`) provides a set of descriptive statistics for specified variables. It is a versatile command that can be tailored with various options to yield more detailed statistical information or to customize the display format.

```
[by varlist:]  summarize [varlist] [weight] [if exp] [in range] ///
[, detail | meanonly format]
```

where

- `by varlist:` performs the summary statistics for each unique value of the variables in `varlist`.
- `varlist` specifies the variables for which summary statistics are to be calculated. If omitted, statistics for all variables are displayed.
- `weight` applies weights to the observations.
- `if exp` computes statistics only for observations that meet the specified condition.

- `in range` computes statistics only for the specified range of observations.
- `, detail` provides additional statistics such as kurtosis, skewness, the four largest and smallest values, and various percentiles.
- `, meanonly` suppresses the display of results and the calculation of the variance (only allowed when `detail` is not specified).
- `, format` displays summary statistics using the display format associated with the variables rather than the default format.

For example,

- `summarize` provides mean, standard deviation, min, and max for all variables in the dataset.
- `sum age` Provides basic summary statistics for the variable `age`.
- `sum y, detail` provides detailed summary statistics for the variable `y`, including additional measures like kurtosis and skewness.
- `sum age if age > 30` provides summary statistics for `age` but only for observations where `age` is greater than 30.

The `summarize` command is an essential tool for obtaining descriptive statistics. By utilizing various options such as `detail`, `meanonly`, and `format`, users can extract basic or detailed statistical summaries, apply conditions, and group results by categories. These features make it highly flexible and useful for preliminary data analysis.

4.3 Means

The `means` command calculates arithmetic, geometric, and harmonic means, along with their respective confidence intervals, for specified variables. This command is useful for summarizing central tendencies of the data.

```
means [varlist] if exp [in range] [, add(#) only level(#)]
```

where

- `varlist` specifies the variables for which means are to be calculated. If omitted, means for all variables are calculated.
- `if exp` computes means only for observations that meet the specified condition.
- `in range` computes means only for the specified range of observations.
- `, add(#)` adds the value `#` to each variable in `varlist` before calculating the mean and confidence interval. This is useful when analyzing variables with non-positive values.
- `, only` modifies the `add()` option such that `#` is added only to variables with at least one non-positive value.
- `, level(#)` specifies the confidence level percentage for the confidence intervals. The default is 95%.

For example,

- `means` calculates and displays arithmetic, geometric, and harmonic means along with confidence intervals for all variables in the dataset.
- `means var1 var2` calculates the means for `var1` and `var2`.
- `means var1 if age > 30` calculates means for `var1` but only for observations where `age` is greater than 30.

Alternatively, The `ci` command can be used if only arithmetic means and corresponding confidence intervals and standard errors are needed.

```
ci [varlist] if exp [in range] [, level(#)]
```

where

- `varlist` specifies the variables for which means are to be calculated. If omitted, means for all variables are calculated.
- `if exp` computes means only for observations that meet the specified condition.
- `in range` computes means only for the specified range of observations.
- `, level(#)` specifies the confidence level percentage for the confidence intervals. The default is 95%.

The `means` command is a comprehensive tool for calculating various types of means and their confidence intervals. The `add()` and `only` options provide flexibility for handling variables with non-positive values. For simpler tasks focused on arithmetic means and confidence intervals, the `ci` command is a suitable alternative.

4.4 Centile

The `centile` command reports the percentiles of specified variables, along with their confidence intervals. By default, the confidence intervals are calculated using a binomial method, which makes no assumptions about the distribution of the variable in question.

```
centile [varlist] [if exp] [in range] [, centile(numlist) cci ///
normal meansd level()]
```

where

- `varlist` specifies the variables for which means are to be calculated. If omitted, means for all variables are calculated.
- `if exp` computes means only for observations that meet the specified condition.
- `in range` computes means only for the specified range of observations.

- `, centile(numlist)` specifies the percentiles to be reported (e.g., `centile(25 50 75)` for the 25th, 50th, and 75th percentiles). If not specified, the median (50th percentile) is reported by default.
- `, cci` conservative confidence interval, prevents interpolation when calculating confidence limits without distributional assumptions.
- `, normal` specifies that confidence intervals should be obtained under the assumption that the estimated percentiles are normally distributed.
- `, meansd` calculates confidence intervals assuming that the estimated percentiles are normally distributed using the mean and standard deviation.
- `, level(#)` specifies the confidence level percentage for the confidence intervals. The default is 95%.

For example,

- `centile` calculates and displays the median for all variables in the dataset.
- `centile var1 var2` calculates the percentiles for var1 and var2.
- `centile var1, centile(25 50 75)` reports the 25th, 50th, and 75th percentiles for var1.
- `centile var1 if age > 30` calculates percentiles for var1 but only for observations where age is greater than 30.

Morover, the `pctile` command creates a new variable containing the percentiles of an expression. This is useful for further analysis or plotting.

```
pctile newvar = exp
```

where

- `newvar` is the name of the new variable to be created.

- `exp` is the expression whose percentiles are to be calculated.

For example,

- `pctile p_income = income` creates a new variable `p_income` that contains the percentiles of `income`.

The `centile` command is a powerful tool for calculating and reporting percentiles and their confidence intervals for specified variables. With options for conservative confidence intervals, normal distribution assumptions, and specifying confidence levels, it offers flexibility for detailed statistical analysis. The `pctile` command complements this by allowing the creation of new variables that store percentile values, facilitating further data manipulation and analysis.

4.5 Cumul

The `cumul` command is used to create a new variable containing the cumulative distribution function (CDF) of an existing variable. This can be useful for understanding the distribution and cumulative frequencies of your data.

```
cumul varname [weight] [if exp] [in range], gen(newvar) ///
[freq by(varlist)]
```

where

- `varname` is the existing variable for which you want to calculate the cumulative distribution.
- `weight` specifies weights to be applied to the observations.
- `if exp` conditions to apply to the observations.
- `in range` specifies a range of observations to include.

- `gen(newvar)` specifies the name of the new variable that will contain the CDF values (this is not optional).
- `freq` requests that the CDF be in units of frequency. If not specified, the CDF is normalized so that `newvar` equals 1 for the largest value of `varname`.
- `by(varlist)` specifies that the CDF should be calculated separately for groups defined by `varlist`.

For example,

- `cumul varname, gen(cdf_varname)` creates a new variable containing the CDF of `varname`.
- `cumul varname [pweight=weight_var], gen(cdf_varname)` creates a weighted CDF.

The `cumul` command is an essential tool for generating the cumulative distribution function of a variable. By using this command, you can analyze the cumulative frequencies and distribution characteristics of your data, either across the entire dataset or within specific groups. The command is flexible, allowing for weighted calculations, conditional inclusion of observations, and grouping by other variables.

4.6 Correlate

The `correlate` command is used to report the variance or correlation matrix for specified variables. It provides a useful way to understand the relationships between variables within your dataset. Observations with missing values may be excluded from the calculations.

```
[by varlist:]  correlate [varlist] [weight] [if exp] [in range] ///
[, means noformat covariance wrap]
```

where

- `varlist` is the list of variables for which the correlation or variance matrix will be computed.
- `weight` specifies weights to be applied to the observations.
- `if exp` conditions to apply to the observations.
- `in range` specifies a range of observations to include.
- `means` reports additional statistics such as means, standard deviations, minimum, and maximum values.
- `noformat` reports the summary statistics requested by the `means` option in the default `g` format, regardless of the format associated with the variables.
- `covariance` reports the covariance matrix instead of the correlation matrix.
- `wrap` ensures no special action is taken to make large matrices readable.

For example,

- `correlate var1 var2 var3` computes the correlation matrix of variables `var1`, `var2`, and `var3`.
- `correlate var1 var2 var3 [pweight=weight_var]` computes a weighted correlation matrix.

The `correlate` command is a tool for examining the relationships between variables through correlation and covariance matrices. It offers flexibility with options for including weights, specifying conditions and ranges, and reporting additional statistics. Understanding these relationships can provide valuable insights into the structure and dynamics of your data.

4.7 Tables

Table

The `table` command is used to create customized tables summarizing statistical information. These tables can present data across various categories and can be tailored with a wide range of options to suit specific needs.

```
table rowvar [colvar [supercolvar]] [weight] [if exp] [in range] ///
[, options]
```

where

- `rowvar` is the variable that defines the rows of the table.
- `colvar` is the variable that defines the columns of the table.
- `supercolvar` is an optional variable that creates an additional level of columns, grouping the main columns.
- `weight` specifies weights to be applied to the observations.
- `if exp` specifies a condition to include only certain observations.
- `in range` specifies a range of observations to include.

The `contents(clist)` option specifies the statistics to display in the table's cells, allowing up to five different statistics. If not specified, the default is `contents(freq)`, showing frequencies. The `by(superrow_varlist)` option adds another level of rows for grouping. The `row` and `col` options add totals for each row and column, respectively. The `format(%fmt)` option defines the numerical format for values, while `center` and `left` control the alignment of data within the cells. The `concise` option creates a more compact table by omitting unnecessary lines and spaces. The `missing` option displays missing data as periods (.), and `replace` allows overwriting an existing table. The `name(string)` option assigns a name to the table for future reference. `cellwidth(#)`, `csepwidth(#)`, `scsepwidth(#)`, and `stubwidth(#)` adjust the widths of the table cells, column separators, super-column separators, and row label areas, respectively.

For example, suppose you want to create a table showing the average income (`income`) across different education levels (`educ`) and genders (`sex`):

- `table educ sex, contents(mean income) row col format(%9.2f)`

In this example, The table will show the mean of `income`, with the rows defined by educ and the columns by `educ`, including row and column totals, and all numbers formatted to two decimal places (`format(%9.2f)`).

To briefly sum up, the `table` command allows you to generate detailed tables that can include various summary statistics, display totals, and format the results to meet specific presentation requirements. It is particularly useful for reporting summary statistics in a structured and easily interpretable format.

Tabulate

The `tabulate` command is used to generate frequency tables for one or two variables, along with various measures of association, such as Pearson's chi-squared, the likelihood ratio chi-squared, and Fisher's exact test.

One-Way Frequency Table

```
[by varlist:]  tabulate varname [weight] [if exp] [in range] ///
[, options]
```

Two-Way Frequency Table

```
[by varlist:]  tabulate varname1 varname2 [weight] [if exp] ///
[in range] [, options]
```

The `all` option includes all available statistical tests, such as chi-squared, likelihood ratio chi-squared, Cramér's V, Goodman and Kruskal's gamma, and Kendall's tau-b.

The `cell` option displays relative frequencies for each cell in a two-way table, while `chi2` computes Pearson's chi-squared statistic for testing the independence of rows and columns. The `column` option shows relative frequencies within each column, and `exact` provides significance levels using Fisher's exact test. The `gamma` option shows Goodman and Kruskal's gamma with its standard error, and `generate(varname)` creates indicator variables for each observed value. The `lrchi2` option displays the likelihood ratio chi-squared statistic, while `matcell(matname)`, `matcol(matname)`, and `matrow(matname)` save cell frequencies, column totals, and row totals to specified matrices, respectively. The `missing` option includes missing values in statistical calculations. The `nofreq` option suppresses the display of frequencies, and `nolabel` displays numeric codes instead of value labels. The `plot` option produces a bar chart of relative frequencies for a one-way table, `row` shows relative frequencies within each row, `subpop(varname)` excludes certain observations, `taub` displays Kendall's tau-b with its standard error, and `V` shows Cramér's V statistic.

Let's see some examples:

- `tabulate sex`
- `tabulate sex education`
- `tabulate education, generate(edu_dummy)`

In the latter example, the command generates a separate variable for each category of the `education` variable. For instance, if `education` has seven categories, the command creates seven dummy variables, each corresponding to one of those categories. These new variables will be named `edu_dummy1`, `edu_dummy2`, . . . , `edu_dummy7`.

Tabsum

The `tabulate, summarize()` command is used to produce summary statistics tables with one or two variables. While the `table` command may offer more flexibility, `tabulate, summarize()` is generally quicker. The general syntax is:

```
[by varlist:]  tabulate varname1 [varname2] [weight] [if exp] ///
[in range], summarize(varname3) [options]
```

Here, the `summarize(varname3)` option specifies the variable for summary statistics. You can use `[no]means`, `[no]standard`, `[no]freq`, and `[no]obs` to include or suppress means, standard deviations, frequencies, and the number of observations, respectively. The `missing` option treats missing values as valid categories instead of excluding them from the analysis.

For instance, if you want to generate a table showing the summary statistics (such as means and standard deviations) for a variable `income` across categories of `education` and `gender`, the command would look like this:

- `tabulate education gender, summarize(income) means standard nofreq noobs`

In this example:

- The table will include means and standard deviations of `income` for each combination of `education` and `gender`.
- Frequencies and the number of observations will be suppressed from the output.

This command is particularly useful when you need a quick summary of your data, organized by specific categories.

4.8 Loops

The `foreach` command is a powerful tool for repeating the same procedure on multiple variables or items. This is especially useful when working with large datasets or applying the same operation to several variables.

```
foreach var of local list_name {
    // Commands using `var'
    }
```

where

- `local list_name` defines a local macro containing a list of items (e.g., variable names).
- `foreach var of local list_name {` starts the loop, where `var` iterates over each item in `list_name`.
- `// Commands using var` specifies the commands to be executed for each item in the list.
- } ends the loop.

The benefits of using `foreach` are:

- **Efficiency:** it automates repetitive tasks, saving time and reducing the chance of errors.
- **Scalability:** it is easily scalable for large datasets or numerous variables.
- **Flexibility:** it can be combined with various Stata commands to perform complex data manipulations and analyses.

The `foreach` command is a versatile tool for automating repetitive tasks on multiple variables or items. By defining a local macro and looping through each item, you can efficiently apply commands and manage large datasets easily. The correct use of quotes and curly braces is essential to ensure proper execution of the loop.

tips: how to insert `

- on Mac or Linus press option + \
- on Windows press Alt + 96 or AltGr + ’

Exercises

1. Count the number of cars in the dataset `auto.dta` with a price greater than $10,000.
2. Provide summary statistics for the variable `weight`.
3. Get detailed summary statistics for the variable `mpg`.
4. Create a frequency table for the variable `foreign`.
5. Calculate the mean of the variable `price`.

CHAPTER 5

Data Manipulation

The main commands used to *"combine"* data are `append` and `merge`. To modify the data format, we will study other commands such as `reshape`, `collapse`, `contract`, `expand`, and `fillin`.

5.1 Append

The `append` command combines datasets vertically by adding observations. This technique is used to integrate a dataset currently in memory, referred to as the master data, with one or more datasets stored on disk in STATA format, known as the using data.

The basic syntax is:

```
append using filename [, nolabel]
```

where `nolabel` prevents copying value label definitions from the dataset on disk. If the filename is specified without an extension, it is assumed to be a Stata file, i.e., `.dta`.

The `append` command is designed to address the problem of adding more data to an existing dataset. This command combines different datasets by appending observations vertically, and it is not necessary for all variables to be common. For instance, consider a dataset, say `second.dta`, that might share variables **a** and **b** with another dataset, say `first.dta`, but also include variable **d**, which `first.dta` does not have. `append` handles this situation by appending observations regardless of whether all variables are present in both datasets. Observations from the appended dataset will have missing values for variable **d**, if it exists only in `second.dta`.

In the `append` command, the dataset currently in memory is referred to as the *"master"* dataset, while the dataset being attached is called the *"using"* dataset. This distinction is crucial for understanding the next command.

5.2 Merge

The `merge` command combines datasets horizontally by adding variables. Its simplest form is one-to-one matching. This command combines observations from the dataset currently in memory (referred to as the master dataset) with the dataset stored as filename (referred to as the using dataset). The basic syntax is:

```
merge [varlist] using filename ///
[, nolabel update replace nokeep _merge(varname)]
```

where `nokeep` instructs `merge` to ignore observations in the using dataset that do not have corresponding observations in the master dataset (by default, these observations would be added to the results of the merge and marked with `merge==2`). `merge(varname)` specifies the name of the variable that will indicate the source of the resulting observation (i.e., whether it comes from the master, the using data, or is common to both datasets). The default is `merge(_merge)`.

There must be one or more variables common to both datasets used for the `merge` command; these are called *identifier(s).* The identifier(s) must uniquely identify each observation.
Here are the meanings of the codes in `_merge`:

- `_merge==1`: observation only from the master, no merge, no update.
- `_merge==2`: observation only from the using file, no merge, no update.
- `_merge==3`: observation merged but not updated.
- `_merge==4`: observation merged and updated, meaning at least one missing value in the master has been updated with a non-missing value from the using data.
- `_merge==5`: observation merged but not updated, meaning the update was rejected. The result is the same as `_merge==3`, but there were updates in the using data inconsistent with what was already present, so the update was rejected.

note: which `_merge` codes should be eliminated must be decided on a case-by-case basis.

5.3 Collapse

`collapse` replaces the data in memory with a new dataset consisting of averages, medians, and other summary statistics for the specified variables. The syntax is:

```
collapse clist [weight] [if exp] [in range] [, by(varlist) cw fast]
```

where `clist` can be:

- `[(stat)] varlist [(stat)...]`
- `[(stat)] target_var=varname [target_var=varname...]  [(stat)...]`
- any combination of `varlist` or `target_var`, and `stat` is one of the following: `mean`, `sd` (standard deviation), `sum`, `rawsum` (sum ignoring any specified weights), `count` (number of non-missing observations), `max`, `min`, `median` , `p#` (#-th percentile), or `iqr` (interquartile range).

If stat is not specified, mean is assumed by default. The option by(varlist) specifies the groups over which means and other statistics are to be calculated. cw specifies cases to exclude (if not specified, all observations are considered), and fast indicates that you can return to the original dataset by pressing Break while collapse is being executed.

5.4 Contract

contract creates a dataset of frequencies. It replaces the data in memory with a new dataset consisting of all combinations of variables in varlist present in the data, along with a new variable containing the frequency of each combination. The syntax is:

```
contract varlist [weight] [if exp ] [in range] ///
[, freq(varname) zero nomiss]
```

where

- freq(varname): specifies the name of the frequency variable. If not specified, STATA automatically creates a variable named _freq, provided that name isn't already in use.
- zero: Specifies that combinations with zero frequency should be included.
- nomiss: Specifies that observations with missing values in any of the variables in varlist should be excluded. If not specified, all possible observations are used.

For example, the command contract var1 var2 var3 would create a dataset with all unique combinations of values from var1, var2, and var3, along with a _freq variable indicating the count of each combination.

5.5 Expand

The expand command replaces each observation in the current dataset with n copies of that observation, where n is the integer value of the specified expression (if the expression is less than one or missing, it is interpreted as one, and the observation is not duplicated). Here is the syntax:

```
expand [=]exp [if exp] [in range]
```

where

- `exp`: specifies the expression that determines how many times each observation should be expanded.
- `[if exp]`: optional condition specifying which observations to expand based on a condition.
- `[in range]`: optional range specifying which observations within a range should be expanded.

For example, the command `expand 2` would duplicate each observation in the dataset twice. If you use `expand = 0.5`, each observation would effectively be duplicated once, as the integer part of 0.5 is 0.

5.6 Fillin

The `fillin` command rectangularizes a dataset by adding observations with missing data so that all interactions of variables in `varlist` exist. This command also adds the `_fillin` variable to the data, which equals 1 if an observation was created and 0 otherwise. Here is the syntax:

```
fillin varlist
```

For example, if you have observations where `gender=1` and `perc=1`, `gender=1` and `perc=0`, but only `gender=2` and `perc=0` exist, this command will add a record for `gender=2` and `perc=1`.

The `_fillin` variable helps identify which observations were added by the `fillin` command to complete the rectangular structure of the dataset based on the combinations specified in `varlist`.

Exercises

1. Append the dataset `auto2.dta` to the dataset `auto.dta`.

2. Merge the dataset `auto.dta` with `auto2.dta` by the variable `make`.

3. Create a dummy variable for cars that are foreign and count how many they are.

4. Collapse the dataset to show the mean `price` and `mpg` by foreign.

5. Create a summary dataset of `auto.dta` that includes the number of observations, the minimum, maximum, and mean values of `price` and `weight` by `foreign`, and label all the variables.

CHAPTER 6

Estimation

Estimation commands are used to perform statistical analyses and model fitting. These commands typically follow a standard syntax, and various options can be applied to customize the estimation process. The basic syntax is:

```
[by varlist:]  command yvar xvarlist [if exp] [in range] [, options]
```

where

- `by varlist:` repeats the command for each group defined by `varlist`.
- `command` is the estimation command (e.g., regress for linear regression).
- `yvar` is the *dependent variable*.
- `xvarlist` is the list of *independent variables*.
- `if exp` restricts the estimation to observations that satisfy the condition `exp`.
- `in range` restricts the estimation to observations within the specified `range`.
- `, options` are additional optional options to customize the estimation.

The commonly used options are:

- `robust`: it calculates robust standard errors to correct for heteroskedasticity using White's correction.

```
command y x1 x2, robust
```

- `level(#)`: it sets the confidence level for confidence intervals. The default is 95%.

```
command y x1 x2, level(90)
```

6.1 OLS regression

The `regress` command is used to perform linear regression, estimating the relationship between a dependent variable and one or more independent variables.

```
regress yvar xvarlist [if exp] [, robust]
```

where

- `yvar` is the *dependent variable*.
- `xvarlist` is the list of *independent variables*.
- `if exp` restricts the estimation to observations that satisfy the condition `exp`.
- `, robust` specifies that robust standard errors should be used to account for heteroskedasticity.

For example,

- `regress income education experience` performs a linear regression where `income` is the dependent variable, and `education` and `experience` are the independent variables.

- `regress income education experience if age > 30` restricts the regression to individuals older than 30.
- `regress income education experience, robust` performs the same regression but adjusts the standard errors to be robust to heteroskedasticity.
- `regress income education experience if age > 30, robust` restricts the regression to individuals older than 30 and uses robust standard errors.

The `regress` command is a powerful tool for performing linear regression analysis. By using the `robust` option, you can ensure that the standard errors of your estimates are reliable even if the assumption of homoskedasticity is violated. This makes your statistical inferences more robust and trustworthy.

6.2 Predicted values and residuals

The `predict` command is used after an estimation command, such as `regress`, to generate predicted values and residuals. These generated variables can be useful for further analysis and diagnostics. The basic syntax is:

```
predict newvar [, statistic]
```

where

- `newvar` is the name of the new variable that will store the predicted values or residuals.
- `statistic` specifies the type of prediction or residual to calculate. Common statistics include `xb` (predicted values) and `residual` (residuals).

When no statistic is specified, Stata predicts the values of the dependent variable using the fitted model (`xb`). If `residual` is specified, Stata computes the difference between the observed and predicted values of the dependent variable.

The `predict` command is a versatile tool for generating predicted values and residuals following a regression or other estimation commands. By using different statistics, you can obtain various types of predictions and diagnostics, which are essential for model evaluation and further analysis.

6.3 Hypothesis testing

Hypothesis testing is a fundamental aspect of statistical analysis, and Stata provides several commands for testing differences in means between variables or groups.

Testing Differences in Means Using t-tests

Paired t-test

This test compares the means of two related groups (e.g., measurements before and after an intervention).

```
ttest var1 = var2
```

Two-Sample t-test

This test compares the means of a variable between two independent groups.

```
ttest var, by(groupvar) [unequal]
```

where

- `by(groupvar)` specifies the grouping variable.
- `unequal` specifies not to assume equal variances between the two groups.

Testing Differences in Multivariate Means Using Hotelling's T-squared Test

This test compares the means of multiple variables simultaneously between two groups.

```
hotelling varlist, by(groupvar)
```

where

- `varlist)` is a list of variables whose means are to be compared.
- `by(groupvar)` specifies the grouping variable.

Stata offers a robust set of commands for hypothesis testing, allowing users to test differences in means between related groups, independent groups, and across multiple variables simultaneously. These tests provide critical insights into whether observed differences in sample data are statistically significant.

Linear Hypothesis Testing (Wald Test) After Regression

The `test` command performs Wald tests of linear hypotheses about the coefficients after a regression model. While the `regress` command provides overall F-tests and individual t-tests, the `test` command allows for more specific hypothesis testing.

Testing Equality of Coefficients

```
test coefficient1 = coefficient2
```

Testing Linear Combinations of Coefficients

```
test linear_combination
```

Testing Sets of Coefficients

```
test coefficientlist
```

The `test` command is a powerful tool for hypothesis testing after a regression. It allows for testing the significance of single coefficients, sets of coefficients, and complex linear combinations. This flexibility makes it an essential part of regression analysis, enabling researchers to rigorously evaluate their hypotheses.

> *note:* for `test`, both `varname` and `_b[varname]` denote the coefficient on `varname`.

6.4 Instrumental Variables

Instrumental variable (IV) regression is used when one or more of the explanatory variables (endogenous variables) are correlated with the error term, which can lead to biased and inconsistent estimates. The `ivregress` command allows you to estimate a linear regression model using instrumental variables. The basic syntax is:

```
ivregress 2sls yvar [exogvarlist] (endogvarlist = IVvarlist)
```

where

- `yvar` is the dependent variable.
- `exogvarlist` is the list of exogenous (independent) variables.
- `endogvarlist` is the list of endogenous variables suspected to be correlated with the error term.
- `IVvarlist` is the list of instrumental variables used to instrument the endogenous variables. These instruments should be correlated with the endogenous variables but uncorrelated with the error term.

`ivregresss` allows the optional command `robust` to correct for heteroskedasticity. For example, suppose you want to estimate the effect of `education` (`endogvarlist`) on `wages` (`yvar`), but `education` is endogenous. You have other exogenous variables like `experience` and `gender` (`exogvarlist`), and you use `parental_education` and `distance_to_the_nearest_college` (`IVvarlist`) as instruments.

```
ivregress 2sls wage (education = parental_education //
distance_to_the_nearest_college) experience gender
```

This command estimates the effect of `education` on `wages` using `parental_education` and `distance_to_the_nearest_college` as instruments for `education`, while controlling for `experience` and `gender`.

In the first stage of 2SLS, regress the endogenous variable (`education`) on the instruments (`parental_education` and `distance_to_the_nearest_college`) and the exogenous variables (`experience` and `gender`), and then predict the values of `education` from this regression. In the second stage, regress the dependent variable (`wage`) on the predicted values of `education` from the first stage, along with the exogenous variables (`experience` and `gender`).

Instrumental variable regression using the `ivregress` command helps address endogeneity issues by providing consistent estimates. The process involves identifying suitable instruments that are correlated with the endogenous variables but not with the error term, ensuring valid estimation of causal relationships.

6.5 Logit Model

The `logit` command estimates the coefficients of a logistic regression model using maximum likelihood estimation. This model predicts the probability that the dependent variable equals 1, given the independent variables.

```
logit depvar [indepvars] [ f exp] [in range] [, options]
```

where

- `depvar` is the binary dependent variable.

- `indepvars` is the list of independent variables.
- `if exp` specifies conditions for the observations to be included.
- `in range` specifies a range of observations to include.
- `, options` are additional options for the logistic regression model.
  - `, robust` computes robust standard errors to correct for heteroskedasticity.
  - `level(#)` sets the confidence level for the confidence intervals of the coefficients (default is 95%).

Marginal Effects

The `dlogit` command, or difference in logits, is similar to the `logit` command but focuses on reporting marginal effects instead of coefficients. It shows the change in the probability for an infinitesimal change in each continuous independent variable and the discrete change in the probability for dummy variables.

```
dlogit depvar [indepvars] [if exp] [in range] [, options]
```

where

- `depvar` is the binary dependent variable.
- `indepvars` is the list of independent variables.
- `if exp` specifies conditions for the observations to be included.
- `in range` specifies a range of observations to include.
- `, options` are additional options for the logistic regression model.
  - `, robust` computes robust standard errors to correct for heteroskedasticity.
  - `level(#)` sets the confidence level for the confidence intervals of the coefficients (default is 95%).

For example, suppose you have a binary dependent variable `purchase` (1 if a purchase was made, 0 otherwise) and independent variables `income` and `age`.

```
logit purchase income age, robust // estimate a logistic regression model with robust standard errors
dlogit purchase income age, robust // obtain marginal effects instead of coefficients.
```

The `logit` command is used to estimate the coefficients of a logistic regression model, which predicts the probability of a binary outcome. The `dlogit` command, on the other hand, provides marginal effects, showing the change in probability for small changes in continuous variables or discrete changes for dummy variables. Both commands offer options for robust standard errors and setting confidence levels, making them flexible tools for binary outcome analysis.

6.6 Probit Model

The `probit` and `dprobit` commands are used for analyzing binary outcome models using a probit regression approach. These commands are similar to `logit` and `dlogit` but are based on a different underlying distribution (normal distribution for `probit` versus logistic distribution for `logit`).

The `probit` command estimates the coefficients of a probit regression model using maximum likelihood estimation. This model predicts the probability that the dependent variable equals 1, given the independent variables, assuming that the error terms are normally distributed.

```
probit yvar xvarlist, robust
```

where

- `yvar` is the binary dependent variable.

- `xvarlist` is the list of independent variables.
- `, robust` requests robust standard errors to correct for heteroskedasticity.

For example,

```
probit outcome age income education, robust
```

estimates a probit model where `outcome` is the binary dependent variable, and `age`, `income`, and `education` are the independent variables. Robust standard errors are used.

Marginal Effects

The `dprobit` command estimates the same probit model as `probit` but reports marginal effects instead of coefficients. It shows the change in the probability for an infinitesimal change in each continuous independent variable and the discrete change in the probability for dummy variables.

```
dprobit yvar xvarlist, robust
```

where

- `yvar` is the binary dependent variable.
- `xvarlist` is the list of independent variables.
- `, robust` requests robust standard errors to correct for heteroskedasticity.

For example,

```
dprobit outcome age income education, robust
```

estimates the same model as `probit` but reports the marginal effects, i.e., the change in the probability of `yvar` for an infinitesimal change in each continuous independent variable and the discrete change for dummy variables.

The `probit` command is used to estimate the coefficients of a probit regression model, which predicts the probability of a binary outcome assuming normally distributed error terms. The `dprobit` command provides marginal effects, showing the change in probability for small changes in continuous variables or discrete changes for dummy variables. Both commands offer options for robust standard errors and setting confidence levels, making them flexible tools for binary outcome analysis.

6.7 Post-Estimation: Publication-Ready Tables

While the standard Stata output is excellent for analysis, it is rarely in a format ready for academic papers or professional reports. To produce high-quality tables that can be exported to formats like LaTeX, Word, or Excel, you need **user-written commands**.

Installing User-Written Commands (SSC)

The Statistical Software Components (SSC) archive hosts thousands of community-developed Stata programs. To install a package, use the `ssc install` command. This only needs to be done once per Stata installation.

```
ssc install estout // Installs the estout package for estimation output
```

You can then use the new commands like any built-in Stata command.

Creating Tables with `estout`

The `estout` command (and its wrapper `esttab`) is widely used to compile and format estimation results (such as those from `regress`, `probit`, etc.) into a single, clean table.

1. Run the estimation and save the results using `estimates store`.

2. Use `estout` or `esttab` to format and export the stored results.

```
sysuse auto, clear

regress price mpg weight

estimates store Reg1

regress price mpg weight foreign

estimates store Reg2

esttab Reg1 Reg2 using "results/mytable.tex", replace se nogaps
```

This sequence stores the output of two regressions (`Reg1` and `Reg2`) and then combines them into a single `.tex` file, ready to be included in a LaTeX document.

Exercises

1. Using the dataset `auto.dta`, perform a linear regression with `price` as the dependent variable and `mpg` and `weight` as independent variables.

2. Generate predicted values after running the regression from Exercise 1.

3. Generate residuals after running the regression from Exercise 1.

4. Test the hypothesis that the coefficient of `mpg` is equal to the coefficient of `weight`.

5. Perform a robust regression with `price` as the dependent variable and `mpg` and `weight` as independent variables.

CHAPTER 7

Graphs

Data visualization is an essential step in data analysis, helping to explore data, check assumptions, and present results. Stata's graph system is powerful and highly customizable, all revolving around the core command: `graph`.
The general syntax for creating a graph is:

```
graph type varlist [if] [in] [, options]
```

The `graph type` specifies the kind of visualization (e.g., `twoway`, `histogram`, `bar`).

7.1 Histograms: Distribution of a Single Variable

A histogram displays the frequency distribution of a single continuous variable.

```
sysuse auto, clear
histogram price, normal
histogram price, discrete percent
```

The `normal` option overlays a normal density curve on the histogram, useful for assessing the variable's distribution.

7.2 Twoway Graphs: Relationships Between Variables

The `twoway` command is the umbrella for scatter plots, line plots, and fitted lines, used to visualize the relationship between two variables.

Scatter Plots

The most common two-way graph is the scatter plot, which shows the correlation between two variables.

```
twoway (scatter mpg price)
```

Fitted Lines (Regression)

To overlay a fitted regression line on a scatter plot, you combine the `scatter` plot type with the `lfit` (linear fit) plot type using parentheses:

```
twoway (scatter mpg price) (lfit mpg price)
```

7.3 Bar Charts: Summarizing Categorical Data

Bar charts are ideal for displaying summary statistics (like means) by category.

```
graph bar (mean) price, over(foreign) ylabel(0(5000)15000) title("Mean
Price by Car Origin")
```

This command plots the mean of the `price` variable, grouped by the `foreign` variable.

7.4 Saving and Exporting Graphs

Stata saves graphs in its proprietary `.gph` format. To use them in a document, you must export them, preferably as a vector graphic like PDF.

```
graph export "charts/price_hist.pdf", replace
```

Exporting as **.pdf** or **.eps** is recommended for high-quality, scalable graphics in academic papers.

Exercises

1. Using the dataset `auto.dta`, create a histogram for the variable `weight`. Overlay a normal distribution curve on the histogram.

2. Generate a scatter plot showing the relationship between `mpg` (on the x-axis) and `price` (on the y-axis). Then, modify the graph to include a linear fitted regression line (`lfit`).

3. Create a bar chart that displays the mean of the variable `weight`, broken down by the categorical variable `foreign`.

4. Export the scatter plot created in Exercise 3 to a PDF file named `mpg_price_fit.pdf` in a folder named `charts`.

5. Create separate histograms for the variable `length` for domestic and foreign cars by using the `by` option.

References

Anderson, Mike - *An Introduction to Stata*

Coda Moscarola, Flavia - *Mini STATA COURSE*

Iezzi, Elisa - *STATA TUTORIAL: Primi passi con STATA -Gestione dati, grafici e semplici modelli-*

Sianesi, Barbara - *Basic Stata*