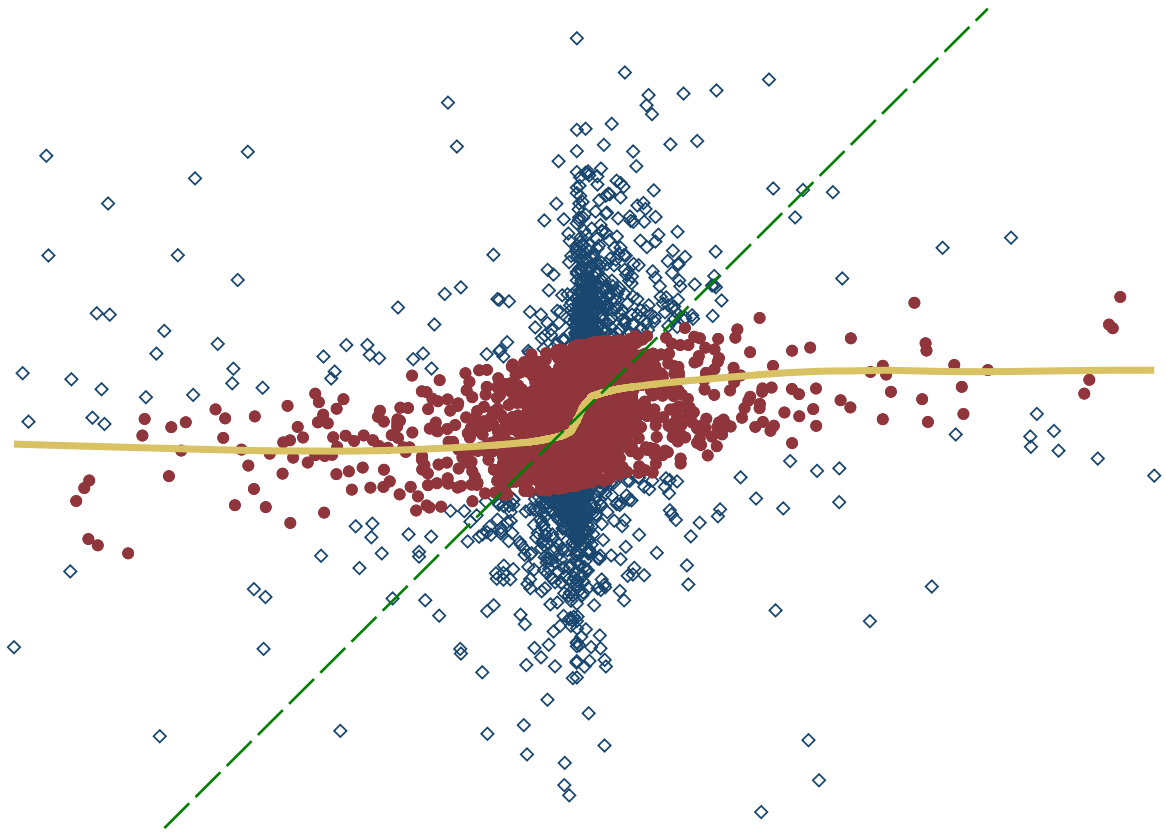
Introductory Guide to Using Stata in Empirical Financial Accounting Research

September 2023



David Veenman
Amsterdam Business School
University of Amsterdam

Contents

1 Introduction **3**

2 Getting started **6**
2.1 Program layout . . . 6
2.2 Using the Stata help function . . . 8
2.3 Opening, adding, and saving data files . . . 8
2.4 A note on decimal separators . . . 10
2.5 Generating and dropping variables . . . 11
2.6 Identifying and eliminating duplicate observations . . . 13
2.7 Conditional statements . . . 15
2.8 Managing variables . . . 17
2.9 Variable types, precision, and storage . . . 19
2.10 Ranking numeric data . . . 23
2.11 Working with dates . . . 24
2.12 Panel data . . . 27
2.13 Introduction to loops . . . 28
2.14 Using do-files . . . 32
2.15 Local and global macros . . . 34
2.16 Identifying local directories and installing new programs . . . 36

3 Advanced data management **39**
3.1 Outliers and scaling . . . 39
3.2 Transposing data . . . 44
3.3 Appending data . . . 45
3.4 Merging data . . . 46

4 Data analysis **49**
4.1 Descriptive statistics . . . 49
4.2 Basic regression estimation . . . 54
4.3 "Robust regression" estimation . . . 56
4.3.1 Reducing bias with robust estimators . . . 57
4.3.2 Increasing precision with robust estimators . . . 61
4.3.3 Getting the standard errors right with robust estimators . . . 63
4.4 Storing statistics and results . . . 66
4.5 Standard error adjustments . . . 70
4.5.1 Technical notes on programs for two-way clustering . . . 73
4.5.2 The restricted wild cluster bootstrap . . . 77
4.6 Fixed effects estimation . . . 80
4.7 Difference-in-differences (DiD) . . . 88
4.7.1 Simple DiD settings . . . 88
4.7.2 Testing for parallel trends . . . 92

4.7.3 "Staggered" DiD settings . . . 97
4.8 Propensity score matching . . . 102
4.8.1 Matching without replacement . . . 103
4.8.2 Matching with replacement . . . 112
4.9 Entropy balancing . . . 119
4.10 Exporting results . . . 124
4.11 Creating graphs . . . 125

5 Simulation and programming **133**
5.1 Illustration of using Stata for simulation analysis . . . 133
5.1.1 Example: quantifying omitted variable bias . . . 134
5.1.2 Example: quantifying bias in standard error estimates . . . 138
5.1.3 Example: bootstrapping standard errors . . . 140
5.2 Creating and using programs in Stata . . . 147
5.3 Creating ado-programs and using Mata . . . 150
5.3.1 Simple example . . . 150
5.3.2 Example including Mata . . . 157

A Applications and examples **169**
A.1 Using Stata to obtain data from WRDS . . . 170
A.2 Estimating discretionary accruals . . . 176
A.3 Estimating Dechow-Dichev's accrual quality measure . . . 185
A.4 Obtaining regression output without a loop . . . 191
A.5 Transforming data from annual to monthly . . . 194
A.6 Computing stock returns from Compustat Global data . . . 196
A.7 Computing Fama-French factor-model abnormal returns . . . 201
A.8 Estimating implied cost of equity capital . . . 206
A.9 Computing marginal effects . . . 213
A.10 Computing earnings surprises using analyst forecast data . . . 215
A.11 Computing earnings surprises using individual forecasts . . . 218
A.12 Special case: measures of past analyst forecast pessimism . . . 221
A.12.1 Past consensus forecast pessimism measure . . . 221
A.12.2 Past individual forecast pessimism measure . . . 223

Chapter 1

Introduction

The objective of this guide is to assist BSc/MSc students, PhD students, and junior researchers in using Stata for empirical archival research. Stata is a powerful program that can be used to analyze many different research questions in the fields of accounting, finance, economics, and beyond. While many statistical packages exist, a major advantage of Stata is that it allows you to carefully manage your data and research process, compute key variables needed in empirical research (e.g., fitted or residual values from a prediction model), and easily merge large sets of data (e.g., combining financial statement data from Compustat with stock market data from CRSP and analyst forecast data from IBES). The purpose of this guide is to give students a head start in using Stata in empirical accounting and finance research settings.

This guide is by no means complete, nor a guide to doing good research. Rather, it is simply a reflection of my experience with Stata over the years in trying to solve many different empirical issues. The examples I provide in this guide are therefore biased towards my own research focus. Stata also has many more useful features than those described in this guide, and often there are more ways to tackle a particular issue in Stata (note that the solutions presented here are not necessarily the best or most efficient). The Stata `help` function provides extremely detailed and helpful documentation and should be used as the main reference. This guide is not exhaustive and should only be used as *supplemental* information for researchers who start to work on empirical research.

In addition, this is not a guide to econometrics. Please refer to introductory statistics and econometrics textbooks for your questions in this area. Excellent examples are Stock and Watson [2018], Wooldridge [2019], and Hansen [2022]. For an important and deep discussion on the fundamentals of causal inference, see Morgan and Winship [2015]. For alternative,

slightly less formal (but very useful) textbooks focused on causal inference, see Angrist and Pischke [2009] or Cunningham [2021]. For more specific discussions on the use of econometric methods in accounting and finance research settings, see for example Roberts and Whited [2012] and Verbeek [2021]. For a (much) more comprehensive guide on how to use Stata for applied econometrics, see Cameron and Trivedi [2010]. To learn more about Stata and Mata programming, see Baum [2015] and Gould [2018].

Also note that Stata has its own specific features and advantages and is not superior to other programming languages such as Python, R, or Perl in performing specific tasks. For example, virtually anything described in this guide can also be done in R. Also, although Stata contains functions to download data from the web or to parse through text, other languages are much more powerful and efficient in that respect. My experience is that research projects often rely on a combination of different programs. For example, in Leung and Veenman [2018] we relied on Stata to construct our initial sample, used Perl to download and parse 8-K filings from the SEC's EDGAR system, attached the results from the Perl analyses to Stata to perform additional data cleaning and analyses, and used Excel to present our results in nicely formatted tables and figures. Alternatively, being able to perform all of these tasks within one and the same program sounds ideal, but my experience is that this can also be somewhat inefficient.

Finally, as it has always been, this guide is free of charge. It is therefore also very likely to contain errors. As a result, the examples throughout the text and in the appendices should be read and used with considerable caution. Any errors you find can be posted as an 'issue' on the accompanying Github page (`https://github.com/dveenman/stataguide`) or emailed to me (d.veenman@uva.nl). Your comments are highly appreciated and I aim to regularly update the guide and the accompanying code repo on Github when issues are detected. Also, any suggestions for additions to future versions of this guide are also appreciated. However, please do not email me specific questions about the Stata coding and data for your research project. Thesis/dissertation questions should be addressed to the relevant advisor at your university.

Version history:

- Version 5.0: September 2023
- Version 4.1: May 2019
- Version 4.0: January 2019
- Version 3.0: December 2013

- Version 2.0: December 2011
- Version 1.0: December 2010

Some examples of the updates in version 5:

- Update to Stata 18;
- Publication of code on Github;
- Improved and expanded section on standard errors;
- Improved section of fixed effects estimation;
- Improved section on matching and added entropy balancing;
- Added section on robust regression estimators in Chapter 4;
- Added section on difference-in-differences estimators in Chapter 4;
- Added section on creating and formatting graphs in Chapter 4;
- Added a separate chapter on simulations and programming in Chapter 5.1;
- Added Appendix A.1 on the use of Stata for downloading of data from WRDS;
- Improved Appendix A.8 on implied cost of capital estimation;

Chapter 2

Getting started

2.1 Program layout

Figure 2.1 presents a screenshot of Stata (version 18) that is shown when the program is started. The **Output/results window** presents an overview of all output created by Stata, with each piece of output being displayed as a new line at the bottom of the window. The window shows both the commands processed (for example, when opening or saving a data file) and the output from programs and functions, such as summary statistics and regression results.

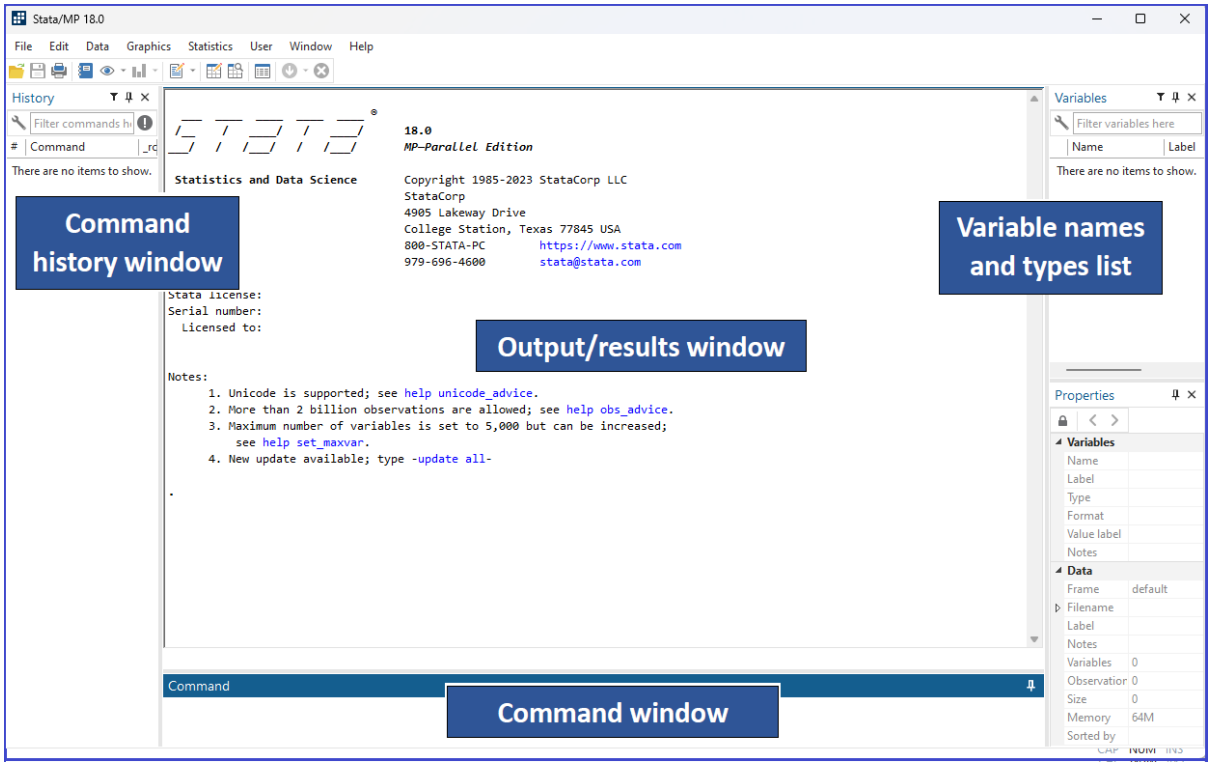


Figure 2.1: *Stata workwindow*

In the **Command window**, you can type in code and commands that Stata should process for you, for example to open or save a data file (e.g.: `use "C:\FOLDER\file.dta", clear`), to provide summary statistics for variables (e.g.: `sum price, d`), or to estimate a linear regression using OLS (e.g.: `reg price bps eps`). The **Command history window** provides a list of all the commands that were previously processed during the Stata session. By clicking on one of these commands in the **Command history window**, the command is automatically copied to the **Command window** and can be processed again. By double-clicking, the command will be re-executed immediately. Lastly, the window **Variable names and types list** lists all the variable names and their descriptions of the dataset that is currently in use.

Figure 2.2 provides displays the shortcut buttons available at the top of the Stata screen. In addition to using code and commands to let Stata perform certain actions for you, many actions can be performed using the shortcut buttons or the menus (File, Edit, Data, etc.) at the top of the screen. When starting to work with Stata for the first time, it will probably be useful to go through each of the menus and use the shortcut buttons in order get an idea of the commands associated with each of the actions. These commands will appear in the Stata **Output/results** and **Command history windows**.

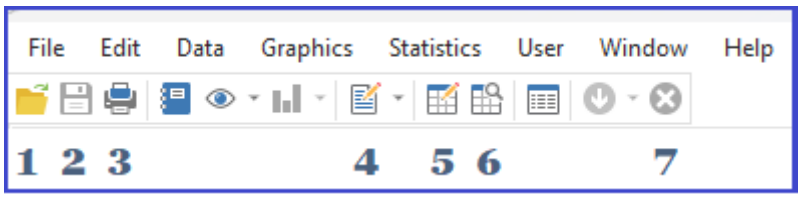


Figure 2.2: *Stata shortcuts*

With button **1** in the Stata shortcuts, we can open Stata data files. These files are in *.dta format, which is the format specific to Stata. Such *.dta files can be created from Stata directly after importing data from other sources or are directly obtained from data providors such as WRDS. Button **2** provides the option to save *.dta files. Button 3 is a print button. Button **4** opens the "Do-file" editor (more on this in Section 2.14). Button **5** opens the Data editor, while button **6** opens the Data browser. The main difference between the Data editor and Data browser is that in the Data editor you can change the data as in an Excel spreadsheet, while in the Data browser you can only view the data. Lastly, button **7** is a break button that may be useful when running a loop that needs to be terminated (for more information on loops, see Section 2.13).

2.2 Using the Stata help function

Stata has a very extensive help function that can be accessed through the Help menu, but also by typing:

```
help aaa
```

in the command window, where `aaa` refers to a specific Stata function. Try, for example, typing and executing:

```
help reg
```

to get information on how to execute the `reg` command to estimate a linear regression using ordinary least squares (OLS). Throughout this guide, I will provide references to the help function as follows: `[help reg]`, simply because Stata's help function is so extensive and typically contains the answers to all your questions.

2.3 Opening, adding, and saving data files

In some cases, Stata data files (*.dta) are directly available to the researcher, for example through the use of WRDS. In such cases, we can use the command:

```
use "C:\STATAguide\funda.dta", clear
```

where the example file `funda.dta` is stored in the location `C:\STATAguide\`. The command "`, clear`" is added to ensure Stata's memory is cleared before the file is loaded. Note that the `clear` command can also be used as a standalone command to restart with a clean Stata memory.

When the data are available in Stata format but the file is too large to load in Stata at once, we can add a conditional statement to the command (see more on this in Section 2.7). For example, we could add a conditional statement that specifies that only one year of data in a panel dataset should be loaded in Stata (note how the conditional statement appears *before* the comma):

```
use "C:\STATAguide\funda.dta" if fyear==2018, clear
```

In other cases, Stata data files might have to be created from other data files. One possibility is to use the File menu and choose Import. This allows you to import data from many other types of data files. Another possibility is to open the Data editor and simply paste data directly

from an Excel spreadsheet into Stata. When doing so, make sure that the first line in the Excel spreadsheet contains the variable names and that each column represents a variable (note that variable names in Stata cannot start with a number). This is an important step, since the data should be structured as blocks with columns representing variable and rows representing observations (data points).

As an example, insert the following information into an Excel spreadsheet. Next, open the Data editor in Stata, copy the block of data in Excel, and paste the data into Stata's Data editor:

```
firmid    name       year    assets
1         Ab Corp    2008     10000
1         Ab Corp    2009     11000
1         Ab Corp    2010     12000
2         Cd Inc     2008     59000
2         Cd Inc     2009     60000
2         Cd Inc     2010     61000
3         Ef & Co    2008      4500
3         Ef & Co    2009      4400
3         Ef & Co    2010      4300
4         Gh Corp    2008    100000
```

You will notice that Stata indicates "The first row on the Clipboard contains values that can be used as valid variable names" and asks whether "you want to treat the first row as variable names or data?". Because the first row contains the variable names, we should click the option "Variable names" in response to the question. Now you can see in the Data editor that the first line of data is converted to variable names, with the remaining lines being observations. Also, the data for the `name` variable are colored red, while the data of the remaining for variables are colored black. This is because `name` is recognized as a string variable, while the other variables are recognized as numeric (more on variable types in Section 2.8).

After pasting the data into the Data editor this way, the data are entered into Stata's memory. Note, however, that in contrast to Excel where we can have multiple worksheets in one file, Stata can only open one dataset (block of data) at a time. Consequently, is important to always save your data after making changes to the data and before opening a new dataset. Importantly, ensure you never overwrite the original files and always save your data to different filenames.

We can save *.dta files using the following command:

```
save "C:\STATAguide\firmyears.dta", replace
```

where the command "`, replace`" is added to ensure Stata overwrites the file when already in existence. Of course, the more cautious is option is to exclude this option to ensure the existing files are not overwritten. But that may be less useful in the common case where you want to rerun a long block of code in a do file, because the execution of the do file will stop when Stata tries to save a file to a name that already exists in the absence of the replace option (for more on do-files, see Section 2.14).

`[help use] [help clear] [help save] [help edit]`

2.4 A note on decimal separators

When copying data from another program, it is important to be aware of potential differences in the use of decimal separators. In Stata, decimals are separated by a dot (`.`) while thousands are separated by a comma (`,`), as is common in the US, UK, China, and Australia. In continental Europe, however, the convention is the other way around: decimals are separated by a comma, while thousands are separated by a dot. For example, three-and-a-half is written as 3.5 in the US or UK, while it is written as 3,5 in Germany and the Netherlands. The good news is that Stata is consistent and always uses the US/UK convention and it does not rely on thousands separators (unless this is used for formatting purposes). The bad news is that other programs, such as Excel, typically use the local convention for decimal separators, which means that a researcher in Germany might have trouble copying and pasting data directly from Excel into Stata (3,5 would be seen as a string instead of a number). Similarly, the researcher will have trouble importing data from a text (csv) file when the data in that file is created based on a different convention.

To circumvent these issues, first try to change your mindset to the US/UK convention and think of three-and-a-half as 3.5 instead of 3,5. Next, make sure that any data file you create or obtain from the web is also based on this same convention. If not, try changing the notation in the data (without altering any of the content). Finally, make sure that other programs such as Excel also think in the same way and change their settings. For example, in my version of Excel, clicking through File / Options / Advanced, I can change the decimal and thousands separators that Excel uses. Another option is to change these settings for Windows, or any other operating system, more broadly. Although for some researchers in some countries this may all seem a bit counterintuitive, it will save a lot of trouble along the way. Keep in mind

that this is also the convention that is used in virtually all of the academic papers you read.

2.5 Generating and dropping variables

Stata can generate new variables using very simple commands. For instance, in financial accounting research we often rely on ratios such as the return on assets, or accruals as a percentage of average total assets. The `gen` command allows us to compute a new variable called `roa` as follows:

```
gen roa=ni/lag_assets
```

where `ni` is a variable for net income and `lag_assets` is a variable reflecting the assets of firm `i` in year *t-1* (more on creating "lagged" variables in Section 2.12). Note that unlike other programming languages, we should explicitly invoke the `gen` functipn before listing the name of the new variable in the command. We can also perform more complex calculations, such as:

```
gen prob=exp(pv)/(1+exp(pv))
```

where `pv` is the input variable and `exp()` is an exponential function.

The `egen` function is a very useful and powerful tool that allows researchers to perform calculations beyond what is possible with the `gen` function. It can generate new variables based on summary measures, such as mean, min, max, sum, count, etc. For example, when using a panel dataset with multiple companies over multiple years, we can create a variable that counts the number of annual observations available per company (stored in a new variable named `countobs`):

```
egen countobs=count(year), by(gvkey)
```

where `year` is the variable for the calendar or fiscal year and `gvkey` is the variable that identifies the firm (GVKEY is Compustat's company identifier). We can also calculate the sum of profits made by each company over all the years in the sample (stored in `cumulative_profit`), as follows:

```
egen cumulative_profit=sum(ni), by(gvkey)
```

We can do the same to calculate total profits made by all firms for each year:

```
egen cumulative_profit2=sum(ni), by(year)
```

The ", by" can also be omitted. This will result in the summary measures being computed over the whole sample instead of different groups of observations.

When egen is combined with group(varlist), we can also create a new variable that takes on values 1, 2, ... for all the unique combinations available for varlist. For example, we may want to create a new variable that identifies each firm in our sample from 1 through n. This can be done as follows:

```
egen firmid1=group(gvkey)
```

or:

```
egen firmid1=group(coname)
```

Note that the group function cannot be combined with ", by". We can also use the group function to combine several variables into one variable. For example, we may want to create a new variable that captures every unique firm-year or unique firm-year-month in our sample:[1]

```
egen firmyearid=group(gvkey year)
```

and:

```
egen firmyearmonthid=group(gvkey year month)
```

We can replace existing variables by using the replace command. This command changes the contents of an existing variable:

```
replace year=year+1
```

And we can also generate lead and lagged variables from existing variables. For example:

```
gen lagyear=year[_n-1]
```

creates a new variable named lagyear that equals the value of variable year in the previous observation. We can also create a new variable named nextyear that equals the value of variable year in the observation one row below:

```
gen nextyear=year[_n+1]
```

Because we often need lagged and forward data, such as lagged total assets or next year's earnings, this function might seem helpful. However, this function is too simplistic as it does not take into account the possibility that the previous or next observation (row) may reflect a

[1] These are just a few examples of how to use egen in Stata. Many more options are available, see help egen. Moreover, the egenmore package, which can be installed via Stata's SSC repository (see Section 2.16), provides even more options. See https://ideas.repec.org/c/boc/bocode/s386401.html.

different company. Or worse, the data might be sorted in a way that is different from what we think. Therefore, we need an alternative way of creating lead and lag variables. The best way to do this is by using the `tsset` function, which is described in Section 2.12.

The `_n` function can be used to generate a variable equal to the row number in the data:

```
gen nr=_n
```

where `_n` refers to the row numbers 1 through n (note that `_N` refers to the total number of rows in the data). We can do the same for groups of observations, for example by creating a variable that captures the year number for each company:

```
gen yearnr=_n, by(gvkey)
```

Keep in mind that, again, the outcome of this function is critically affected by the way the data is sorted. Lastly, we can eliminate variables and observations from our data. With the following commands, we can eliminate or retain a specific set of variables:

```
drop var3
```

or:

```
keep var1 var2
```

2.6 Identifying and eliminating duplicate observations

Our data often contain duplicate observations, for a variety of reasons. As we typically do not want to double count and give too much weight to these duplicate observations, we should eliminate these them from the data. The `duplicates drop` command does the trick for us:

```
duplicates drop
```

The above statement eliminates duplicate observations in terms of all variables. Most often, however, we need more specific statements such as with the elimination of duplicate firm-*year* observations. In these cases, the command can be expanded to:

```
duplicates drop gvkey year, force
```

What is essential in using the `duplicates drop` command is that you understand why these duplicates exist, and to understand which of the duplicates will be eliminated from your dataset. For example, it is not uncommon to run into a dataset with duplicate observations by firm-year, even though observations do not contain the same information. This can happen,

for instance, when a firm changes its fiscal year-end month during the year 2018 and therefore may have two observations for 2018. The first observation might reflect accounting data from the annual report filed for the period that ended in June 2018 (the old fiscal year end), while the second observation might reflect accounting data from the second report filed for the period that ended in December of 2018 (the new fiscal year end). Both are relevant observations, but our data processes often require that we have a panel of data in which duplicate observations for combinations of firm and year cannot exist (for example when using the `tsset` command, see Section 2.12).

Another issue is that with `duplicates drop`, it is important to know which of the two (or more) duplicate observations gets eliminated from the data. Stata always keeps the first instance of the observations, which means that the sorting of the data is important. If, for example, the sorting of the data is random, we never know exactly which of the duplicates gets eliminated. Therefore, it is useful to first sort the data in a way that helps you better understand which of the duplicate observations gets eliminated. Moreover, it is helpful to actually inspect the data. One possibility is to use `duplicates report`, as follows:

```
duplicates report gvkey fyear
```

which gives us the following example output:

```
Duplicates in terms of gvkey fyear

--------------------------------------
   copies | observations       surplus
----------+---------------------------
        1 |        79442             0
        2 |         1664           832
        3 |           84            56
--------------------------------------
```

The output here suggests there are $1664 + 84$ observations that are flagged as duplicates in terms of `gvkey` and `fyear`. For 832 $(1664 - 832)$ *unique* firm-year combinations we see that there are two copies each, while for 28 $(84 - 56)$ there are even three copies each. Dropping the duplicates means we would eliminate $832 + 56$ observations from the sample.

Although `duplicates report` is helpful to understand how many duplicate observations exist in the data, it does not help us directly to inspect these observations and better understand why these duplicates exist. Because there are many different reasons for the existence of duplicates, and these reasons vary by the types and sources of data, we can use the following

set of commands to help me better inspect and understand the nature of these duplicates. For example, returning to the above example of duplicate firm-year combinations, we can use the following code that generates a new variable `dupid` that reflects the number of duplicates in each combination of `gvkey` and `fyear`:

```
gduplicates tag gvkey fyear, gen(dupid)
gsort dupid gvkey fyear
```

With this block of code, the bottom rows of the dataset now provide all the duplicate observations clustered together. Identify why these duplicates exist, decide which observations should be retained, and take action to ensure the `duplicates drop` command will only drop the observations you want to be eliminated (for example, by sorting the data in the right way). An alternative to using the `duplicates drop` command is to use conditional statements to drop specific observations to keep full control of the data elimination process. The deletion of observations using conditional statements is explained in Section 2.7.

`[help gen]` `[help egen]` `[help replace]` `[help drop]` `[help duplicates]`

2.7 Conditional statements

We often want to perform some actions on the data that apply only to a subset of observations. For this purpose we can use the `if` statement. For example, we may want to delete observations (rows) with missing data fields for a certain variable:

```
drop if ni==.
```

where "`.`" indicates the numeric field is empty. Alternatively, we can use:

```
drop if missing(ni)
```

Also note that "`==`" is used instead of "`=`", similar to other programming languages. This is required in all conditional statements and reflects the difference between the *evaluation* and the *assignment* of variables. For example, `a=1` means we assign a value of 1 to variable `a`. On the other hand, `a==1` means we evaluate whether the value of `a` equals 1. The outcome of this evaluation is boolean: if `a` equals 1, Stata internally returns the value "True"; if not, it returns the value "False". We can also let Stata delete the observations for which multiple variables have missing fields at the same time, for example by typing:

```
drop if ni==. & assets==.
```

Or we can let Stata drop those observations where at least one of the variables has a missing field:

```
drop if ni==. | assets==.
```

where the vertical bar (|) means "or". The "and" and "or" arguments can also be combined, but it is important to use parentheses in the proper way to deal with the precedense of the operators (for example, & takes precedense over |):

```
drop if (ni==. | assets==.) & gvkey==.
```

We can do the same with `keep` instead of `drop`. And we can use the `if` statement for situations where a variable should not be equal to a certain value, or greater/smaller than a certain value.

```
keep if dividend!=0
```

where "`!=`" means "not equal to". To give an example, we can use the greater/smaller-than functions to create dummy variables as follows:

```
gen lossdummy=0
replace lossdummy=1 if ni<0

gen profitdummy=0
replace profitdummy=1 if ni>0

gen dummy1=0
replace dummy1=1 if var1>=0

gen dummy2=0
replace dummy2=1 if var1<=0
```

Note that it is important to be very careful when using the greater-than command when variables have some missing values. The reason is that, internally, Stata stores missing values for numeric variables (`.`) simply as large numbers (more specifically, as a value larger than the maximum of the variable in the dataset). Therefore, when using a conditional statement, such as for the creation of `dummy1` as in the example above, be sure to check that `var1` does not any having missing values. Alternatively, we can add "`& var1!=.`":[2]

```
gen dummy1=0
replace dummy1=1 if var1>=0 & var1!=.
```

If we would not add this additional option, Stata would set `dummy1` equal to 1 for all missing values of `var1` as well, which is typically not what we want.

`[help if]`

[2] See the following link for more explanation on this issue: https://www.stata.com/support/faqs/data-management/logical-expressions-and-missing-values/.

2.8 Managing variables

Variables in Stata come in two main forms: string variables and numeric variables. String variables contain text elements, such as company names ("Alphabet Inc.") or company identifiers ("38259P508"). Numeric variables contain values (e.g., financial data) that we can use for calculations. In general, the variables in our data are already correctly set as strings or numeric variables. Sometimes, however, they are not. We can use the command:

```
tostring sic, replace
```

to change the variable `sic` (which reflects a 4-digit numeric code for industry membership) from a numeric variable into a string variable. We can also reverse the action by changing a variable from a string to a numeric variable:

```
destring gvkey, replace
```

The `tostring` command can always be used to change numeric variables into string variables. However, `destring` can only be used when the string variable contains numeric values only. For variables containing combinations of letters and numbers (such as "38259P508"), this command can not be used.

Sometimes we need a string variable that contains only a component of a longer string variable. For example, we may need to create a variable that captures a 2-digit instead of a 4-digit industry code. In such cases, we can use the `substring` function:

```
gen sic2=substr(sic,1,2)
```

where `sic` is the original 4-digit string variable, 1 denotes the location where to start, and 2 indicates the length of the string to be returned. Another useful string command is `reverse`, which can be used to generate a new variable equal to the mirror image of the original variable (for example, "38259P508" becomes "805P95283"). The `length` function can be used to return the length of a string variable into a numeric variable. Using the `concat` function in combination with `egen` allows us to combine two string variables into one new variable, for example:

```
egen cusip8=concat(cusip6 cusip2)
```

where we combine two string variables `cusip6` (6-digit CUSIP company identifier) and `cusip2` (2-digit addition to the CUSIP company identifier) into one new string variable (the 8-digit CUSIP company identifier).

When generating or replacing a string variable, we should always use quotes (`""`) with the input. For example:

```
gen shareid="A" if iid=="01W"
replace shareid="B" if iid=="02W"
```

Names of variables can be changed using the `rename` command (or shorter: `ren`). For example:

```
rename coname CompanyName
```

changes variable name `coname` to `CompanyName`. Note that variable names in Stata are case-sensitive. That is, it matters whether you use lowercase or uppercase letters. To ensure this does not lead to errors or confusion, it might be useful to change variable names to lowercase letters using the following command:

```
rename CompanyName, lower
```

which leads to the variable name `companyname`. To do this for all variables in the data, we can simply use the asterisk:

```
rename *, lower
```

In a similar vein, the contents of a string are case sensitive. For example, let's say variable `name1` contains `AJAX`, while variable `name2` contains `Ajax`, a comparison of `name1==name2` would return "False". If instead we want the variable contents to be case *in*sensitive, we can change the variables' contents by replacing all capital letters with lower case letters:

```
replace name1=lower(name1)
replace name2=lower(name2)
```

Alternatively, we can also use `upper()` to change all content to capital letters. Lastly, we can easily sort data in Stata using the `sort` command:

```
sort coname
```

This command sorts the data based on company name (if the variable name is `coname`). We can also sort on multiple variables at the same time, for example to order all available years per company in the data, by typing:

```
sort coname year
```

`[help tostring]` `[help destring]` `[help substring]`

`[help reverse]` `[help length]` `[help sort]` `[help compress]`

2.9 Variable types, precision, and storage

Besides the distinction between numeric and string variables, variables can have different "types". For example, in a sample that is downloaded from Compustat Global in Appendix A.1, the string variables `gvkey` and `fic` have types `str6` and `str3`, respectively. The types `str6` and `str3` correspond to the maximum of six and three characters stored in these variables, respectively. In other words, they reflect the maximum amount of space that is needed for the variable to store all information, which means that type `str6` takes up more memory in Stata than `str3`. To identify the variable types, we can use the `describe` command (an alternative is to use `codebook`):

```
. describe gvkey fic

Variable      Storage   Display    Value
    name         type    format    label      Variable label
------------------------------------------------------------------------------
gvkey           str6    %9s
fic             str3    %9s
```

Also note the distinction between (storage) type and display format. The former relates to how the information is stored in memory, while the latter relates to how the information is displayed in Stata's output window or data browser/editor. The good news is that we often have to pay little attention to variable types, as Stata figures this out for us. Some exceptions are described below.

The precision of information storage can matter when working with numeric variables. This becomes important typically when variables contain values that can become very large or when working with non-integer numbers. When creating a new variable in Stata using `gen`, the default is that Stata stores this as a floating point numeric variable. The storage as floating point numbers is a way in which computers can store numbers with a high degree of precision. But this precision might not always be sufficient. Consider the following example, where we create a variable that contains a very large number (e.g., 100 billion dollars in market value). To be able to observe in the data editor/browser what the number is that Stata actually stores, we also adjust the format of this variable:

```
. gen num1=100000000000 // (i.e., 100,000,000,000 or 100 billion)

. format num1 %15.0g

. describe num1
```

```
Variable      Storage   Display    Value
    name         type    format    label      Variable label
------------------------------------------------------------------------------
num1            float   %15.0g
```

Even though the input value is exactly 100 billion, inspection of the variable in the data editor/browser suggests the value stored by Stata is actually 99999997952. This happens because of the way floating point numbers are stored and the maximum possible precision of this storage. To better understand the consequence of this difference, consider the following transformation of the variable. Even though the value of the transformed variable should be zero, it is not:

```
. gen num1adj=num1-100000000000

. sum num1adj

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
     num1adj |          1       -2048           .      -2048      -2048
```

This problem can be addressed by using a variable type that is more precise: `double` instead of `float`. When we add the qualifier `double` between `gen` and the variable name, we tell Stata to store the variable in double precision. Doing so, we can see in the data editor/browser that this variable *is* correctly stored as 100,000,000,000:

```
. gen double num2=100000000000

. format num2 %15.0g

. describe num2

Variable      Storage   Display    Value
    name         type    format    label      Variable label
------------------------------------------------------------------------------
num2            double  %15.0g
```

Similar issues arise for variables with fractional values. Consider the following example where we assign the value of 0.01 to variable `a`. Next, we create variable `b` and set its value equal to 1 when `a==0.01`. Because we assigned the value 0.01 to `a`, this is true and we should see that `b` is set equal to 1. But it is not:

```
. set obs 1

. gen a=0.01

. format a %15.0g
```

```
. gen b=1 if a==0.01
(1 missing value generated)
```

The explanation why `a==0.01` is not evaluated as true is that when we inspect the value of `a` in the data editor/browser, we can see that the value stored by Stata is actually 0.0099999997765. Because the difference between 0.0099999997765 and 0.01 is tiny, this is very unlikely to make a difference in our statistical analyses. However, it does affect the outcome of evaluations as in the example above. If we instead create `a` as a `double`, we do not encounter this issue.

Another example where this matters is the following. We create a new variable with 100,000 values randomly drawn from a uniform distribution between 0 and 1. Generating such a random variable can be useful as a tool to randomly sort our data, for example in simulation settings (for more discussion on simulations, see see Section 5.1). Because of the maximum possible precision of the storage of floating point variables, we see that even though all values were drawn independently and are not exactly equal, the truncation of values at a fixed number of digits causes the dataset to contain duplicate values for the variable we created:

```
. set obs 100000
Number of observations (_N) was 0, now 100,000.

. set seed 1234

. gen r1=runiform()

. duplicates report r1

Duplicates in terms of r1

--------------------------------------
Copies | Observations       Surplus
----------+---------------------------
     1 |        99596             0
     2 |          404           202
--------------------------------------
```

For instance, we can see that the values of the following observations are identical:

```
. list r1 if _n<=2

   +----------------------+
   |                   r1 |
   |----------------------|
1. | .02772381342947483063 |
2. | .02772381342947483063 |
   +----------------------+
```

This can become problematic when we sort the data on this random variable or when we rank the data. If, instead, we generate the variable as type `double`, we can see that we no longer have duplicates even though we generated the exact same sequence of random numbers:

```
. set seed 1234

. gen double r2=runiform()

. format r2 %40.0g

. duplicates report r2

Duplicates in terms of r2

--------------------------------------
   Copies | Observations       Surplus
----------+---------------------------
        1 |       100000             0
--------------------------------------
```

Looking at the exact same observations we identified above as duplicates, we can see that their values are now different:

```
. list r2 if _n<=2

   +-----------------------+
   |                    r2 |
   |-----------------------|
1. | .02772381282706171124 |
2. |  .0277238142757431083 |
   +-----------------------+
```

Finally, a useful function in the context of data storage is `compress`. When we work with very large datasets, Stata can become slow as it reaches memory constraints. The benefit of `compress` is that it allows us to reduce the amount of memory that is needed for the dataset, without changing the data and its precision. That is, `compress` evaluates whether variables can be stored in types that use up less memory without losing any information. The following example shows how a large dataset is successfully compressed using `compress`. Before the compression, the size of the dataset in memory is 1,867,604,968. After the compression, the size of the dataset in memory is 1,216,719,872:

```
. compress
  variable datadate was double now int
  variable fyear was double now int
  ...
  variable sich was double now int
```

```
  variable rank was double now byte
(650,885,096 bytes saved)
```

[help describe] [help codebook] [help compress]

2.10 Ranking numeric data

In many settings it is useful to rank observations in the data. For example, you might need to make portfolios of observations based on firm size to track the earnings or stock returns among firms of similar size. Another example where ranking is useful is when filtering data from outliers (see more on this in Section 3.1). The `xtile` command in Stata helps by creating a new variable that contains the rank of an observation for a specific variable, relative to all other observations in the full sample of data. Stata easily allows us to rank the data in any number of groups based on any variable of interest. Often, however, we rank observations into quintiles, (`q=5`), deciles (`q=10`) or percentiles (`q=100`). For example, to allocate observations into decile (ten) portfolios based on firm size, do:

```
xtile sizedecile=mv, nq(10)
```

where `mv` is a measure of firm size (stock market capitalization), which is the variable on which we are ranking, and `nq(10)` denotes the number of quantiles to be created. For example, let's say our dataset has 1000 observations. Using this command, we would assign a rank to each of the 1000 observations and create 10 portfolios of 100 observations each. Observations with rank 1 would represent the 100 observations with the smallest values of the variable (`mv` in this case), while observations with rank 10 would represent the 100 observations with the largest values of the variable. Depending on the existence and number of ties (observations with the same values), it is likely that the groups will not all contain exactly 100 observations.

After creating these quantiles and generating the new variable `sizedecile`, we can now, for example, compute a new variable such as the size-adjusted abnormal stock returns for a company over the period. That is, we can i) compute the average stock return for each portfolio and time-period combination, and ii) generate a new variable that is equal to the difference between the realized return and the average return of firms in the same portfolio during the same time period:

```
egen portfolioreturn=mean(ret), by(year month sizedecile)
gen abret=ret-portfolioreturn
```

This example generates a variable for abnormal stock returns based on firm-size-decile adjustments for each firm-month in the sample. In a similar vein, we can create new variables that contain an observation's *percentile* rank for a specific variable using `nq(100)`:

```
xtile xroa=roa, nq(100)
```

Assuming we have a set of firm-years and we want to create a percentile rank by year, we can also combine `xtile` with `if` statements as follows:

```
gen roa_perc=.
xtile xroa=roa if year==2015, nq(100)
replace roa_perc=xroa if year==2015
drop xroa
xtile xroa=roa if year==2016, nq(100)
replace roa_perc=xroa if year==2016
drop xroa
xtile xroa=roa if year==2017, nq(100)
replace roa_perc=xroa if year==2017
drop xroa
xtile xroa=roa if year==2018, nq(100)
replace roa_perc=xroa if year==2018
drop xroa
```

As shown in Section 2.13, this set of commands can be simplified and automated using `forvalues` loops.[3]

`[help xtile]`

2.11 Working with dates

Stata records dates as numbers. For example, December 31, 2009 is stored as the number 18262, which is the number of days elapsed since January 1, 1960. The advantage of storing dates numerically is that we can easily use the dates for the construction of other variables that are adjustments of these dates. For example, we might be interested in the time elapsed between a company's fiscal year end and its annual earnings announcement:

```
gen delay=rdq-datadate
```

where `rdq` is the date of the earnings announcement and `datadate` is the date of fiscal year end. Of course, this works only when both variables are stored as numeric variables.

[3] If the `gtools` package is installed in Stata, you can use the `gquantiles` function in combination with the `, by()` and `xtile` options. The `gtools` package can be installed with `ssc inst gtools`, see Section 2.16. For more information on `gtools`, see https://gtools.readthedocs.io/en/latest/.

A disadvantage of recording dates numerically is that when viewing the data, we see 18262 instead of December 31, 2009 and we do not directly know what date this number refers to. To solve this issue, we can let Stata display the date variable in date format, using the format command:

```
format datadate %d
```

where `datadate` is the date variable of interest and `%d` tells Stata to store the variable as a date variable. The Help files provide much more insights on the different ways to format variables.

Stata also allows us to directly extract the year, month, and day of a date variable, and store them into new variables. This can be achieved using the following commands:

```
gen yr=year(datadate)
gen mth=month(datadate)
gen dy=day(datadate)
```

We can also reverse the process and create a date variable from three variables that capture year, month, and day:

```
gen datadate=mdy(mth,dy,yr)
format datadate %d
```

Sometimes a date is stored as a string variable, for example as `12/31/2009`. In such a situation, we can extract the month, day, and year from this string variable using the `substr` function, then use `destring` on these variables to change them to numeric, and then combine the numeric variables into a new date variable. This is done as follows:

```
gen mth=substr(datadate,1,2)
gen dy=substr(datadate,4,2)
gen yr=substr(datadate,7,4)
destring mth dy yr, replace
gen datadate2=mdy(mth,dy,yr)
format datadate2 %d
```

Fortunately, Stata has simpler options to achieve the same objective with the `date` function. For example, when the `datadate` variable is formatted as `12/31/2009`, we can use the following command to create a new variable that contains the numeric value of the date:

```
gen datadate3=date(datadate, "MDY")
format datadate3 %d
```

Or when the `datadate` variable is formatted as `2009-12-31`:

```
gen datadate4=date(datadate, "YMD")
format datadate4 %d
```

In some datasets, strings may contain both dates and times combined. For example, you might encounter a variable that is formatted as `2010-01-02 17:00:00`. In this case, we may want to construct two separate numeric variables for date and time, respectively. To create the date-only variable, we can use the `date()` function again with the additional `#` sign to ignore information that comes after the day:

```
gen date_only=date(date, "YMD#")
format date_only %d
```

To create the time variable, we can use the `clock()` function on the variable. Because this results in a numeric variable with a very large value, it is essential that we generate the variable in double precision instead of floating-point precision (here, the value numeric value for `2010-01-02 17:00:00` equals 1578070800000, which is the number of milliseconds elapsed since January 1, 1960):

```
gen double time=clock(date, "YMDhms")
format time %tc
```

From this new time variable, we can also construct additional variables that capture the hours, minutes, and seconds, respectively. We can also combines these into a new variable that only contains the time of day without the date. As a final step, we can change the format of the latter variable to ensure its value is displayed as `17:00:00`:

```
gen hours=hh(time)
gen minutes=mm(time)
gen seconds=ss(time)
gen time_of_day=hms(hours, minutes, seconds)
format time_of_day %tcHH:MM:SS
```

Another function that might be useful here is `dofc()`, which can be used to extract the date from a numeric date-time variable such as the `time` variable created above:

```
gen date_only2=dofc(time)
format date_only %d
```

anntims

`[help date] [help format] [help datetime conversion]`

`[help datetime_display_formats]`

2.12 Panel data

In accounting and finance research, we often work with panel data, which means we have a sample that is a combination of cross-sectional (e.g., multiple firms) and time-series (e.g., multiple years per firm) data. Stata can use the panel structure of the data to perform time-series calculations and analyses. For example, we can use lag (`l.`) and lead (`f.`) functions to compute lagged and forward variables as follows:

```
gen roam1=l.roa
gen roam2=l2.roa
gen roap1=f.roa
gen lag_assets=l.assets
```

We can also use these functions to compute stock returns from daily stock prices and dividends:

```
gen ret=(price-l.price+div)/l.price
gen ret_alt=(d.price+div)/l.price
```

where `d.` denotes a difference operator. We can also assess the correlation between the current and lagged values of some variable (autocorrelation):

```
pwcorr roa l.roa
pwcorr roa l.roa l2.roa
```

This looks very similar to using the `[_n...]` function as described in Section 2.5, but there are two main advantages. First, after correctly specifying the panel structure of the data (see below) we do not run into the problem that a lead or lag in a variable can capture a value of a different company. Second, the use of leads and lags as specified above is not affected by the way the dataset is sorted, in contrast to the use of the `[_n...]` function.

Before we can actually use the lead and lag functions described above, we should inform Stata about the panel structure of our data. This is done with the `tsset` command. Before using this command, we should ensure we have two numeric variables that capture all unique values of firms (1...n) and time (1...t), respectively. To achieve this, first create new variables `firmid` and `timeid`. When the dataset consists of firm-years, do the following:

```
egen firmid=group(gvkey)
egen timeid=group(year)
```

When the dataset consists of firm-months, such as with stock price/return data, do:

```
egen firmid=group(gvkey)
egen timeid=group(year month)
```

When the dataset consists of firm-days, such as with stock price/return data, we should first create a variable that captures all unique trading days in the data. If we would not do this, we run the risk that when computing a daily stock return, as in the example above, we may be comparing the closing stock price on Monday with the closing stock price on Sunday (which, of course, does not exist), instead of comparing the closing stock price on Monday with the closing stock price on Friday. Accordingly, when using daily data, make sure to create a new variable (`dateid`) that only captures trading days:

```
egen dateid=group(date)
egen firmid=group(gvkey)
egen timeid=dateid
```

After creating the firm and time identifiers, we can now teach Stata the panel structure of the data using the `tsset` command:

```
tsset firmid timeid
```

An example of the output shown on the **Output/results window** is presented below. As can been seen in this example, the panel is "unbalanced", in the sense that not every firm has the maximum number of (here: 10) yearly observations. This is very common in the empirical data we encounter, as the analyses are generally focused on firms that survive as well as those that drop out (e.g., go bankrupt or are acquired) during the sample period.

```
. tsset firmid timeid
        panel variable:  firmid (unbalanced)
         time variable:  timeid, 1 to 10, but with gaps
                 delta:  1 unit
```

Note that the `tsset` command will return an error when not all firm-time combinations are unique. Therefore, we should first use the `duplicates drop` command before using `tsset`. Before doing so, however, be careful to first understand why duplicates exist in the data and to not throw out valuable data.

`[help tsset]`

2.13 Introduction to loops

In many research settings it is helpful and time-efficient to use loops. For example, in the case of assigning observations to portfolios as in Section 2.10, it may be necessary to assign

firms to portfolios on a periodic basis. As in the case of calculating size-adjusted abnormal stock returns, it is common to assign firms in size deciles based on the market values of these firms at the beginning of the calendar year (or month), instead of assigning observations to portfolios based on the full sample period.

Loops can be performed using different commands, but we will focus on the `forvalues` command here. `forvalues` repeatedly sets a local macro to each element of a range and executes the commands enclosed in braces "`{...}`". The loop is executed zero or more times. `forvalues` provides a fast way to execute a block of code for different numeric values of a local macro. In the following example code, we run a loop over a range of 1...100 for a local macro variable `i`. The command is to simply display (in the **Output/results window**) the value of `i` for every `i` in the range of 1...100 with steps of 1:

```
forvalues i=1(1)100{
    di `i'
}
```

Note the way in which the index of the loop is referred to in the code: `` `i' ``. That is, the index `i`, which is a local macro, is "called" by enclosing it using a backtick on its left (`` ` ``) and using a single quote (`'`) on its right side. The same procedure applies to any other local macro that we may want to use in our code. Also note the indentation (tab/spaces) before the command that is included within the braces of the loop. This is not required, but it is highly recommended to make it easier to read the code. This is especially important when the code becomes more complex and, for example, has nested loops. We can do the same with steps of 2 instead of steps of 1:

```
forvalues i=1(2)100{
    di `i'
}
```

Or we can start and end at different values:

```
forvalues i=100(1)200{
    di `i'
}
```

In case we take steps of 1, the code can also be simplified to:

```
forvalues i=1/100{
    di `i'
}
```

We can put any command within these braces. For example, where loops often come in handy is when we want to estimate a large number of regressions for different subsets of our data. For example, we may want to estimate a regression for each year in the sample:

```
forvalues i=1995(1)2010{
    reg price bps eps if year==`i'
}
```

While in this example we have manually entered the starting and ending values of `i` in the range (i.e., 1995 and 2010, respectively), we can also let Stata determine the starting and ending values based on minimum and maximum observations in our data. This allows our code to become more generic and not dependent on the specific dataset we use. To achieve this, we need to use the `summarize` command (explained later in Section 4.1) and the summary statistics stored in local macros (see Section 4.4 for more information on the contents of the `r(min)` and `r(max)` variables):

```
sum year
local m=r(min)
local n=r(max)
forvalues i=`m'(1)`n'{
    reg price bps eps if year==`i'
}
```

Now the loop runs over a variable range of years identified in the data.

When running large or long loops, we are not always interested in seeing the steps and output of each iteration. Therefore, we can also add the `quietly` (or short: `qui`) prefix. When doing so, however, it might be useful to include a display command (`di`) to be able to see at which iteration Stata currently is when running the loop:

```
...
forvalues i=`m'(1)`n'{
    qui reg price bps eps if year==`i'
    di `i'
}
```

Let's go back to the example of the calculation of monthly abnormal stock returns, and consider a dataset composed of firm-months that contains monthly market values (`mv`) and stock returns (`ret`). Firms are identified using the `gvkey` variable, years with `year` and months with `month`. Now we want to assign firm-month observations to decile portfolios based on the market value at the beginning of each calendar year, that is, at the end of December of the *previous* calendar year. The following statements generate a new variable `lagmv` for January firm-months only, equal to a firm's market value at the end of December:

```
egen firmid=group(gvkey)
egen timeid=group(year month)
tsset firmid timeid
gen lagmv=l.mv if month==1
```

After creating `lagmv`, we can use a `forvalues` loop in order to assign firms into deciles each January in the sample, based on end-of-December market values. To do so, we should first create an empty variable `decile` and assign local macros based on the first and last year of data in the sample:

```
drop if lagmv==. & month==1
drop if ret==.
gen decile=.
sum year if month==1
scalar minyr=r(min)
scalar maxyr=r(max)
local m=minyr
local n=maxyr
forvalues i=`m'(1)`n'{
    qui xtile xmv=lagmv if year==`i',nq(10)
    qui replace decile=xmv if year==`i'
    qui drop xmv
    di `i'
}
```

Now we have a variable `decile` that captures a firm's size-portfolio membership at the beginning of the year. To ensure the firm is assigned to the same portfolio throughout the year, we can simply use the `egen` command. Next, average monthly portfolio returns can be computed and, finally, abnormal monthly returns can be computed:

```
egen helpvar=min(decile), by(gvkey year)
replace decile=helpvar
drop if decile==.
drop helpvar
egen portfolioreturn=mean(ret), by(decile year month)
gen abret=ret-portfolioreturn
```

In the examples presented above, we ran a loop over a known set of numeric values. In some situations, however, we may want to run a loop over the elements of list of non-numeric items. In these situations, we can use a `foreach` loop instead of a `forvalues` loop. For example, assume we want to store the means of a list of variables ("btm", "size", and "age") in our data and we want to simplify this process by using a loop. We can use the `foreach` as follows:

```
foreach var of varlist btm size age {
    sum `var'
```

```
    gen `var'_mean=r(mean)
}
```

Alternatively, we could have stored the list of variables in a local macro and then loop over the contents of the local macros string to achieve the same objective:

```
local vlist="btm size age"
foreach var of local vlist {
    sum `var'
    gen `var'_mean=r(mean)
}
```

Chapter A.1 provides another example of the use of a `foreach` loop, where data is downloaded separately for a list of countries.

`[help forvalues]` `[help display]` `[help quietly]` `[help foreach]`

2.14 Using do-files

Using "do-files" is an essential part of using Stata. With do-files, you can save all commands used for a specific project, run large chunks of code and, most importantly, run an entire project from beginning to end all over again. Being able to run an entire project again, from opening the initial dataset to running the regression analyses, is very important as it allows you to easily make changes without permanently altering the original data. In addition, the use of do-files (or any other file that stores programming code) is essential in light of the need for replicability and transparency in our research. Do-files can be created by clicking the **Do-file** button in the Stata shortcut menu (see Figure 2.2). Do-files are simple text files that are saved as *.do, but they can be opened and edited in Notepad or any other text editor. Below is an example of the contents of a do-file, in which datasets are merged (more on merging in Section 3.4) and saved into a new file.

```
use "C:\dv\InFiles\id.dta", clear
keep if nr==2
keep ticker cusip sdates sdatesm1
sort cusip
save "C:\dv\OutFiles\id_2.dta", replace

use "C:\dv\InFiles\id.dta", clear
keep if nr==1
keep ticker cusip sdates sdatesp1
sort cusip
save "C:\dv\OutFiles\id_1.dta", replace
```

```
use "C:\dv\InFiles\stocknames.dta", clear
drop if year(end_date)<2000
keep permno ncusip cusip exchcd siccd shrcd
replace ncusip=cusip if ncusip==""
drop cusip
ren ncusip cusip
duplicates drop permno cusip exchcd, force
sort cusip
merge cusip using "C:\dv\OutFiles\id_1.dta"
gen ibesticker=ticker
drop ticker sdates* _merge
drop if permno==.
sort cusip
merge cusip using "C:\dv\OutFiles\id_2.dta"
drop if permno==.
replace ibesticker=ticker if ibesticker==""
drop ticker sdates* _merge
sum permno
sum permno if ibesticker!=""
sort cusip
save "C:\dv\OutFiles\stocknames_0.dta", replace
```

By clicking `Ctrl+D` on a Windows computer when the do-file editor is opened, you can run the code of the entire do-file. Alternatively, you can also select one or more lines of code and click `Ctrl+D` to let Stata process only part of the code. For Mac OS, the keyboard shortcut is `Shift+Cmd+D`.

It may also be helpful to write comments in your do-file to be able to easily recognize the steps that you've taken in the process. Comments can be placed by starting a line with `*` or `//` and then writing down your comment. All lines starting with `*` or `//` will not be processed by Stata as commands. For example, Stata will only run the line including `drop if sic>5999 & sic<7000` in the following block of code:

```
************************************************
*Remove all financial firms (SIC codes 6000-6999)
************************************************
drop if sic>5999 & sic<7000
```

Alternatively, you can also let Stata ignore whole blocks of code (multiple lines) by using the combination of `/*` and `*/`, where `/*` is placed at the start of the code block and `*/` is placed at the end of the code block that should be excluded. For example, the following code will gives us the exact same results as in the previous example:

```
/*
Remove all financial firms (SIC codes 6000-6999)
```

```
*/
drop if sic>5999 & sic<7000
```

[help do] [help doedit]

2.15 Local and global macros

Besides the variables we observe in our dataset, Stata can also make use of other variables ("macros") that are not part of the dataset, similar to other programming languages.[4] We have seen an example of such a macro before, as `` `i' `` in the `forvalues` loop (see Section 2.13). There are two flavors of macros, namely `local` and `global` macros. The key difference between these is their scope. After having assigned content to a `global` macro, the contents of this macro will be available to us from Stata's memory until we exit the program. The scope of a `local` macro is more narrow, in that its contents will be will be available to us from Stata's memory only during a particular session in the **Command window** or execution of a code block in a do-file. Assuming we always work from a do-file, this means that every time we run code from the do-file editor we need to re-assign the contents of a `local` macro. For the `global` macro, this is not needed. Hence, we can say `global` macros have a broader scope.

Let's see more closely how these macros work. In the following example, a new `global` macro named `glob1` is assigned a numeric value of 20.

```
global glob1 20
```

We can now use this macro in subsequent commands, for example by just printing its content or by using the content in the creation of a new variable. To use the information stored in the global macro, we can "call" it using the dollar sign followed by its name:

```
. di $glob1
20

. gen var1=$glob1+5

. sum var1

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
        var1 |      1,000          25           0         25         25
```

[4] Although this section focuses on macros, you can also use scalars for almost similar purposes (see `help scalar`). Scalars are commonly used as part of programs and their disadvantage is that it may be difficult to differentiate them from variables. The reason is that unlike local and global macros, they are referred to in the same way as variables (for example, `x` might refer to variable `x` or scalar `x`, while we would refer to `` `x' `` and `$x` with local and global macros, respectively).

We can also assign string content to `global` macros, which allows us to make strings in other parts of our code become more flexible. For example, assume you are working with data that is stored in a local directory on your computer with a relatively long path name, e.g., "`C:\home\dv\PROJECTS\STATA\Files`". It would be inefficient to write down this path every time we open a new file from this location. More importantly, when sharing a project with a co-author, for whom the path to these data files is likely to be very different on their computer, having to replace the path names in the code each time someone else takes a turn on the data is not very efficient. What we can do instead is use a macro to make this path flexible throughout the code, by simply assigning the string of the path name to a `global` macro and to call upon this macro every time we need to open a file from this location:

```
global filepath "C:\home\dv\PROJECTS\STATA"

use "$filepath\InFiles\compdata.dta", clear
...
...
save "$filepath\OutFiles\compdata_clean.dta", replace

use "$filepath\InFiles\crspdata.dta", clear
...
...
save "$filepath\OutFiles\crspdata_clean.dta", replace

...
```

Note that because Stata reads the lines in do-files from top to bottom, it is important to assign to `global` macro as early as possible. In this specific example, it would be wise to place the assignment of the macro at the top of the do-file, to allow a co-author to easily identify and change the assignment.

For this example, using a `local` macro would not have been helpful, since its content is lost after every time we run a block of code from a do-file editor. As we saw before, `local` macros come in handy when we need a temporary variable that easily changes values within a program, such as in a `forvalues` loop. Here is an example of a block of code we have seen before:

```
sum year
local m=r(min)
local n=r(max)
forvalues i=`m'(1)`n'{
    reg price bps eps if year==`i'
}
```

Here we allow the macro content to be variable and depend on the summary statistics given by `sum year`. As you can see, the macro content can be used both as a direct input in the `forvalues` command and as part of the code that is included within the loop. The difference between macro `i` on the one hand, and `m` and `n` on the other hand, is that the latter are fixed after the assignment, while the former represents the index of the loop and varies depending on the iteration of the loop. To better understand the issue of the scope of `local` macros, try running the above block of code from a do-file editor either i) all at once, or ii) line-by-line.

As discussed with the earlier example, an additional useful feature of the use of macros is that we can allow strings to become flexible. For example, let's say we have 1000 files that we need to process in our program, named `file_1.dta` through `file_1000.dta`. We wouldn't want to write out the `use "$pathname\InFiles\file_1.dta", clear` command and change the filename a thousand times in our code. Instead, we can simply use a macro and allow part of the filename to become variable:

```
forvalues i=1(1)1000{
        use "$pathname\InFiles\file_`i'.dta", clear
        ...
        ...
        save "$pathname\OutFiles\newfile_`i'.dta", replace
}
```

`[help macro]`

2.16 Identifying local directories and installing new programs

In the previous examples in which Stata files were opened or saved, the complete file path was used to refer to a file (or we replaced the file path with a `global` macro). An alternative option is to use Stata's working directory, which can be identified using the `cd` command. For example:

```
. cd
C:\home\pkg\Stata18

. pwd
C:\home\pkg\Stata18
```

If we put all our data files in this folder, we can simply ignore paths, and open and save files as follows:

```
. use file
```

```
. gen newvar=oldvar+2

. save file2
file file2.dta saved
```

Of course, this is probably not so useful when working on multiple projects, so we can also adjust Stata's working directory using the `cd` command:

```
. cd "C:\home\dv\PROJECTS\STATA"
C:\home\dv\PROJECTS\STATA
```

Similar to the use of a `global` macro, we can include this line at the top of our do-file to allow the path to the folder with data files to become flexible and to make it easier to share code with co-authors.

Another useful command is `sysdir`, as it provides the paths to several of Stata's key directories. On my computer, the command gives me the following information:

```
. sysdir
   STATA:  C:\Program Files\Stata18\
    BASE:  C:\Program Files\Stata18\ado\base\
    SITE:  C:\Program Files\Stata18\ado\site\
    PLUS:  C:\Users\dv\ado\plus\
PERSONAL:  C:\Users\dv\ado\personal\
OLDPLACE:  c:\ado\
```

This information is useful, because it allows us to identify where Stata stores all of its commands, which are simply separate sets of programming codes stored in `*.ado` files. For example, file `C:\Program Files\Stata18\ado\base\r\regress.ado` contains the code behind Stata's regression command (make sure to not change anything in this code).

The "PLUS" folder allows us to add self-created commands or additional commands that are available from the web. For example, for a long time it was common for researchers to use the command `cluster2` to compute two-way clustered standard errors, but this command is not preinstalled in Stata.[5] To be able to use this command, we should first download it from an external source, such as: http://www.kellogg.northwestern.edu/faculty/petersen/htm/papers/se/cluster2.ado. By saving this ado-file in the folder `C:\Users\dv\ado\plus\c`, we simply "install" it for use in Stata (note that the file needs to be saved in a separate subfolder with the name of the first letter of the command). We can similarly install other functions from

[5] In recent years it has become more common for researchers to use `reghdfe` to compute multi-way clustered standard errors, which applies better corrections for degrees of freedom and is available for installation in a different way, as explained below

the web, such as the programs to compute cluster-robust standard errors for robust regression estimators from Gassen and Veenman [2023]: `https://github.com/dveenman/outliers`.

Other commands that are not preinstalled in Stata might be available in Stata's online repository, and can be installed directly within Stata. For example, the command `xtfmb`, which can be used to perform Fama-Macbeth cross-sectional regressions, can be installed directly via the **Command window** using the `ssc install` command:

```
. ssc install xtfmb
checking xtfmb consistency and verifying not already installed...
installing into c:\ado\plus\...
installation complete.
```

Another example of a command that can be installed this way is `reghdfe`, which can be used to include multiple fixed effects structures in a regression (see Section 4.6) or to easily adjust standard errors for clustering in multiple dimensions (see Section 4.5). This function should be installed together with a package called `ftools`:

```
. ssc inst reghdfe
checking reghdfe consistency and verifying not already installed...
installing into C:\Users\dv\ado\plus\...
installation complete.

. ssc inst ftools
checking ftools consistency and verifying not already installed...
installing into C:\Users\dv\ado\plus\...
installation complete.
```

`[help cd]` `[help dir]` `[help sysdir]` `[help ssc]`

Chapter 3

Advanced data management

3.1 Outliers and scaling

Summary statistics and regression estimates can be highly influenced by a small number of extreme observations ("outliers"). These extreme observations can arise from database errors, but often simply arise as a result of scaling. For example, we often generate variables based on ratios, such as a firm's return on assets (ROA):

```
gen roa=ib/lag_assets
```

If the denominator (`lag_assets`) has values that are close to 0 (accounting variables are generally reported in millions of dollars/euros in databases), the values of the `roa` variable can become extremely large. This problem becomes even worse with a variable such as return on equity (ROE), for which the denominator (i.e., the book value of shareholders' equity) can also take on negative values.

As we are generally interested in the behavior or relations of variables for the average firm (or stock, or manager, or analyst, etc.) in a cross-section, we should ensure that the influence of these few extreme observations is limited. Two common (but imperfect) ways to deal with these issues are data truncation and winsorization. With truncation, we simply drop the most extreme (most positive and most negative) observations from the sample. With winsorization, we set the most extreme small and large observations equal to the values of less extreme small and large observations, respectively. Both methods are used in the literature, although winsorization is mostly commonly in recent times. Relative to winsorization, the drawback of truncation is that applying this procedure in a setting with many continuous variables can lead to significant sample attrition. Because the observations identified as "outliers" are often not random (i.e.,

they are often from smaller firms), truncation can lead to substantial deletion of a non-random subset of the data.

Researchers typically choose the most extreme percentiles of the distribution of a variable to identify the outliers. That is, continuous variables are typically truncated or winsorized at the 1st and 99th percentiles of their distributions. To determine these "1st and 99th percentiles", we can use the `xtile` command we saw earlier to construct new variables that contain the percentile ranks of the variables of interest. When we choose truncation, we can first create percentile rank variables for each of the variables to be filtered, and next drop the observations where the percentile for any variable is equal to 1 or 100. Make sure to first drop all missing observations:

```
drop if roa==. | accr==. | cfo==.
xtile xroa=roa, nq(100)
xtile xaccr=accr, nq(100)
xtile xcfo=cfo, nq(100)
drop if xroa==1 | xroa==100
drop if xaccr==1 | xaccr==100
drop if xcfo==1 | xcfo==100
```

When truncating the data for multiple variables at the same time as in the above example, it is important to make sure that you create percentile variables on the full sample first before you drop outlier observations.To better understand why, compare the outcomes from the above code with outcomes from the following reshuffling of the code:

```
drop if roa==. | accr==. | cfo==.
xtile xroa=roa, nq(100)
drop if xroa==1 | xroa==100
xtile xaccr=accr, nq(100)
drop if xaccr==1 | xaccr==100
xtile xcfo=cfo, nq(100)
drop if xcfo==1 | xcfo==100
```

The second option is winsorization. With winsorization, we set observations in the smallest percentile (1) equal to the smallest value of the 2nd percentile, while we set observations in the largest percentile (100) equal to the largest value of the 99th percentile. We can do this as follows:

```
sum roa, d
replace roa=r(p1) if roa<r(p1)
replace roa=r(p99) if roa>r(p99) & roa!=.
sum accr, d
replace accr=r(p1) if accr<r(p1)
```

```
replace accr=r(p99) if accr>r(p99) & accr!=.
sum cfo, d
replace cfo=r(p1) if cfo<r(p1)
replace cfo=r(p99) if cfo>r(p99) & cfo!=.
```

We can also use the `xtile` command to achieve the same result (try this for yourself as an exercise). The result of winsorization is that the tails of the distribution of the winsorized variable show small spikes in a histogram, as is displayed in Figure 3.1. Before winsorization, the distribution looks very different, as can be seen in Figure 3.2. The latter distribution is not useful for a proper analysis (whether simple summary statistics or regression analyses), because any results will be strongly influenced by a few extreme observations, that often have little economic meaning.

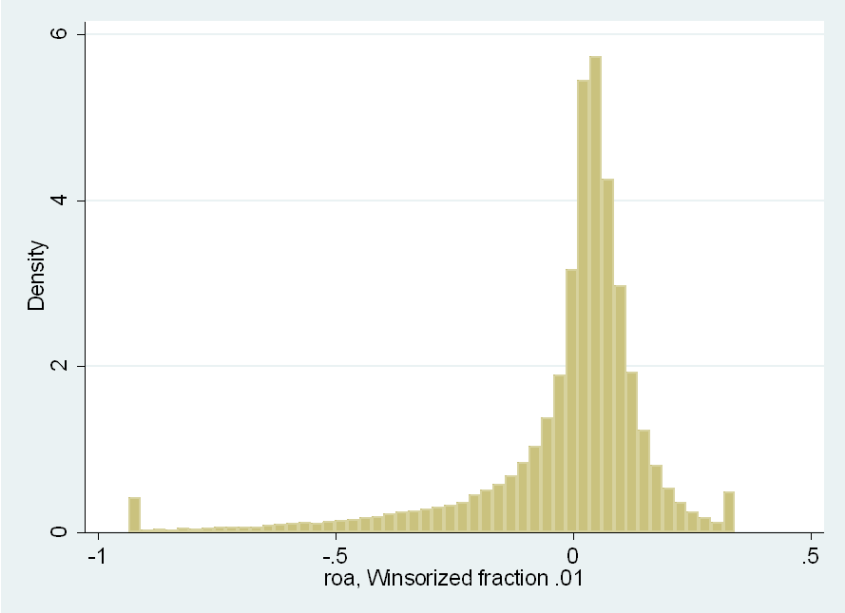


Figure 3.1: *Histogram of variable after winsorization*

As you can see in this section, I did not rely on existing commands that are designed to winsorize a variable directly. The reason is that it is essential for the researcher to observe the outlier problem with each of the variables and see whether the adjustments are actually effective (or even needed!), rather than to blindly use a command without seeing what truly happens with the data. Truncation and winsorization are not panacea for the treatment of outliers and the procedures have their own limitations and drawbacks because we reduce the influence of extreme values at the expense of losing potentially relevant information.

Still, there are programs available that can perform the procedure more quickly (at the expense of being intransparent to use as researchers). For example, `winsor` can be install via

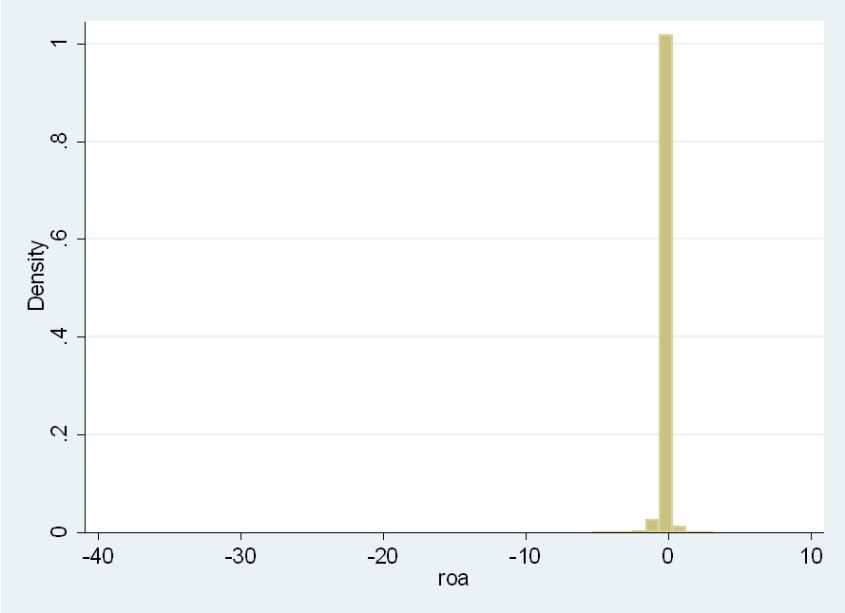


Figure 3.2: *Histogram of variable before winsorization*

`ssc inst winsor`. It can then be used to create a new variable that is winsorized based on an input variable at the extreme percentiles (`p(0.01)`). This gives us the same distribution as we would get with code using `sum roa, d`:

```
winsor roa, gen(roa_w) p(0.01)
```

The program `winsor2`, available via `ssc inst winsor2`, is quite similar to `winsor` except that its syntax is different and it makes it easy for us to the winsorization by group. For example, depending on the setting and research question, it can be useful to winsorize variables by sample year instead of the pooled sample of observations. The `winsor2` command can then be used as follows (note that we do not specify `gen(roa_w)` as option because the program automatically creates a new variable with the `_w` suffix):

```
winsor2 roa, cuts(1 99) by(fyear)
```

Another important issue to consider is that the outlier filtering should always be used as a final step in the data management process. It is the final step before creating descriptive statistics for tabulation and performing our estimation. Exceptions arise, for example when variables are created based on summary statistics or regression output in an intermediate step. Sometimes we need to winsorize before both the intermediate step and at the final step before the analyses as in, for example, Appendix A.2). But a key mistake that is occasionally made is that with ratio variables such as `roa=ib/lag_assets` as we had above, researchers winsorize

(or truncate) the input variables `ib` and `lag_assets` rather than the output variable `roa` (see DeFond et al. [2016] for a brief discussion on this issue).

Assume for example that we would have first winsorized the numerator and denominator before we computed the ratio. In this case, the distribution of the ratio variable looks very different. We can see from the summary statistics and the histogram (Figure 3.3) that the ratio variable based on winsorized numerator and denominator still has extreme values and much stronger skewness and kurtosis than the variable that was winsorized after the ratio was calculated:

```
                          roa_w_alt
-------------------------------------------------------------
      Percentiles      Smallest
 1%     -.930086        -13.667
 5%    -.4340918        -10.061
10%    -.2397475       -7.662127       Obs               122,902
25%     -.039572        -7.22072       Sum of wgt.       122,902

50%     .0324221                       Mean           -.0194555
                        Largest        Std. dev.       .2503066
75%     .0789836        2.841781
90%     .1369681        4.718609       Variance        .0626534
95%     .1861113        5.621406       Skewness       -5.747905
99%     .3400844        9.760044       Kurtosis         180.797
```

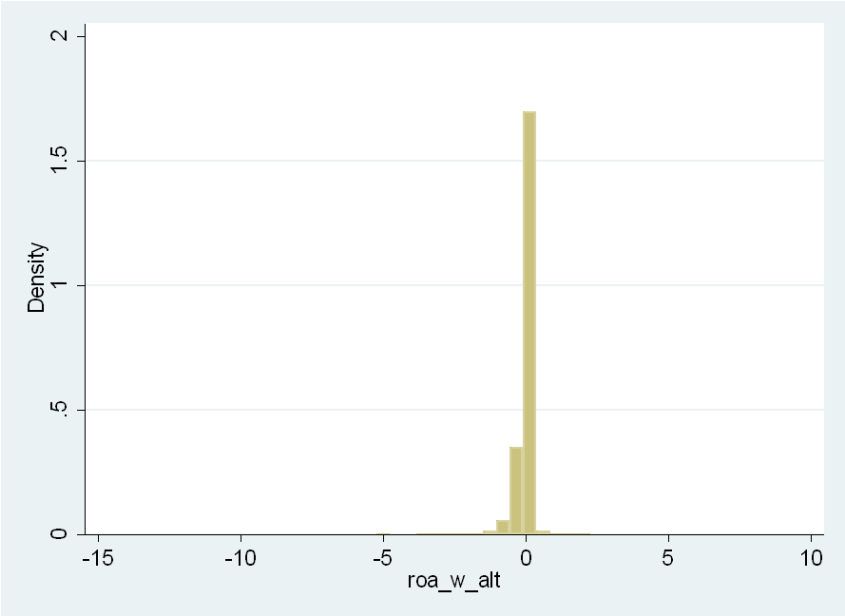


Figure 3.3: *Histogram of variable with winsorization of input variables*

A related empirical problem is the role of "scale effects" in accounting research. Regression analyses are likely to be less reliable when performed on variables that are either undeflated (in

millions of dollars/euros) or on a per-share basis, compared with variables that are expressed as ratios. This is why we often see regression variables being scaled by (lagged or average) total assets or (lagged) market values. Total assets and market values are often viewed as measures of firm size, or "scale". By scaling variables of interest (e.g., net income, accruals, cash flows) by a measure of firm size, we can make observations comparable while controlling for differences in firms' sizes. Also, by scaling variables by a measure of firm size, we make sure that a few of the largest firms are not driving our results. This phenomenon is known as the "scale effect" and is related to, but different from, the effect of outliers. For more information on scale effects see for example Easton [1998, 1999], Easton and Sommers [2003], or Goncharov and Veenman [2014].

3.2 Transposing data

In some situations our data is structured in a way that is not useful for statistical analyses. For example, we might have the following annual stock return data for a set of ten firms:

id	ret2002	ret2003	ret2004	ret2005	ret2006	ret2007
1	-0.049	0.003	0.020	-0.078	0.075	0.067
2	-0.101	-0.153	-0.123	0.094	-0.142	0.075
3	-0.193	-0.084	-0.077	0.026	-0.166	-0.071
4	-0.130	0.009	-0.134	-0.072	-0.170	-0.023
5	0.066	0.054	0.010	-0.007	-0.190	0.110
6	-0.003	0.017	0.123	0.110	0.048	-0.186
7	0.003	0.078	-0.031	-0.089	-0.089	0.082
8	0.037	0.081	-0.104	0.040	-0.032	-0.110
9	-0.073	-0.066	-0.119	-0.193	-0.061	-0.078
10	-0.110	-0.003	-0.119	-0.119	0.012	0.068

As we may want to merge these data with firm-year accounting data, the preferred structure of the data would be:

id	year	ret
1	2002	...
1	2003	...
1	2004	...
1	2005	...
1	2006	...
1	2007	...
2	2002	...
2	2003	...
...	...	...

While manually transforming the data in the above example is possible, transforming data manually for larger samples will be too time consuming and is not needed. Instead, we can use the `reshape` command to let Stata do the trick for us. For the above example, we can type:

```
reshape long ret, i(id) j(year)
```

where the command `reshape long` transposes the data from a "wide" format to a "long" format. Of course, we can also transpose the data back to the wide format:

```
reshape wide ret, i(id) j(year)
```

What is essential in these steps is that before we are able to reshape, we need to carefully prepare and understand the structure of our data. Specifically, in the **wide** format, we need a variable for the dimension `i` (which in this case is variable `id`) that captures every unique observation (row) in the data. At the same time, the data to be transposed needs to be contained in a set of variables that start with the same name (`ret` in this case) and end with a unique value (`200X` in this case). This latter unique value will be stored in a new variable `j`, which we label `year` in this case. Instead, in the **long** format, we should have both the `i` and `j` variables prepared in the data. The combination of `i` and `j` should be unique (e.g., a unique firm-year combination), while `i` can have multiple duplicate observations (e.g., one company has multiple observations, each for a different year). Last, the values of `j` should be unique *within* each value of `i` (e.g., a firm-year dataset should only contain one value for the year 2003 for company A). Appendix A.5 provides an example of an alternative method to reshape the data.

`[help reshape]`

3.3 Appending data

Stata can also stack different datasets, using the `append` command. If we have two different datasets with exactly the same set of variables but, for example, different time periods (see example A in Figure 3.4) we can use the `append` command to let Stata combine these datasets into one. It can also be that only a subset of variables are overlapping and that some variables are unique in each dataset (see example B in Figure 3.4). In those cases, we can also use the `append` command. The variables that are unique to one dataset will simply remain empty for the observations in the other dataset. The `append` command can be applied as follows:

```
use "C:\...\erdport1_2002.dta", clear
append using "C:\...\erdport1_2003.dta"
append using "C:\...\erdport1_2004.dta"
```

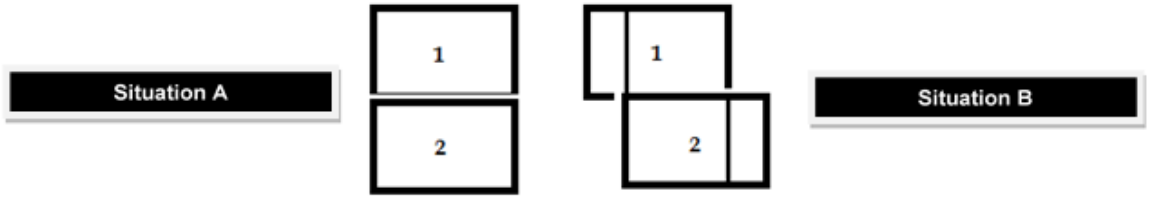


Figure 3.4: *Appending datasets*

In this example, three datasets containing stock return data for different years (reflecting situation A in Figure 3.4) are stacked.

```
[help append]
```

3.4 Merging data

In accounting and finance research settings, we often rely on different data sources. These sources need to be combined before we are able to test our hypotheses. For example, we may need to combine accounting data (from databases such as Compustat or Worldscope) with stock return data (from databases such as Compustat, CRSP, or Datastream) or with analyst forecast and coverage data (from IBES, for example). Before being able to merge these datasets, we should ensure that each of the datasets has common variables on which we can merge (most often: firm and/or time variables). Because each database has its own unique variable that identifies a company, and because merging based on company names (which are spelled differently in every database) is not desirable or possible, we should therefore always first ensure that both datasets have the same common company and time identifiers. Examples of common company identifiers that are identified in research settings are GVKEY, PERMNO, CUSIP, ISIN, CIK, etc.

In the case of merging, there are always a few common variables in the datasets. Always ensure that the variables used for merging (e.g., `gvkey` and `fyear`) are the only overlapping variables. In situation A of Figure 3.5 we can see two datasets that have some common variables in the grey area, unique variables in the white areas, and exactly the same number of observations. In situation B, which is more common, we see that (1) some observations are only available in dataset 1, (2) some observations are only available in dataset 2, while (3) observations in the grey area are available in both datasets. When our analyses require data from both datasets 1 and 2, it makes sense to drop all observations that are only available in dataset 1 or only in

dataset 2.

Figure 3.5: *Merging datasets*

There are multiple commands that can be used to merge data in Stata. My preferred command is `joinby` (a common alternative is `merge`). The `joinby` command follows the structure `joinby [varlist] using "filepath/filename.dta", [options]`. That is, the `joinby` command is followed by (a) the common variables on which to merge, (b) the term "using", and (c) the location of the dataset to be merged with. Note that after have successfully performed the merge, a new variable `_merge` is created. This variable equals 1 or 2 based on the areas depicted in Figure 3.5, or 3 for the grey-shaded area. The variable equals 1 for observations that are available only in dataset 1, equal to 2 for observations available only in dataset 2, and equal to 3 for observations that are available in both datasets. When we are interested only in the intersection of the data, we can type:

```
use "C:\...\dataset1.dta", clear
joinby gvkey year using "C:\...\dataset2.dta"
```

The default option for the `joinby` command is to retain only those observations that are available in both datasets, or alternatively only those cases for which `_merge==3`. This is similar to adding the option "`, unmatched(none)`" to the code:

```
use "C:\...\dataset1.dta", clear
joinby gvkey year using "C:\...\dataset2.dta", unmatched(none)
```

That is, the `unmatched` option indicates which of the unmatched observations should be retained in Stata's memory. Alternative options are "`, unmatched(both)`" which does not exclude any observations, whether matched or unmatched; "`, unmatched(master)`" which retains only the observations in the original ("master") dataset, whether matched or unmatched, and drops those of the other ("using") dataset that remain unmatched; and "`, unmatched(using)`" which drops only the unmatched observations from the original dataset. My default choice is to add the "`, unmatched(master)`" option. It is also useful to briefly inspect the values contained in `_merge` to assess the success of the merge, although this variable typically has limited infor-

mation content unless we use the "`, unmatched(both)`" option. It is typically wise to drop the `_merge` variable to prevent errors in subsequent merges with other datafiles.

```
use "C:\...\dataset1.dta", clear
joinby gvkey year using "C:\...\dataset2.dta", unmatched(none)
tabstat _merge, by(_merge) stats(N)
```

We can also merge on other variables. For example, we might want to merge firm-year accounting data with the analyst EPS forecast that was made in the last month of the fiscal year. That is, for a firm ending its year on December 31, we might need the average analyst EPS forecast from the IBES database that was outstanding in the month December. For a firm ending its year on May 31, we would need the average analyst EPS forecast from the IBES database that was outstanding in the month May. Again, to be able to merge these databases, we need common variables for both firms and time. In this case, common time variables can be created by generating a year variable and a month variable for both datasets, while making sure that the names of the variables are also the same. A common company identifier could in this case be CUSIP, although we have to ensure that the CUSIP variables in both datasets have the same number of digits. Some databases report 9-digit CUSIPs, while others contain 8-digit CUSIP codes (see Section 2.8 for string functions such as `substr`, which can help to transform 9-digit string variables into 8-digit string variables).

```
use "$path\ibesdata.dta", clear
gen year=year(statpers)
gen month=month(statpers)
drop if cusip==""
keep cusip year month feps anfollow
sort cusip year month
save "$path\ibesdata2.dta", replace

use "$path\compustat_firmyears.dta", clear
gen year=year(datadate)
gen month=month(datadate)
drop if cusip==""
joinby cusip year month using "$path\ibesdata2.dta", unmatched(master)
tabstat _merge, by(_merge) stats(N)
drop _merge
```

`[help joinby]` `[help merge]`

Chapter 4

Data analysis

4.1 Descriptive statistics

The most time-consuming part of doing research is typically the data management process. Ultimately, however, we are interested in the statistics. Let's therefore have a look at some examples of how we can use Stata to obtain statistics. The simplest command to obtain some statistics is the `summarize` command. With this command, we can obtain key statistics for a specific (set of) variable(s). For example, we can summarize a variable `nseg`, which in my example captures the number of business segments that a firm reports in a year:

```
sum nseg
```

The output looks as follows:

```
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
        nseg |     15667    2.047999    1.496323          1          7
```

As can be seen from the output, the standard `summarize` command provides us the number of observations, mean, standard deviation, minimum, and maximum value for a variable. When we summarize a dummy variable, we can see the fraction of observations for which the dummy variable is equal to 1. For example, consider the variable `big4`, which equals 1 for firm-years audited by a Big 4 auditor, and 0 for all other firm-years. In my example data, I get:

```
. sum big4

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
        big4 |     15667    .8313653    .3744409          0          1
```

Here we learn that 83.1% of the total of 15,667 firm-years are audited by a Big 4 auditor. We can also easily summarize multiple variables at the same time:

```
sum nseg fscore
```

where the output looks as follows:

```
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
        nseg |     15,667    2.047999    1.496323         1          7
        loss |     15,667    .2494415    .4327036         0          1
```

Or we can add conditional statements to tabulate summary statistics only for some subset of the data:

```
. sum nseg loss if loss==0

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
        nseg |     11,759    2.138702    1.542525         1          7
        loss |     11,759           0           0         0          0
```

Stata can also produce additional summary statistics when we add the `,d` ("detail") option:

```
. sum nseg, d

                          nseg
-------------------------------------------------------------
      Percentiles      Smallest
 1%            1              1
 5%            1              1
10%            1              1       Obs                15667
25%            1              1       Sum of Wgt.        15667

50%            1                      Mean           2.047999
                      Largest         Std. Dev.      1.496323
75%            3              7
90%            4              7       Variance       2.238983
95%            5              7       Skewness       1.287639
99%            7              7       Kurtosis       3.763741
```

Now we also see the median, the distribution of the variable across percentiles, and additional relevant statistics such as skewness and kurtosis. At first sight, this table might be a bit difficult to interpret. For example, I often experience students trying to understand the connection between the two left-hand columns (those with the headers "Percentiles" and "Smallest"). These columns should be viewed separately. In the "Percentiles" column we can see the values of the variable associated with different percentile ranks. For example, we can see that for at

least 75 percent of observations, the value of the `nseg` variable is 3 or lower. For 95 percent of observations, the value is 5 or lower. The "Smallest" and "Largest" columns simply provide us the four lowest and four highest values of the variables, respectively.

For a visual inspection of the distribution of the data, we can also create a histogram using the `hist` command. The output can be found in Figure 4.1 (note that the default graph settings changed in Stata 18, so the histogram will look slightly differently in older versions of Stata):

```
hist nseg
```

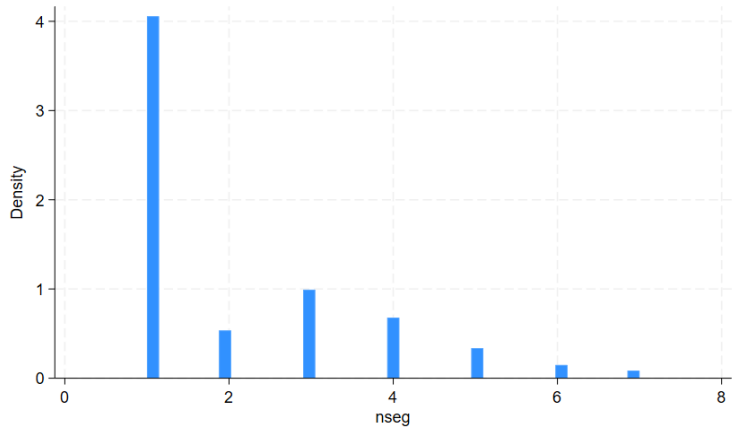


Figure 4.1: *Example of a histogram*

We can also use the `tabstat` command for summary statistics. One advantage of using `tabstat` is that we can specify exactly which statistics we want to be displayed. Simply typing `tabstat` plus the variable(s) of interest gives us:

```
. tabstat nseg

    variable |      mean
-------------+----------
        nseg |  2.047999
------------------------
```

Adding the option `, stats(...)` provides us with a set of additional statistics, for example:

```
. tabstat nseg, stats(N mean sd p25 median p75)

    variable |         N      mean        sd       p25       p50       p75
-------------+--------------------------------------------------------------
        nseg |     15667  2.047999  1.496323         1         1         3
----------------------------------------------------------------------------
```

Another advantage of using `tabstat` is that we can summarize variables by different groups. For instance, we can easily summarize the statistics of firms audited by a Big 4 versus a non-Big 4 auditor:

```
. tabstat nseg loss, by(big4)

Summary statistics: Mean
Group variable: big4

    big4 |      nseg      loss
---------+--------------------
       0 |  1.598789  .3050719
       1 |  2.139117  .2381574
---------+--------------------
   Total |  2.047999  .2494415
------------------------------
```

Or we can summarize a variable (for example, a firm's annual stock return in variable `bhr`) for every year in our sample:

```
. tabstat bhr, by(fyear) stats(N mean median)

Summary for variables: bhr
by categories of: fyear (Data Year - Fiscal)

   fyear |         N      mean       p50
---------+------------------------------
    2004 |      2451  .1908479  .1497006
    2005 |      3259  .1051607  .0446429
    2006 |      3396  .1544559  .1173893
    2007 |      3512  .0003045 -.0659407
    2008 |      3049 -.3743498 -.4045276
---------+------------------------------
   Total |     15667  .0124271 -.0074322
--------------------------------------
```

Another advantage of `tabstat` is that when the output is displayed as above, we can directly copy the statistics in table format into Excel. To achieve this, select the block of output with your mouse, starting with the line containing `fyear` and ending with the line containing `Total`. Then right-click your mouse and choose `Copy Table`. This table can now be pasted into Excel, which recognizes the format and puts every statistic into a different cell.

Another function that can be helpful is `tabulate`, which is a versatile tool to create frequency tables (see the help file for more details). I often use this function when I want to tabulate the means of a variable for different groups that are split by two categorical variables. For example, let's say I want to examine stock illiquidity around 2005 for firms in countries with

versus without IFRS adoption (as captured by dummy variable `ifrs`). I can then tabulate the means of the liquidity variable along the dimensions `year` and `ifrs` as follows:

```
. tabulate year ifrs, summarize(zeroreturn) means

Means of zeroreturn

           |        ifrs
      year |         0          1 |     Total
-----------+----------------------+----------
      1997 | .26316084  .40436974 | .29443972
      1998 | .26435275  .41703674 | .30416961
      1999 | .23038658  .43127403 | .28014852
      2000 |  .2377778   .4109584 | .27814002
      2001 | .24320788  .42387623 |  .2857064
      2002 | .23410513  .45509432 | .28343205
      2003 | .21828692  .44823993 | .26681914
      2004 | .19536001  .41468009 | .23724099
      2005 |  .1905788  .37712961 | .22436797
      2006 | .18036518   .3250164 | .20567873
      2007 | .17165488  .30147589 |  .1938751
      2008 | .20508533  .32021188 | .22403615
      2009 | .24238386  .34643473 |  .2590272
      2010 | .24348337  .33805393 | .25822946
      2011 | .23819951  .31802251 | .25028696
      2012 | .24446376  .32334254 | .25616878
-----------+----------------------+----------
     Total | .22159004  .38408544 | .25296829
```

It is also important to inspect the correlations between variables, and it is often useful to tabulate these in the paper. Such correlation tables can be made in Stata. For instance, using the `pwcorr` command in the following way:

```
. pwcorr nseg loss big4 bhr

             |     nseg     loss     big4      bhr
-------------+------------------------------------
        nseg |   1.0000
        loss |  -0.1052   1.0000
        big4 |   0.1352  -0.0579   1.0000
         bhr |   0.0504  -0.2764   0.0442   1.0000
```

We can also add more statistics to the `pwcorr` command, such as a significance level (p-value), although this is typically not very insightful given the large samples we often examine:

```
. pwcorr nseg loss big4 bhr, sig

             |     nseg     loss     big4      bhr
-------------+------------------------------------
```

```
        nseg |   1.0000
             |
             |
        loss |  -0.1052   1.0000
             |   0.0000
             |
        big4 |   0.1352  -0.0579   1.0000
             |   0.0000   0.0000
             |
         bhr |   0.0504  -0.2764   0.0442   1.0000
             |   0.0000   0.0000   0.0000
```

Besides normal correlations, it is also often useful to examine and present Spearman correlations. These correlation are based on the ranks of variables instead of their actual values. In essence, all variables are first ranked and assigned a new value based on their rank (`1...n`, where `n` is the total number of observations available in the data for the variable). One reason to use these Spearman rank correlations is to ensure the observed correlations are not obscured by a few extreme observations or other nonlinearities in the data. For example:

```
. spearman nseg loss big4 bhr

Number of observations = 15,667

             |     nseg     loss     big4      bhr
-------------+------------------------------------
        nseg |   1.0000
        loss |  -0.1065   1.0000
        big4 |   0.1421  -0.0579   1.0000
         bhr |   0.0764  -0.3342   0.0668   1.0000
```

`[help sum]` `[help hist]` `[help tabstat]` `[help tabulate twoway]`

`[help pwcorr]` `[help spearman]`

4.2 Basic regression estimation

As stated in the introduction, this is not a guide to econometrics. Please refer to introductory statistics and econometric textbooks for your questions in this area. The guidance provided here will therefore be limited. This is, however, also a consequence of the fact that the econometric methods used in accounting research are typically not very complex. We often see published studies rely on simple summary statistics (mean, median, etc.) and linear regression models that are estimated by OLS.[1]

[1]Note that a common mistake researchers make is to say they "estimate an OLS model." As Wooldridge [2019] explains, this is wrong because OLS is the estimation method and not a model. A related mistake is

The command for estimating a regression with OLS in Stata is `regress`, or shorter `reg`. For example, we can estimate a regression of annual stock returns (`bhr`) on firm size (`lnmv`) and book-to-market (`btm`) as follows:

```
reg bhr lnmv btm
```

which provides:

```
      Source |       SS       df       MS              Number of obs =   15667
-------------+------------------------------           F(  2, 15664) = 2043.92
       Model |  724.239314     2  362.119657            Prob > F      =  0.0000
    Residual |  2775.17237 15664  .177168818            R-squared     =  0.2070
-------------+------------------------------           Adj R-squared =  0.2069
       Total |  3499.41168 15666  .223376208            Root MSE      =  .42091

------------------------------------------------------------------------------
         bhr |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
        lnmv |   .0449926   .0021232    21.19   0.000     .0408308    .0491544
         btm |  -.3818091   .0075421   -50.62   0.000    -.3965926   -.3670257
       _cons |  -.0743153   .0164856    -4.51   0.000     -.106629   -.0420016
------------------------------------------------------------------------------
```

When the dependent variable is an indicator variable, we can also estimate the model using logit or probit. For example, we could use a logit estimation to explain the likelihood that a firm is audited by a Big 4 auditor based on its characteristics:

```
. logit big4 lnmv btm

Iteration 0:   log likelihood =  -7108.349
Iteration 1:   log likelihood = -5973.2101
Iteration 2:   log likelihood = -5794.5818
Iteration 3:   log likelihood = -5782.7644
Iteration 4:   log likelihood =  -5782.684

Logistic regression                               Number of obs   =      15667
                                                  LR chi2(2)      =    2651.33
                                                  Prob > chi2     =     0.0000
Log likelihood =  -5782.684                       Pseudo R2       =     0.1865

------------------------------------------------------------------------------
        big4 |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
        lnmv |   .9010106   .0217003    41.52   0.000     .8584788    .9435425
         btm |   .3462287   .0457056     7.58   0.000     .2566474      .43581
       _cons |  -4.099292   .1374559   -29.82   0.000    -4.368701   -3.829884
------------------------------------------------------------------------------
```

that researchers often refer to "multivariate regression" when they intend to refer to a "multiple" regression, i.e., a linear regression model that includes multiple independent variables. The term "multivariate" is incorrect because it refers to the modeling of multiple dependent variables.

In some settings it is useful to estimate weighted regressions using weighted least squares (WLS). For example, the matching and balancing procedures discussed in Sections 4.8 and 4.9 produce weights that can be used in subsequent regression estimations. Another example is robust regression estimation (Section 4.3), where observations with large standardized regression residuals are downweighted. As a final example, Easton and Sommers [2003] propose using WLS with weights equal to the inverse of firms' squared market capitalization to account for scale effects in price-levels regressions.

In all of these examples, the weights can be considered "analytical weights" and the weighted regression estimation should be implementated by using the `aweight=...` (or short: `aw=...`) option. The following example is from Section 4.9, where each observation receives a positive weight that is stored in variable `_webal`:

```
. reg lnauditfee big4 lnassets aturn curr lev roa salesgr [aw=_webal]
(sum of wgt is 56,168)

      Source |       SS           df       MS      Number of obs   =    36,115
-------------+----------------------------------   F(7, 36107)     =   6090.78
       Model |  21503.0813         7  3071.86876   Prob > F        =    0.0000
    Residual |  18210.4696    36,107  .504347346   R-squared       =    0.5415
-------------+----------------------------------   Adj R-squared   =    0.5414
       Total |  39713.5509    36,114   1.0996719   Root MSE        =    .71017

------------------------------------------------------------------------------
 lnauditfees | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
        big4 |    .367632    .007474    49.19   0.000     .3529828    .3822812
    lnassets |   .4758699    .002582   184.30   0.000     .4708091    .4809306
       aturn |   .1396813    .005085    27.47   0.000     .1297145     .149648
        curr |    .008929   .0023308     3.83   0.000     .0043605    .0134975
         lev |  -.2912367   .0203989   -14.28   0.000    -.3312193   -.2512542
         roa |  -.7417336   .0305394   -24.29   0.000    -.8015918   -.6818754
 salesgrowth |  -.1013017   .0116453    -8.70   0.000    -.1241268   -.0784766
       _cons |   10.43007   .0253165   411.99   0.000     10.38045    10.47969
------------------------------------------------------------------------------
```

[help reg] [help logit] [help probit] [help weights]

4.3 "Robust regression" estimation

As an alternative to the winsorization and truncation procedures discussed in Section 3.1, recent studies commonly estimate "robust regressions" to address the potential influence of outliers in their tests (Leone et al. [2019]) or to increase the precision of their estimates (Gassen

and Veenman [2023]). These robust regressions reflect a collection of a linear regression methods that, similar to OLS, use a specific objective function to estimate the parameters of the regression model for a sample of data. What most robust regressions do is perform an iterative procedure in which observations with large regression residuals get downweighted. Because different estimators and their different implementations in Stata vary in their effectiveness of controlling for outliers (Gassen and Veenman [2023]), this section illustrates how to implement alternative robust estimators in Stata.

4.3.1 Reducing bias with robust estimators

To set the stage, consider the following simulated example dataset of 1,000 observations where variable x and y have a simple relation: $y = x + \varepsilon$, where x and ε are variables drawn from a standard normal distribution with mean 0 and standard deviation 1. For this sample, a regression of y on x should produce a coefficient estimate on x that is close to 1. But let's assume that for a randomly drawn 10 percent of observations, the decimal point of x was placed incorrectly so that the true values of x are all multiplied by 10 in the observed data. When we estimate a simple regression using OLS on the dataset, we get:

```
. reg y x

      Source |       SS           df       MS      Number of obs   =     1,000
-------------+----------------------------------   F(1, 998)       =    240.09
       Model |  375.152084         1  375.152084   Prob > F        =    0.0000
    Residual |  1559.43429       998   1.5625594   R-squared       =    0.1939
-------------+----------------------------------   Adj R-squared   =    0.1931
       Total |  1934.58637       999  1.93652289   Root MSE        =      1.25

------------------------------------------------------------------------------
           y | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   .1904487   .0122912    15.49   0.000     .1663293    .2145682
       _cons |  -.0489038   .0395293    -1.24   0.216    -.1264738    .0286663
------------------------------------------------------------------------------
```

The contamination in the x variable (formerly, "bad leverage points") biases the coefficient on x away from the true value of 1. Using the post-estimation command `avplots` to create Figure 4.2, we can see how this bias is caused by the unusual values of the `x` variable.

Let's have a look at how robust estimators in Stata address this bias. Some studies use Stata's built-in function `rreg`, but we can see this does not solve the problem as the coefficient is still downward-biased:

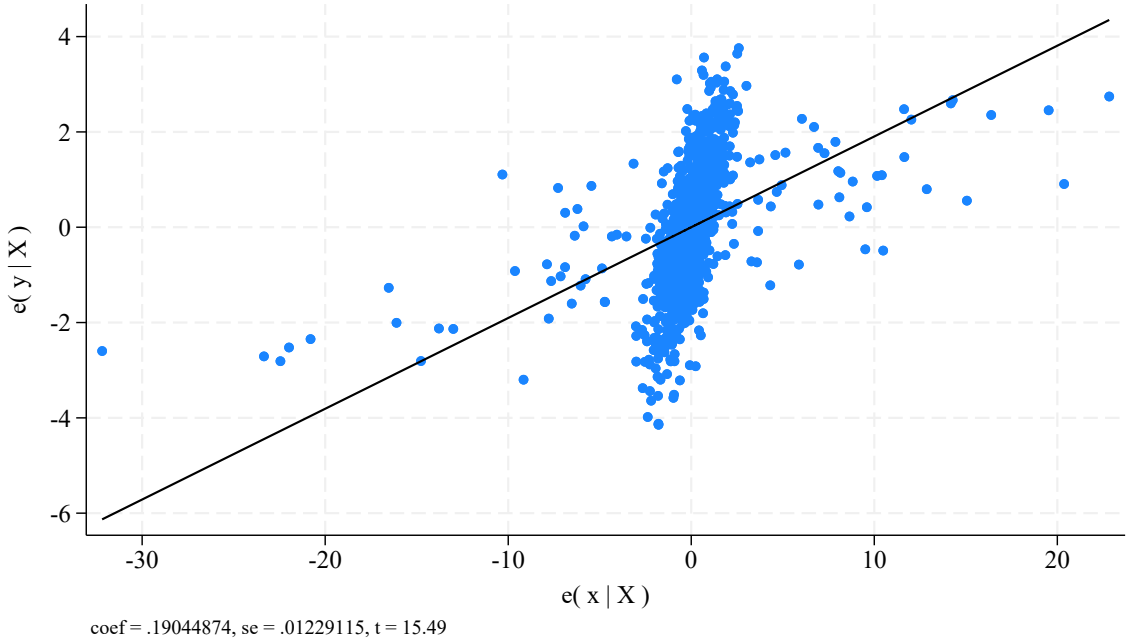


Figure 4.2: *Illustration of effect of outliers in independent variable*

```
. rreg y x

   Huber iteration 1:  Maximum difference in weights = .55024935
   Huber iteration 2:  Maximum difference in weights = .0230179
Biweight iteration 3:  Maximum difference in weights = .15866441
Biweight iteration 4:  Maximum difference in weights = .01408579
Biweight iteration 5:  Maximum difference in weights = .0055938

Robust regression                               Number of obs     =      1,000
                                                F(  1,       998) =     236.59
                                                Prob > F          =     0.0000

------------------------------------------------------------------------------
           y | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   .1952291   .0126926    15.38   0.000     .1703219    .2201362
       _cons |  -.0486608   .0408202    -1.19   0.234    -.1287642    .0314425
------------------------------------------------------------------------------
```

An alternative program we can use is `robreg`, which is available via `ssc inst robreg`. Using the default settings of the "MM"-estimator that is part of `robreg` (the program contains many forms of robust estimation, including for example median regression), we obtain results that look much better than those based on `rreg`. The coefficient on x is not significantly different from the true value of 1 at $p < 0.05$, although the point estimate is still a bit too low

at 0.91:

```
. robreg mm y x
preparing data for subsampling ... done
enumerating 20 candidates 0%....20%....40%....60%....80%....100%
refining 5 best candidates ..... done
iterating RLS ......................... done
fitting empty model ... done
computing standard errors ... done

MM regression (85% efficiency)                  Number of obs     =      1,000
                                                Wald chi2(1)      =     350.40
                                                Prob > chi2       =     0.0000
                                                Pseudo R2         =     0.2071
                                                Breakdown point   =         50
                                                M-estimate: k     =  3.4436898
                                                S-estimate: k     =   1.547645
                                                Scale             =  1.0900179

------------------------------------------------------------------------------
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   .9055091   .0483735    18.72   0.000     .8105836    1.000435
       _cons |  -.0453602   .0347995    -1.30   0.193    -.1136487    .0229283
------------------------------------------------------------------------------
Hausman test of MM against S:    chi2(1) = .46758636      Prob > chi2 = 0.4941
```

The difference between `rreg` and `robreg mm` is that the latter estimates a form of robust estimation that is more resistant to unusual values in the independent variable(s), while the former is not. More precisely put, the latter is an MM-estimator, while the former is an M-estimator.

Also keep in mind is that besides these differences in *estimators*, the estimators can have different *implementations*. As the output above shows, the default implementation of `robreg mm` relies on a "85% efficiency", which refers to the "normal efficiency" of the estimator. This parameter is important, because recall from your econometrics class that when the regression errors satisfy the homoskedasticity assumption (i.e., they have constant variance), OLS provides the most precise estimates among all linear unbiased estimators (i.e., OLS is "BLUE"). When using robust estimators, we should therefore specify how much of this precision (efficiency) we accept to lose when OLS would actually be the appropriate estimator. When OLS is the appropriate estimator but we use an alternative such as an MM-estimator, we lose precision (efficiency) because the robust estimator downweights observations. Because of this downweighting, we lose information that could be relevant for the estimation. The lower we set the "normal

efficiency", the more the information is downweighted. The advantage of this downweighting is that when we have situations as those in the example described above, where OLS is not the best estimator, we can obtain estimates that are less biased and more precise. See Gassen and Veenman [2023] for more detailed discussions and explanations.

What happens when we change the default level of normal efficiency with `robreg mm`? Let's see what happens when we use a 95% normal efficiency:

```
. robreg mm y x, eff(95)
preparing data for subsampling ... done
enumerating 20 candidates 0%....20%....40%....60%....80%....100%
refining 5 best candidates ..... done
iterating RLS ..................................................... done
fitting empty model ... done
computing standard errors ... done

MM regression (95% efficiency)                  Number of obs     =      1,000
                                                Wald chi2(1)      =      51.05
                                                Prob > chi2       =     0.0000
                                                Pseudo R2         =     0.1677
                                                Breakdown point   =         50
                                                M-estimate: k     =  4.6850649
                                                S-estimate: k     =   1.547645
                                                Scale             =  1.0900179

------------------------------------------------------------------------------
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   .1982051   .0277405     7.14   0.000     .1437687    .2526414
       _cons |  -.0493576   .0417241    -1.18   0.237    -.1312347    .0325196
------------------------------------------------------------------------------
Hausman test of MM against S:    chi2(1) = 176.13872      Prob > chi2 = 0.0000
```

Interestingly, the results show us that when we increase the normal efficiency to 95% in this particular example, the estimates are as biased as we saw for OLS. This suggests that it is important to assess the sensitivity of the results to using alternative levels of normal efficiency and that we should not necessarily strive for normal efficiency levels that are high. When we instead lower the normal efficiency to 70%, we can see that the coefficient estimate increases to 0.94 (compared to 0.91 when using 85% normal efficiency). At least as important, we can see that the standard error (t-value) of the coefficient on x declines (increases), which shows how the robust estimation with lower normal efficiency both reduces bias and increases the precision of estimation:

```
. robreg mm y x, eff(70)
preparing data for subsampling ... done
```

```
enumerating 20 candidates 0%....20%....40%....60%....80%....100%
refining 5 best candidates ..... done
iterating RLS ..................... done
fitting empty model ... done
computing standard errors ... done

MM regression (70% efficiency)                  Number of obs     =      1,000
                                                Wald chi2(1)      =     604.66
                                                Prob > chi2       =     0.0000
                                                Pseudo R2         =     0.2245
                                                Breakdown point   =         50
                                                M-estimate: k     =  2.6972206
                                                S-estimate: k     =   1.547645
                                                Scale             =  1.0900179

------------------------------------------------------------------------------
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   .9397855   .0382185    24.59   0.000     .8647877    1.014783
       _cons |  -.0383068   .0369404    -1.04   0.300    -.1107965    .0341829
------------------------------------------------------------------------------
Hausman test of MM against S:    chi2(1) = .07367695      Prob > chi2 = 0.7861
```

In Figure 4.3, I plot the coefficient estimates and their 95% confidence intervals for a broader set of alternatives to OLS. Besides the `rreg` and `robreg mm` implementations shown above, I plot the results from using median regression (with `qreg`), MM-estimations with 50 and 90 percent efficiency, and an S-estimator (`robreg s`) that can be seen as an MM-estimation with the lowest possible normal efficiency (roughly 28.7 percent). The figure illustrates the difference between the estimators that fail versus those that are close to unbiased, as well as the differences in the actual efficiency (precision) versus normal efficiency of the estimators. For this example, we can see that the MM-estimation with 0.70 efficiency is most precise, as evidenced by the smallest confidence interval. See Gassen and Veenman [2023] for more discussion and evidence on the difference between normal and practical efficiency of the estimators.

4.3.2 Increasing precision with robust estimators

Another useful illustration to showcase what robust estimators do relative to OLS, is one where OLS is unbiased but it is simply not as precise as other estimators. This happens when regression errors fail to satisfy the homoskedasticity assumption, which almost always happens when the errors are non-normal. In the following lines, we create the values of x and y in a way that is similar to that in the example from the previous section, except that we now do not

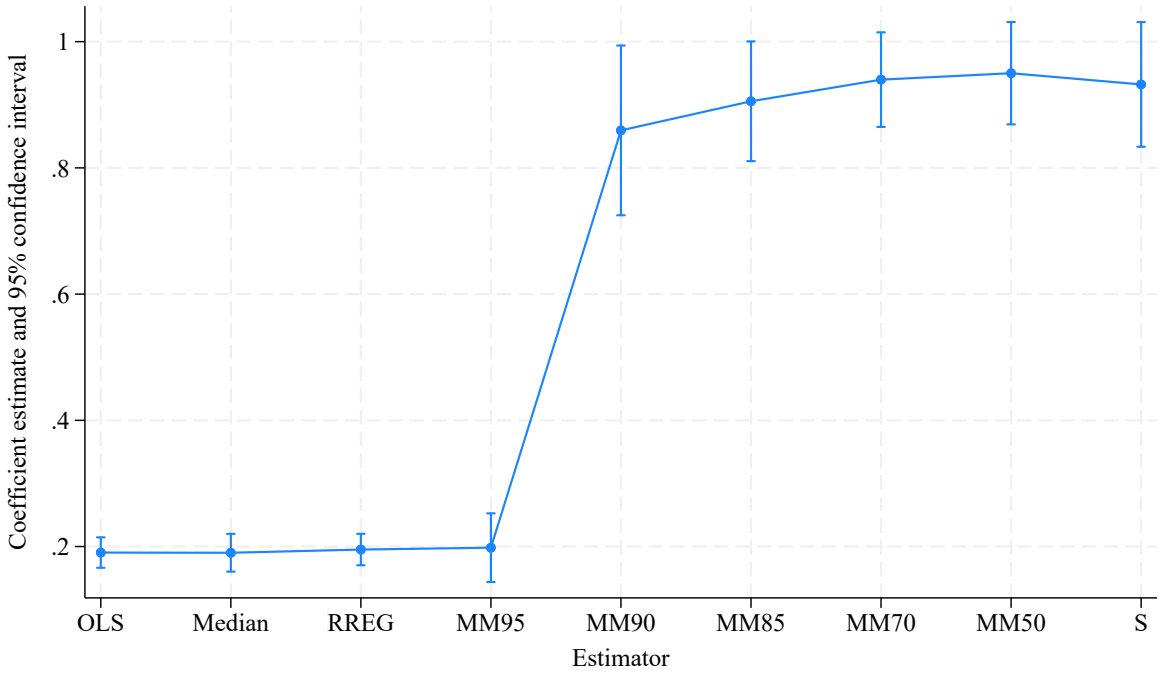


Figure 4.3: *Estimates based on different estimators*

contaminate the x variable and we make the regression errors skewed:

```
gen x=rnormal()
gen y=x+exp(rnormal())
```

Although the regression errors are skewed, OLS estimates are unbiased:

```
. reg y x

      Source |       SS           df       MS      Number of obs   =     1,000
-------------+----------------------------------   F(1, 998)       =    194.97
       Model |  924.363301         1  924.363301   Prob > F        =    0.0000
    Residual |  4731.55711       998  4.74103919   R-squared       =    0.1634
-------------+----------------------------------   Adj R-squared   =    0.1626
       Total |  5655.92041       999  5.66158199   Root MSE        =    2.1774

------------------------------------------------------------------------------
           y | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   .9450322   .0676803    13.96   0.000     .8122202    1.077844
       _cons |   1.516893   .0688552    22.03   0.000     1.381775     1.65201
------------------------------------------------------------------------------
```

But because of the skewed regression errors, the homoskedasticity assumption is violated and OLS is no longer the most precise estimator. As we can see from the difference in confidence intervals, all the displayed alternative estimators are more precise than OLS.[2] Also, although

[2]Because of the focus on the confidence intervals in this graph, I used estimators that estimate standard

the differences are relatively small in this example, we can see that the estimators with lower normal efficiencies again can have higher efficiencies (smaller confidence intervals) in practice.

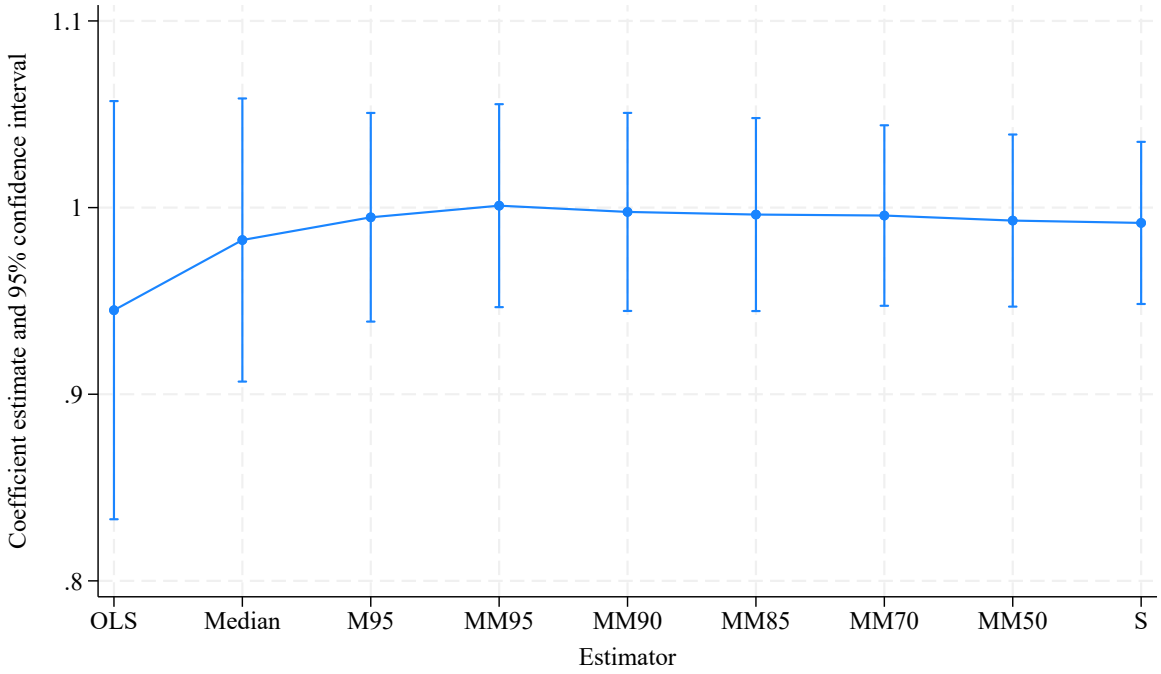


Figure 4.4: *Precision of estimates with skewed regression errors*

4.3.3 Getting the standard errors right with robust estimators

Lastly, it is important to discuss the clustering of standard errors. Before the introduction of an update to `robreg` in 2021 (Jann [2021]), a formal cluster-robust variance estimator was not available for robust estimators. As Gassen and Veenman [2023] discuss, this has led researchers to use an ad-hoc approach that exploited the fact that the *coefficients* obtained from robust estimators can be retrieved using weighted least squares (WLS). By storing each individual observation's weight determined by the robust estimator, some researchers have obtained clustered standard errors by using those weights in a second-step WLS estimation with cluster-robust standard errors.

This WLS approach is inappropriate. To show why, consider the same data structure as in Section 4.3.1 (with contamination in x) but with a dependence in the regression errors that requires us to cluster standard errors by firm (the dataset now consists of 100 hypothetical firms with 10 years of data each). We can see that when we use the appropriate standard error

errors on a similar basis. I therefore used `robreg q` in place of `qreg` and `robreg m y x, eff(95)` in place of `rreg`.

correction now available in `robreg` (Jann [2021]), the standard errors are more conservative (i.e., higher) than when we do not account for clustering of the errors:

```
. robreg mm y x, eff(70)
preparing data for subsampling ... done
enumerating 20 candidates 0%....20%....40%....60%....80%....100%
refining 5 best candidates ..... done
iterating RLS ..................... done
fitting empty model ... done
computing standard errors ... done

MM regression (70% efficiency)                  Number of obs     =      1,000
                                                Wald chi2(1)      =     690.19
                                                Prob > chi2       =     0.0000
                                                Pseudo R2         =     0.2316
                                                Breakdown point   =         50
                                                M-estimate: k     =  2.6972206
                                                S-estimate: k     =   1.547645
                                                Scale             =  1.5479873

------------------------------------------------------------------------------
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   1.015867   .0386682    26.27   0.000     .9399868    1.091747
       _cons |   .0073708   .0521136     0.14   0.888     -.094894    .1096356
------------------------------------------------------------------------------
Hausman test of MM against S:    chi2(1) = .51368697      Prob > chi2 = 0.4735

. robreg mm y x, eff(70) cluster(firm)
preparing data for subsampling ... done
enumerating 20 candidates 0%....20%....40%....60%....80%....100%
refining 5 best candidates ..... done
iterating RLS ..................... done
fitting empty model ... done
computing standard errors ... done

MM regression (70% efficiency)                  Number of obs     =      1,000
                                                Wald chi2(1)      =     300.57
                                                Prob > chi2       =     0.0000
                                                Pseudo R2         =     0.2316
                                                Breakdown point   =         50
                                                M-estimate: k     =  2.6972206
                                                S-estimate: k     =   1.547645
                                                Scale             =  1.5479873

(Std. err. adjusted for 100 clusters in firm)
------------------------------------------------------------------------------
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
```

```
           x |   1.015867   .0585959    17.34   0.000     .8996002    1.132134
       _cons |   .0073708   .1142998     0.06   0.949    -.2194247    .2341663
------------------------------------------------------------------------------
Hausman test of MM against S:     chi2(1) = .55188146      Prob > chi2 = 0.4575
```

The t-value on the coefficient on x drops from 26.27 to 17.34 due to the more conservative standard error estimate, and in Gassen and Veenman [2023] we show this clustered standard error estimate is unbiased. Similarly, bootstrapped cluster-robust standard errors using the insights of Salibian-Barrera and Zamar [2002] implemented in program `roboot` produces similarly unbiased standard error estimates:[3]

```
. roboot y x, nboot(9999) eff(70) seed(1234) cluster(firm)

STEP 1: Obtaining robust regression estimates, residuals, and scale estimate..

STEP 2: Obtaining cluster-robust bootstrapped standard errors...
Status: 0%....20%....40%....60%....80%....100%


MM-estimator with 70% efficiency and bootstrapped clustered SEs

Number of bootstrap samples =  9999                    Number of obs =    1000
Number of clusters (firm) =     100                    Pseudo R2 =      0.2316
------------------------------------------------------------------------------
           y | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   1.015867   .0589569    17.23   0.000     .8988839     1.13285
       _cons |   .0073705   .1158184     0.06   0.949    -.2224382    .2371793
------------------------------------------------------------------------------
SE clustered by firm

Average weight assigned to observations:                0.7527
Fraction of observations with weight < 0.50:            0.1750
Fraction of observations with weight equal to zero:     0.0790
```

Now let's see what happens when we use the inappropriate WLS approach. From the following output, we can see that the standard error (t-value) of the coefficient on x decreases (increases) relative to the situation where we do not cluster the standard errors. In other words, our inferences can become even more biased than when we fail to account for the clustering of regression errors, because the standard error estimate is too small. See Gassen and Veenman [2023] for more discussion and proof for this bias in standard error estimates when using the WLS approach.[4]

[3] The `roboot` program can be downloaded from https://github.com/dveenman/outliers and installed in the subfolder named "r" in your local "PLUS" folder. See Section 2.16 for more information on how to install external packages.

[4] The WLS output also clarifies another important issue with many robust estimators. Because observations

```
. robreg mm y x, eff(70)
[output omitted, same as above]

. predict w70, weights

. reg y x [aw=w70], cluster(firm)
(sum of wgt is 752.7451663073552)

Linear regression                                   Number of obs     =        921
                                                    F(1, 98)          =     867.00
                                                    Prob > F          =     0.0000
                                                    R-squared         =     0.6387
                                                    Root MSE          =     1.1055

(Std. err. adjusted for 99 clusters in firm)
------------------------------------------------------------------------------
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   1.015867   .0345006    29.44   0.000     .9474017    1.084332
       _cons |   .0073708    .072418     0.10   0.919    -.1363404     .151082
------------------------------------------------------------------------------
```

[help robreg] [help qreg] [help rreg]

4.4 Storing statistics and results

Stata not only allows us to display statistics and regression output, but it also allows us to use and store statistics. For example, we may need the median of a variable to create a dummy variable that equals 1 for those observations where the value of the variable is greater than the sample median, and 0 otherwise. In order to do so, we can use Stata's `return` function. For instance, after summarizing a variable we can use the command `return list`:

```
sum nseg, d
return list
```

to get:

```
scalars:
    r(N)          =  15667
    r(sum_w)      =  15667
```

can get weights equal to zero, we see that the WLS estimation relies only on 921 out of the 1000 observations. In other words, roughly eight percent of the sample gets deleted and many more observations receive relatively low weights. Because in actual research settings these downweighted observations are typically not random draws from the population of interest, it is important to keep in mind that the downweighting of observations by robust estimators can have implications for the inferences we draw about that population. See again Gassen and Veenman [2023] for more discussion on this issue and ways to deal with it.

```
    r(mean)       =  2.047998978745133
    r(Var)        =  2.238982814246371
    r(sd)         =  1.496323098213207
    r(skewness)   =  1.287639280651829
    r(kurtosis)   =  3.763740899282668
    r(sum)        =  32086
    r(min)        =  1
    r(max)        =  7
    r(p1)         =  1
    r(p5)         =  1
    r(p10)        =  1
    r(p25)        =  1
    r(p50)        =  1
    r(p75)        =  3
    r(p90)        =  4
    r(p95)        =  5
    r(p99)        =  7
```

Now we can use the `return` function to generate a new variable that is equal to the median for the sample of all observations. Next, we can use this median to generate a dummy variable:

```
sum nseg, d
gen median_nseg=r(p50)
gen dummy=0
replace dummy=1 if nseg>median_nseg & nseg!=.
```

Alternatively, we can directly use the information stored in the return scalar:

```
sum nseg, d
gen dummy2=0
replace dummy2=1 if nseg>r(p50) & nseg!=.
```

We can perform similar actions with the other statistics stored by Stata in `r(...)`. Besides returning summary statistics from the `summarize` command, Stata can also return statistics from regression analyses. For example, typing `ereturn list` (instead of `return list`) after estimating the regression `reg bhr lnmv btm` (see Section 4.2) gives us:

```
scalars:
    e(N)      =  15667
    e(df_m)   =  2
    e(df_r)   =  15664
    e(F)      =  2043.924326847141
    e(r2)     =  .2069603066207028
    e(rmse)   =  .4209142644292234
    e(mss)    =  724.2393141378839
    e(rss)    =  2775.17236515191
    e(r2_a)   =  .2068590502757871
    e(ll)     =  -8671.945636268807
    e(ll_0)   =  -10488.39331379763
```

```
macros:
    e(cmdline) : "regress bhr lnmv btm"
    e(title) : "Linear regression"
    e(vce) : "ols"
    e(depvar) : "bhr"
    e(cmd) : "regress"
    e(properties) : "b V"
    e(predict) : "regres_p"
    e(model) : "ols"
    e(estat_cmd) : "regress_estat"

matrices:
    e(b) :  1 x 3
    e(V) :  3 x 3

functions:
    e(sample)
```

Now we can, for example, create a new variable that contains the adjusted R^2 (.206859) of the regression:

```
gen rsquared=e(r2_a)
```

Although this does not seem very useful at this point, such a function is quite helpful when estimating many regressions in a loop, such as with the estimation of discretionary accruals by industry-year group, as shown in the appendices of this guide. In a similar vein, we can store regression coefficients into new variables:

```
gen b0=_b[_cons]
gen b1=_b[lnmv]
gen b2=_b[btm]
```

Although the t-values are not directly available when we inspect the information displayed after `ereturn list`, we can easily infer them from the coefficients and standard errors, as follows:

```
gen t0=_b[_cons]/_se[_cons]
gen t1=_b[lnmv]/_se[lnmv]
gen t2=_b[btm]/_se[btm]
```

Storing the p-values is a bit trickier, but we can also inspect the information that is stored in `return list`. For the same regression estimation that we ran before, we get:

```
. return list

matrices:
r(table) :  9 x 3
```

By inspecting the contents of this matrix of information, we get the following information:

```
. matrix list r(table)

r(table)[9,3]
              lnmv         btm       _cons
     b   .04499259  -.38180914  -.07431528
    se   .00212325   .00754211   .01648561
     t   21.190455  -50.623617  -4.5078868
pvalue   2.790e-98           0   6.595e-06
    ll   .04083078  -.39659255  -.10662899
    ul    .0491544  -.36702572  -.04200158
    df       15664       15664       15664
  crit   1.9601154   1.9601154   1.9601154
 eform           0           0           0
```

This means that we can store the p-value associated with the coefficient on variable `lnmv` by extracting the information stored in row 4 and column 1 of the matrix.

```
. gen p1=r(table)[4,1]

. sum p1

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
          p1 |     15,667           0           0          0          0
```

This becomes a bit trickier when we need a test to evaluate the value of the coefficients against a value other than zero. After estimating the regression, we can use the following code to store the two-tailed p-value of the coefficient on `lnmv` when tested against the null hypothesis that this coefficient equals, for example, 0.05:

```
. local t=(_b[lnmv]-0.05)/_se[lnmv]

. gen p1_alt=2*ttail(e(df_r), abs(`t'))

. sum p1_alt

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
      p1_alt |     15,667    .0183674           0   .0183674   .0183674
```

Alternatively, we can store the 95 percent confidence interval for the coefficient on `lnmv`:

```
gen ci_lo_1=_b[lnmv]-invttail(e(df_r), 0.025)*_se[lnmv]
gen ci_hi_1=_b[lnmv]+invttail(e(df_r), 0.025)*_se[lnmv]
```

We can also store the predicted and residual values from the regression, which might be useful for further calculations or as inputs to specific variables of interest (e.g., discretionary accruals, see Appendix A.2):

```
predict fit, xb
predict res, resid
```

```
[help return] [help ereturn] [help predict]
```

4.5 Standard error adjustments

As a researchers, we care about the direction, magnitude, and precision of our coefficient estimates. The precision of our coefficients is captured by the standard error, which is an estimate itself. Given the focus on "statistical significance" as a basis for inference, getting the standard errors right is essential.

The basic formula to estimate standard errors in Stata relies on the assumption that regression errors are homoskedastic (i.e., their variance is constant across observations). But this formula is incorrect when errors are heteroskedastic, that is, when the variance of the regression error term varies with independent variable(s) included in the model. To resolve this issue, we can apply the heteroskedasticity-robust variance estimator. This is more commonly referred to as the use of "robust" standard errors and can be implemented by simply adding the option `, r` at the end of the regression command. For example:

```
reg bhr lnmv btm, r
```

which provides:

```
Linear regression                               Number of obs =   15667
F(  2, 15664) = 3267.87
Prob > F      =  0.0000
R-squared     =  0.2070
Root MSE      =  .42091

------------------------------------------------------------------------------
             |               Robust
         bhr |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
        lnmv |   .0449926   .0021563    20.87   0.000     .0407661    .0492191
         btm |  -.3818091   .0072331   -52.79   0.000    -.3959869   -.3676313
       _cons |  -.0743153   .0186353    -3.99   0.000    -.1108427   -.0377878
------------------------------------------------------------------------------
```

Another important issue to consider in many research settings is that, due to the large numbers of observations and the panel structure of the data, we often have significant correlations between our observations, either over time or across (groups of) firms. For instance, a variable such

as book-to-market (`btm`) is highly correlated over time for a specific firm. When we have a panel dataset with multiple firms and multiple years per firm, such correlation leads to a violation of the independence assumption of OLS and causes the default *and* heteroskedasticity-robust standard error estimates to be biased (Petersen [2009] refers to this time-series dependence as a "firm effect"). Most often, this bias is negative, meaning standard errors are understated and t-statistics are overstated. Therefore, it can be that we reject our null hypothesis simply because of understated standard errors caused by within-cluster correlation of variables (see Petersen [2009]; Gow et al. [2010]; Conley et al. [2018]).

To address this concern, we can compute "cluster-robust" or "clustered" standard errors. These are heteroskedasticity-robust standard errors that are additionally corrected for unwanted correlation within a specified dimension. The basic idea of the clustering adjustment is that the errors are allowed to have an unknown arbitrary correlation within each cluster, while we assume that the errors are uncorrelated across clusters. For example, we often adjust the standard error estimates to allow for arbitrary correlations within firm. In the example below, `gvkey` is the variable that identifies firms:

```
. reg bhr lnmv btm, cluster(gvkey)

Linear regression                                       Number of obs =   15667
                                                        F(  2,  4504) = 2361.16
                                                        Prob > F      =  0.0000
                                                        R-squared     =  0.2070
                                                        Root MSE      =  .42091

(Std. Err. adjusted for 4505 clusters in gvkey)
------------------------------------------------------------------------------
             |               Robust
         bhr |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
        lnmv |   .0449926   .0024016    18.73   0.000     .0402844    .0497008
         btm |  -.3818091   .0078652   -48.54   0.000    -.3972288   -.3663894
       _cons |  -.0743153   .0200124    -3.71   0.000    -.1135494   -.0350812
------------------------------------------------------------------------------
```

We can see that the standard errors and t-statistics are different from those reported before, although the inferences are unchanged in this case. In a similar vein, we can adjust for potential clustering by time by replacing `gvkey` with the variable that captures the time dimension of the panel (here: `fyear`) in the `, cluster()` option. Clustering by time can be important when the dependent and independent variables are subject to shocks that affect all (or many) firms at the same point in time, for example, firm performance variables that are affected by shocks such as

the global financial crisis or Covid (Petersen [2009] refers to this cross-sectional dependence as a "time effect"). Because we hardly ever know the underlying correlation structure, it can be helpful to separately perform the estimation with a cluster-adjustment on each of the dimensions and then pick the adjustment that leads to the most conservative t-values.

It is also common to simultaneously cluster-adjust the standard errors on two dimensions, for example by firm and time. This adjustment relies on a simple combination of the cluster-robust variance matrices that are computed for the individual dimensions and their intersection (see Cameron et al. [2011]; Thompson [2011]). Before applying this adjustment, note that it is important to carefully think about the motivation and underlying assumptions required for two-way clustering, which may be untenable (see Conley et al. [2018]). For example, when clustering by firm and time, we should ask ourselves whether it reasonable to assume that the observation of Uber in 2021 is independent of that of Lyft in 2022. There are multiple programs that can do this, but my preference is to make use of `reghdfe`, which is designed for the estimation of models with high-dimensional fixed effects (see Section 4.6). For example, the following estimates a linear regression of earnings announcement returns on an earnings surprise variable, using OLS with standard errors adjusted for clustering by both firm (`gvkey`) and year-quarter (`qid`):[5]

```
. reghdfe bhar01 qes_ibes, cluster(gvkey qid) noab
(MWFE estimator converged in 1 iterations)

HDFE Linear regression                            Number of obs   =     48,586
Absorbing 1 HDFE group                            F(   1,     35) =     582.48
Statistics robust to heteroskedasticity           Prob > F        =     0.0000
                                                  R-squared       =     0.0875
                                                  Adj R-squared   =     0.0875
Number of clusters (gvkey)   =      2,668         Within R-sq.    =     0.0875
Number of clusters (qid)     =         36         Root MSE        =     0.0896

                            (Std. err. adjusted for 36 clusters in gvkey qid)
------------------------------------------------------------------------------
             |               Robust
      bhar01 | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
    qes_ibes |   .0869702   .0036036    24.13   0.000     .0796546    .0942858
       _cons |   .0004002    .000876     0.46   0.651    -.0013781    .0021786
------------------------------------------------------------------------------
```

The `noab` option was added here because `reghdfe` is designed to "absorb" high-dimensional fixed effects, but we don't do that in this estimation.

[5] Note that this is not the same as clustering by firm-quarter!

We should keep in mind that these adjustments work reasonably well only under specific conditions, such as when there is a sufficient number of clusters available (e.g., Cameron et al. [2008]; Petersen [2009]). When there are too few clusters, even the cluster-robust standard errors are biased. In my example above, we have 2,668 clusters in the first (firm) dimension and 36 clusters in the second (time) dimension. The second dimension has significantly fewer clusters than the first dimension, and has a bit less than the typical rule of thumb of a minimum of 40-50 (or "42", see Angrist and Pischke [2009]) that is needed for reliable inference. The use of only 36 clusters might lead to a downward bias in the standard errors, although this is typically still more conservative than not clustering on that dimension at all (in the above example, the t-value increases to 47.17 when I only cluster by firm). The problem of few clusters and bias in cluster-robust standard errors becomes particularly severe when we have about 10–15 (or fewer) clusters in a dimension, and even more so when the size of clusters varies significantly (see Cameron et al. [2008]; MacKinnon et al. [2023]).

4.5.1 Technical notes on programs for two-way clustering

The reason I prefer `reghdfe` for the calculation of cluster-robust standard errors (whether in one dimensions or two dimensions) is that it appropriately applies the "finite-sample correction" and degrees of freedom adjustment for reliable inference. These issues relate to the issue of having few clusters. With G clusters (e.g., 36 time periods), the degrees of freedom of the estimation equal $G - 1$ (i.e., 35). The finite-sample correction is simply a parameter that is used to multiply the cluster-robust variance matrix from which we obtain the standard errors (i.e., the square roots of the diagonal entries of this matrix), in order to reduce the downward bias in the variance estimates. This parameter is a correction that is based on degrees of freedom and is therefore determined by G. The lower the value of G, the higher the adjustment factor and the more important its calculation becomes. At the same time, the degrees of freedom $(G - 1)$ also directly affect our hypothesis testing through the critical value of a t-test.

Assuming we have a panel dataset consisting of 1000 firms and 20 years (10 years), clustering by year implies we have 19 (9) degrees of freedom. With 19 (9) degrees of freedom, the critical value of a two-sided t-test with $\alpha = 0.05$ equals 2.093 (2.262) instead of 1.96. In a similar vein, if we use a two-way cluster-adjustment by firm and time, the degrees of freedom are determined by the number of clusters in the smallest clustering dimension. In this case, $G_{min} = 20$ ($G_{min} = 10$) and thus the degrees of freedom again equal 19 (9).

To illustrate why this all matters, consider the following simulated panel dataset of 1000 firms and 10 years per firm. When using `reghdfe`, I obtain the following results with standard errors clustered by firm and year:

```
. reghdfe y x, cluster(firm year) noab
(MWFE estimator converged in 1 iterations)

HDFE Linear regression                            Number of obs   =     10,000
Absorbing 1 HDFE group                            F(   1,      9) =       5.12
Statistics robust to heteroskedasticity           Prob > F        =     0.0500
                                                  R-squared       =     0.0015
                                                  Adj R-squared   =     0.0014
Number of clusters (firm)    =      1,000         Within R-sq.    =     0.0015
Number of clusters (year)    =         10         Root MSE        =     1.4022

(Std. err. adjusted for 10 clusters in firm year)
------------------------------------------------------------------------------
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   .0396403   .0175235     2.26   0.050    -5.79e-07    .0792812
       _cons |   -.025882   .0319296    -0.81   0.439    -.0981118    .0463478
------------------------------------------------------------------------------
```

The results reveal that the relation between x and y is statistically significant at $p = 0.05$. However, we also see that the t-value is 2.26, which is well above 1.96. This is, again, because the critical value for a t-test with 9 degrees of freedom $(G - 1)$ is 2.262. Also note that the standard error is reasonably close, albeit slightly too low, compared to the true standard error of 0.0179 (given the way the data were simulated).

Let's see what happens when we use two alternative programs that can be used to obtain two-way clustered standard errors: `vce2way` (installed via SSC) and `cluster2` (from Petersen [2009]).[6]

```
. cluster2 y x, fcluster(firm) tcluster(year)

Linear regression with 2D clustered SEs                Number of obs =   10000
                                                       F(  1,  9998) =   15.48
                                                       Prob > F      =  0.0001
Number of clusters (firm) =    1000                    R-squared     =  0.0015
Number of clusters (year) =      10                    Root MSE      =  1.4022
------------------------------------------------------------------------------
           y | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   .0396403   .0168159     2.36   0.018     .0066778    .0726028
```

[6] `cluster2` is here: https://www.kellogg.northwestern.edu/faculty/petersen/htm/papers/se/se_programming.htm.

```
       _cons |   -.025882   .0304011    -0.85   0.395    -.0854742    .0337102
------------------------------------------------------------------------------

. vce2way reg y x, cluster(firm year)

Linear regression                               Number of obs     =     10,000
                                                F(1, 9999)        =      15.48
                                                Prob > F          =     0.0001
                                                R-squared         =     0.0015
                                                Root MSE          =     1.4022

(Std. err. adjusted for clustering on firm and year)
------------------------------------------------------------------------------
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   .0396403   .0168159     2.36   0.018     .0066778    .0726028
       _cons |   -.025882   .0304011    -0.85   0.395    -.0854742    .0337102
------------------------------------------------------------------------------
```

These alternative programs provide identical results, but these results are different from those obtained using `reghdfe`. First, the t-value of 2.36 is higher than the 2.26 we obtained earlier. This is because the standard error estimate is lower, at 0.0168 instead of 0.0175, further away from the true standard error of about 0.0179. This is because the programs apply a less conservative finite-sample correction.[7] Second, the t-value of 2.36 is associated with $p = 0.018$, but this cannot happen with degrees of freedom equal to 9, because the t-value would have to be about 2.886 to reach that p-value. We can see from the post-estimation command `ereturn list` that these programs rely on degrees of freedom equal to 9998 and 9999, respectively, instead of $G - 1 = 9$, which explains the issue.

The `vce2way` command was superseded by `vcemway` (also available via SSC) in June 2019 and this fixed part of the problem. The degrees-of-freedom are now correct at 9 and give us the appropriate p-value. Still, the standard errors are the same as before and are less conservative than those from `regdhfe` due to the difference in finite-sample correction:

[7] The difference in standard errors is explained by the difference in which `regdhfe` vs. `cluster2` and `regdhfe` apply the finite-sample corrections. Recall that the finite-sample correction is made by multiplying the estimated variance matrix with a constant (this constant is defined in equation (4.2) in Section 4.6). Its purpose is to correct for a downward bias in the estimated standard errors. Because the variance matrix estimator with two-way clustering is determined by three sub-matrices ($V_c = V_1 + V_2 - V_{12}$, where V_1 is the variance matrix obtained when clustering on the first dimension, V_2 is the variance matrix obtained when clustering on the second dimension, and V_{12} is the variance matrix obtained when clustering on the intersection of the two dimensions), the difference lies in when the programs apply the finite-sample correction. `cluster2` and `regdhfe` apply the adjustment to each of the sub-matrices first, and then combine them into V_c. `reghdfe` instead first calculates V_c and then applies the adjustment. Because `reghdfe` computes the adjustment factor based on the degrees of freedom in the smallest clustering dimension, which implies a higher adjustment, the standard errors calculated based on `reghdfe` will be more conservative than those obtained from `vce2way` and `cluster2`.

```
. vcemway reg y x, cluster(firm year)

Linear regression                                   Number of obs     =     10,000
                                                    F(1, 999)         =       4.86
                                                    Prob > F          =     0.0277
                                                    R-squared         =     0.0015
                                                    Root MSE          =     1.4022

(Std. err. adjusted for clustering on firm year)
------------------------------------------------------------------------------
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   .0396403   .0168159     2.36   0.043     .0016001    .0776805
       _cons |   -.025882   .0304011    -0.85   0.417     -.094654      .04289
------------------------------------------------------------------------------
Notes:
Std. Err. adjusted for 2-way clustering on firm year
Number of clusters in firm         = 1000
Number of clusters in year         = 10

Stata's default small-cluster correction factors have been applied.
See vcemway for detail.

Residual degrees of freedom for t tests and F tests = 9
```

Overall, these illustrations highlight the importance of choosing the right program to get the standard errors right. They also show why `reghdfe` is my preferred program for two-way clustered standard errors, as it provides somewhat more conservative (i.e., higher) standard errors.

New in Stata in 2023, introduced with version 18, is a built-in option for multiway clustered standard errors. We can see that this new function provides us the same results as `vcemway` (i.e., the appropriate degrees-of-freedom adjustment, but with a somewhat less conservative finite-sample correction than `reghdfe`):

```
. reg y x, vce(cluster firm year)

Linear regression                                   Number of obs = 10,000
Clusters per comb.:                                 Cluster comb. =      3
  min =     10                                      F(1, 9)       =   5.56
  avg =  3,670                                      Prob > F      = 0.0428
  max = 10,000                                      R-squared     = 0.0015
                                                    Adj R-squared = 0.0014
                                                    Root MSE      = 1.4022

(Std. err. adjusted for multiway clustering)
```

```
------------------------------------------------------------------------------
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   .0396403   .0168159     2.36   0.043     .0016001    .0776805
       _cons |   -.025882   .0304011    -0.85   0.417     -.094654      .04289
------------------------------------------------------------------------------
Cluster combinations formed by firm and year.
```

4.5.2 The restricted wild cluster bootstrap

Recall that cluster-robust standard errors can be unreliable with few clusters or when clusters vary substantially in size. One solution to this problem is to use the "restricted wild cluster bootstrap", or "WCR" bootstrap (Cameron et al. [2008]; MacKinnon et al. [2023]), which can be implemented with the program `boottest` developed by Roodman et al. [2019] (available via SSC). To illustrate this program, consider another simulated dataset that again consists of 1000 firms and 10 years per firm. The data are simulated such that there is a strong need to cluster by year, as can be seen from the regression output without and with cluster adjustment, respectively:

```
. reg y x

      Source |       SS           df       MS      Number of obs   =    10,000
-------------+----------------------------------   F(1, 9998)      =    823.35
       Model |  1373.93814         1  1373.93814   Prob > F        =    0.0000
    Residual |  16683.9192     9,998  1.66872567   R-squared       =    0.0761
-------------+----------------------------------   Adj R-squared   =    0.0760
       Total |  18057.8574     9,999  1.80596633   Root MSE        =    1.2918

------------------------------------------------------------------------------
           y | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   .2382644   .0083036    28.69   0.000     .2219876    .2545412
       _cons |   -.174081   .0134251   -12.97   0.000    -.2003968   -.1477652
------------------------------------------------------------------------------

. reg y x, cluster(year)

Linear regression                               Number of obs     =     10,000
                                                F(1, 9)           =       3.98
                                                Prob > F          =     0.0771
                                                R-squared         =     0.0761
                                                Root MSE          =     1.2918

(Std. err. adjusted for 10 clusters in year)
------------------------------------------------------------------------------
```

```
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   .2382644   .1193802     2.00   0.077    -.0317924    .5083212
       _cons |   -.174081    .291538    -0.60   0.565    -.8335857    .4854238
------------------------------------------------------------------------------
```

The true standard error, approximated by the standard deviation of the coefficient on `x` that I obtained from 1000 datasets created with the same data-generating process, is about 0.155. Hence, it is clear that the standard error of 0.008 without cluster-adjustment is severely understated. However, the 0.119 based on clustering by year is also too low, even though we have correctly identified and specified the dimension in which we should cluster.

The WCR bootstrap is a solution to this problem that provides an alternative p-value for cluster-robust inference (see MacKinnon et al. [2023] for an important discussion of situations when it does *not* work well). Similar to the cluster-robust variance estimator that underlies the clustered standard errors we saw above, the WCR bootstrap allows for arbitrary correlation in regression errors within a cluster. What is different is that the WCR bootstrap generates a distribution of t-values based on a bootstrap procedure in which the bootstrap samples mimic the distribution of the original sample. By imposing the null hypothesis when generating the bootstrap samples, the procedure compares the original t-value from the regression (in the above example, $t = 2.00$) to the distribution of t-values that are obtained after imposing the null on the bootstrap samples.[8] The bootstrapped p-value is then based on the fraction of t-values that are larger than the original t-value.

Let's see what happens when we apply the WCR bootstrap by using `boottest` after estimating the regression with clustered standard errors (`reg y x, cluster(year)`). Because `boottest` is computationally extremely efficient, we can easily rely on a very high number of bootstrap replications (e.g., 9999). However, with only 10 clusters, the number of unique permutations of the bootstraps is limited to 1024 (2^{10}), so we use the default of 999 replications here. Also, because there is a random component in the bootstrap procedure, we should set the seed of the random number generator in order to ensure that results are replicable:

```
. boottest x=0, seed(1234567)
```

[8] In the simple example illustrated here, the null hypothesis is that the coefficient on `x` equals zero. In creating the bootstrap samples, setting the coefficient on `x` to zero means that values of the dependent variable are simulated based on simulated regression errors. In the WCR bootstrap, simulated regression errors are obtained by taking the residuals from the null-imposed regression and then multiplying these residuals for all observations in a cluster by a cluster-specific random variable. This cluster-specific random variable is equal to -1 or 1 with equal probability. See Roodman et al. [2019] for more discussion and illustrations.

```
Wild bootstrap-t, null imposed, 999 replications, Wald test, bootstrap
clustering by year, Rademacher weights:
x=0

                    t(9) =     1.9958
                Prob>|t| =     0.1391

95% confidence set for null hypothesis expression: [-.07613, .5128]
```

The program additionally produces a graph that displays the confidence interval computed with the WCR bootstrap. Here, we see that `t(9) = 1.9958` refers to the t-value of 2.00 for `x` we saw earlier. The results tell us that with the WCR bootstrap, the p-value is 0.1391 instead of the 0.077 we obtained with standard cluster adjustment in the regression. In other words, compared to the result based on the (understated) cluster-robust standard error estimate, we now obtain a more conservative p-value and no longer label the coefficient on `x` as statistically significantly different from zero. This is correct, because I simulated the data based on a data-generating process in which `x` and `y` were independent. The 95% confidence interval provided by `boottest` also informs us that the coefficient is not different from zero. Thus, when clustering on a dimension with few clusters (e.g., ten years), this example illustrates that it might be useful to assess whether conclusions about statistical significance are robust to using the WCR bootstrap.

New in Stata 18 is the function `wildbootstrap`, which performs a similar procedure as `boottest`. Using this function, we obtain similar results (the slight difference in p-value can be attributed to the different programs and the random sampling in the bootstrap procedure):

```
. wildbootstrap reg y x, cluster(year) rseed(1234567)

Performing 1,000 replications for p-value for x = 0 ...
Computing confidence interval for x
Lower bound: .........10.........20...... done (26)
note: lower-bound CI achieved F(-0.14) = 0.0240, but target is F(x) = .025.
note: at least one bootstrap t statistic matches the t statistic under the
null; this prevents the CI bound from converging to the target.
Upper bound: .........10.........20... done (23)

Wild cluster bootstrap                          Number of obs      = 10,000
Linear regression                               Number of clusters =     10
                                                Cluster size:
Cluster variable: year                                       min =   1000
Error weight: Rademacher                                     avg = 1000.0
                                                             max =   1000
------------------------------------------------------------------------------
                   y |   Estimate      t  p-value    [95% conf. interval]
```

```
-------------------------+----------------------------------------------------
constraint               |
                   x = 0 |   .2382644    2.00   0.144    -.140662    .5034466
------------------------------------------------------------------------------
```

```
[help robust] [help reghdfe] [help boottest] [help wildbootstrap]
```

4.6 Fixed effects estimation

Many studies with panel data include "fixed effects" in their regression models, such as time (e.g., year, year-quarter, year-month), industry (e.g., 2-digit SIC code), or firm fixed effects (see Breuer and deHaan [2023] for a comprehensive overview of fixed effects in archival research). The basic idea of these fixed effects is as follows. In virtually any research setting, there are confounding factors that cause an omitted variable bias in our coefficient estimates. Sometimes these confounds can be observed and included as control variables in the regression, but often they are not observable or imperfectly observable. Even when the confounders are unobservable, fixed effects estimation allows us to control for these confounding effects. Specifically, assuming these unobservable confounds are constant at some group level, the inclusion of dimension-specific indicator variables allows us to obtain unbiased coefficients because the indicators control for the group-level effects.

The best way to illustrate this issue is based on an adapted version of the conceptual model of Gormley and Matsa [2014, p. 622], where each observation i represents a firm that is part of a broader group g (e.g., industry or country):

$$y_{ig} = \beta x_{ig} + f_g + \varepsilon_{ig}, \tag{4.1}$$

where y is the dependent variable, x is the independent variable, and f is a variable that is unobservable and constant within each group g. Importantly, when f has a nonzero correlation with x, regressing y on x causes β to be biased. The following regression results illustrate this bias when f is not controlled for. The sample of 5,000 firm-observations, which belong to 100 different groups, was simulated based on the model of Gormley and Matsa [2014], where the true $\beta = 1$. The estimated $\hat{\beta} = 1.472$ is clearly different from this true value of 1:

```
. reg y x

      Source |       SS           df       MS      Number of obs   =     5,000
-------------+----------------------------------   F(1, 4998)      =   7466.32
       Model |  20476.9529         1  20476.9529   Prob > F        =    0.0000
```

```
    Residual |  13707.4047     4,998  2.74257798   R-squared       =    0.5990
-------------+----------------------------------   Adj R-squared   =    0.5989
       Total |  34184.3576     4,999  6.83823917   Root MSE        =    1.6561

------------------------------------------------------------------------------
           y | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   1.471638   .0170313    86.41   0.000     1.438249    1.505027
       _cons |  -.1309914   .0234222    -5.59   0.000    -.1769092   -.0850736
------------------------------------------------------------------------------
```

Even though controlling for f is infeasible since we cannot observe its values, when we assume that the values of f are constant within each group (which, by construction, is true in this example), we can simply plug in G separate indicator variables in the regression, i.e., the "fixed effects". These indicator variables take on the value of 1 when the firm-observation belongs to group $g \in 1...G$, and 0 otherwise. By including these indicator variables in the regression, we control for the group-level means of y. Because f is constant at the group level, this means we take out the portion of y that is due to f and, hence, we control for the omitted variable problem even though the omitted variable is unobservable (Gormley and Matsa [2014]; Breuer and deHaan [2023]). At the same time, inclusion of the indicator variables also leads us to take out the group-level means of x.

Inclusion of these indicator variables in the regression can be done by creating separate indicator variables for each group. A simpler approach is to use the `i.` option. Assume that variable `group` identifies the 100 groups in our example, then by adding `i.group` to the regression line causes Stata to include 100 indicator variables identified based on the values of the categorical `group` variable. The following captures a portion of the Stata output (the "..." point to output for the estimates of fixed effects that I omit for brevity):

```
. reg y i.group x

      Source |       SS           df       MS      Number of obs   =     5,000
-------------+----------------------------------   F(100, 4899)    =    286.53
       Model |  29192.9708       100  291.929708   Prob > F        =    0.0000
    Residual |  4991.38679     4,899  1.01885829   R-squared       =    0.8540
-------------+----------------------------------   Adj R-squared   =    0.8510
       Total |  34184.3576     4,999  6.83823917   Root MSE        =    1.0094

------------------------------------------------------------------------------
           y | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
       group |
          2  |  -.2266453   .2023781    -1.12   0.263    -.6233972    .1701065
```

```
           3  |  -.6663727   .2019934    -3.30   0.001     -1.06237   -.2703751
         ...
          99  |  -1.010017   .2020927    -5.00   0.000    -1.406209   -.6138243
         100  |   1.935685    .202994     9.54   0.000     1.537726    2.333644
              |
           x  |   1.009198   .0143832    70.16   0.000     .9810001    1.037395
       _cons  |  -.4121851   .1429535    -2.88   0.004    -.6924381   -.1319322
------------------------------------------------------------------------------
```

This estimation with the indicator variables for the separate groups is also known as "least squares dummy variable" (LSDV) estimation. From the output, we see that the coefficient on `x` is now very close to the true value of 1 and it is not significantly different from 1. This result is consistent with the fixed effects allowing us to control for confounds that are constant within group, even though they are unobserved. We can also see that Stata provides us output for the coefficients on indicator variables 2 through 100, while that for group 1 is omitted. This is because Stata automatically omits one of the groups to prevent perfect collinearity. By omitting this group, the coefficient on the intercept term (`_cons`) provides us the estimated mean of y for group 1 after controlling for the effect of x. The coefficients on the tabulated indicator variables then capture the incremental average y for each group. For example, after controlling for x, the average y for group 2 equals $-0.4121851 - 0.2266453 = -0.6388304$.

Because Stata's choice of which of the indicator variables to exclude can vary across estimation procedures, and because the interpretation of the coefficients on the other indicator variables depends on the one that is excluded, the tabulation of these individual fixed effects coefficients is often not very informative. This (partly) explains why these fixed effect coefficients typically remain "untabulated" in a paper. In addition, because the indicator variable that is excluded becomes part of the intercept term, and the reader of a paper typically does not know which of the indicator variables was excluded, it is better not to tabulate the coefficient on the intercept term in a regression table.

In practice, we are typically not even interested in the fixed effects estimates. We typically only want to ensure that the unobservable group-level effects are controlled for. We can therefore also simplify the estimation by taking out the group-level fixed effects from our y and x variables, by simply demeaning the variables before estimation. As the following Stata code and output illustrates, this gives us the exact same coefficient estimate, but without the additional output of the coefficients on the fixed effects:

```
. egen meany=mean(y), by(group)
```

```
. gen yadj=y-meany

. egen meanx=mean(x), by(group)

. gen xadj=x-meanx

. reg yadj xadj

      Source |       SS           df       MS      Number of obs   =     5,000
-------------+----------------------------------   F(1, 4998)      =   5022.61
       Model |   5015.9632         1   5015.9632   Prob > F        =    0.0000
    Residual |  4991.38678     4,998  .998676828   R-squared       =    0.5012
-------------+----------------------------------   Adj R-squared   =    0.5011
       Total |    10007.35     4,999  2.00187037   Root MSE        =    .99934

------------------------------------------------------------------------------
        yadj | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
        xadj |   1.009198   .0142401    70.87   0.000     .9812809    1.037114
       _cons |  -2.21e-09   .0141328    -0.00   1.000    -.0277064    .0277064
------------------------------------------------------------------------------
```

We can see that after adjusting both variables for their group-level means, we obtain the same coefficient on x as we did before when including the separating indicator variables (1.009198). A difference, however, is that the standard error is now lower than it was before (0.0142401 vs. 0.0143832), which leads to a higher t-value (70.87 vs. 70.16). This difference is caused by the fact that the second (demeaned) estimation does not account for the loss in degrees of freedom due to the estimation of the demeaned variables. In the first estimation with the indicator variable, we include 100 regressors (x and the 99 indicator variables) plus the intercept. In the second (demeaned) estimation, we include only x plus the intercept. This difference has implications for the variance matrix that Stata estimates. In the first estimation, Stata adjusts this variance matrix by multiplying it with $5000/(5000-101)$, while in the second estimation it multiplies the matrix with $5000/(5000-2)$. This difference implies that the adjustment is a factor 1.020208 higher in the first estimation, leading to a higher standard error estimate. Specifically, this implies that the standard error estimate of the second estimation is understated by a factor $\sqrt{1.020208}$ relative to the first estimation. With this result, we can reconcile the difference in t-values, since $70.16 \times \sqrt{1.020208} = 70.87$!

A second difference is the estimate of the coefficient on the intercept term. Because we have estimated the regression after first taking out the group-level means of y, the coefficient on the intercept is now zero. A third difference is the R^2 values, which are now substantially lower. The

R^2 equals 0.8540 in the first estimation, while it is 0.5012 in the second (demeaned) estimation. The difference is caused by the fact that in the first estimation the x variable and the indicator variables combined explain about 85 percent of the variation in y. In the second estimation, the indicator variables no longer explain variation in the dependent variable because they are not included and because the dependent variable is y adjusted for the group-level means. The R^2 for the second (demeaned) estimation is known as the "within" R^2, since it provides an estimation of the fraction of the variation of the dependent variable that is explained within the groups (after taking out the variation across the groups).

Because we typically do not need the fixed effects estimates, Stata has simpler tools to obtain the within-estimator that adjusts the variables for group-level fixed effects. We can simply instruct Stata to "absorb" the fixed effects using the `areg` function as follows:

```
. areg y x, absorb(group)

Linear regression, absorbing indicators           Number of obs     =   5,000
Absorbed variable: group                          No. of categories =     100
                                                  F(1, 4899)        = 4923.12
                                                  Prob > F          =  0.0000
                                                  R-squared         =  0.8540
                                                  Adj R-squared     =  0.8510
                                                  Root MSE          =  1.0094

------------------------------------------------------------------------------
           y | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   1.009198   .0143832    70.16   0.000     .9810001    1.037395
       _cons |  -.1388663    .014277    -9.73   0.000    -.1668556   -.1108771
------------------------------------------------------------------------------
F test of absorbed indicators: F(99, 4899) = 86.411           Prob > F = 0.000
```

This function provides us with exactly the same coefficient and standard error for variable x as we obtained when including the separate group-level indicator variables. Note that the R^2 is also the same, which implies that this is the overall proportion of the variation in y that is explained by x and the fixed effects. Different from the results we saw before is the coefficient on the intercept term, which further underscores the ambiguous nature of this coefficient for us as researchers. In this case, the estimated intercept can be seen as a global intercept ($\hat{\alpha}$) that ensures $\bar{y} = \hat{\alpha} + \bar{x}\hat{\beta}$, where $\bar{y}$ and $\bar{x}$ are the mean values of y and x, respectively.

While `areg` is preferred over the methods discussed earlier because of the standard error calculation and its simplicity, most researchers use the `reghdfe` package from Sergio Correia (see http://scorreia.com/software/reghdfe/index.html). This package is available in Stata's

SSC and can be installed via `ssc inst reghdfe`. It works similar to `areg`, except that it allows for the absorption of more than one dimension of fixed effects (for example, firm- and year-fixed effects) and interactive fixed effects, it allows for multi-way clustering of the standard errors (see Section 4.5), and it provides separate information on the overall R^2 and the within-R^2 of the estimation. For the example above, `reghdfe` provides the following results, which are similar to when we used `areg` except that we obtain some additional information:

```
. reghdfe y x, absorb(group)
(MWFE estimator converged in 1 iterations)

HDFE Linear regression                            Number of obs   =      5,000
Absorbing 1 HDFE group                            F(   1,   4899) =    4923.12
                                                  Prob > F        =     0.0000
                                                  R-squared       =     0.8540
                                                  Adj R-squared   =     0.8510
                                                  Within R-sq.    =     0.5012
                                                  Root MSE        =     1.0094

------------------------------------------------------------------------------
           y | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   1.009198   .0143832    70.16   0.000     .9810001    1.037395
       _cons |  -.1388663    .014277    -9.73   0.000    -.1668556   -.1108771
------------------------------------------------------------------------------

Absorbed degrees of freedom:
-----------------------------------------------------+
 Absorbed FE | Categories  - Redundant  = Num. Coefs |
-------------+---------------------------------------|
       group |       100           0         100     |
-----------------------------------------------------+
```

The results start to differ again between programs once we start combining the absorbtion of fixed effects with cluster-robust standard errors. Consider the same example as above, but now we additionally cluster the standard error at the same level as the fixed effects. We now obtain different standard errors for `areg` and `reghdfe`, respectively:

```
. areg y x, absorb(group) cluster(group)

Linear regression, absorbing indicators           Number of obs     =   5,000
Absorbed variable: group                          No. of categories =     100
                                                  F(1, 99)          = 3976.61
                                                  Prob > F          =  0.0000
                                                  R-squared         =  0.8540
                                                  Adj R-squared     =  0.8510
                                                  Root MSE          =  1.0094
```

```
(Std. err. adjusted for 100 clusters in group)
------------------------------------------------------------------------------
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   1.009198   .0160037    63.06   0.000     .9774429    1.040952
       _cons |  -.1388663   .0002725  -509.55   0.000    -.1394071   -.1383256
------------------------------------------------------------------------------

. reghdfe y x, absorb(group) cluster(group)
(MWFE estimator converged in 1 iterations)

HDFE Linear regression                            Number of obs   =      5,000
Absorbing 1 HDFE group                            F(   1,     99) =    4056.97
Statistics robust to heteroskedasticity           Prob > F        =     0.0000
                                                  R-squared       =     0.8540
                                                  Adj R-squared   =     0.8510
                                                  Within R-sq.    =     0.5012
Number of clusters (group)   =        100         Root MSE        =     1.0094

(Std. err. adjusted for 100 clusters in group)
------------------------------------------------------------------------------
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   1.009198   .0158444    63.69   0.000     .9777589    1.040636
       _cons |  -.1388663   .0002698  -514.67   0.000    -.1394017   -.1383309
------------------------------------------------------------------------------

Absorbed degrees of freedom:
-----------------------------------------------------+
 Absorbed FE | Categories  - Redundant  = Num. Coefs |
-------------+---------------------------------------|
       group |       100         100           0    *|
-----------------------------------------------------+
* = FE nested within cluster; treated as redundant for DoF computation
```

Where does this difference come from? The `reghdfe` program recognizes that the fixed effects in this case are "nested" within the clusters. Given that the cluster-robust variance matrix is already adjusted to account for the loss in degrees of freedom, `reghdfe` does not additionally count the fixed effects as part of the number of estimated parameters (k) to determine the degrees of freedom. This is why we see the output saying that all the 100 group fixed effects are "treated as redundant for DoF computation". The difference in standard errors stems from the difference in the calculation of the finite-sample correction factor, which is used to adjust

the cluster-robust variance matrix:

$$\left(\frac{G}{G-1}\right) \times \left(\frac{N-1}{N-K}\right), \tag{4.2}$$

where N refers to the number of observations, G refers to the number of clusters, and k refers to the number of estimated parameters to determine the degrees of freedom. With `areg`, Stata computes $(100/99) \times (4999/4899)$ because $k = 101$. Instead, `reghdfe` uses $k = 2$ and calculates $(100/99) \times (4999/4998)$. This difference causes the adjustment to the variance matrix in `reghdfe` to be a factor 0.980192 of the adjustment made in `areg`, resulting in lower standard errors. Again, we can reconcile the numbers. Focusing on the t-values, we see that $63.69 \times \sqrt{0.980192} = 63.06$. Although the difference is relatively small and does not change inferences in this example, `reghdfe` is the preferred program since it better accounts for the nesting of the fixed effects within the clusters.

Related to the degrees of freedom issue is the notion of "singleton" observations. Consider the following part of the output obtain in an estimation discussed in Section 4.7:

```
. reghdfe y post_ifrs, absorb(gvkey year) cluster(countryid)
(dropped 364 singleton observations)
...
------------------------------------------------------------------------------
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
   post_ifrs |  -.0991684    .017839    -5.56   0.000    -.1363798   -.0619569
       _cons |   .2628359   .0014286   183.98   0.000     .2598559    .2658158
------------------------------------------------------------------------------
```

The program warns us that it "dropped 364 singleton observations". These are cases where there is only one observation per fixed effect group (Correia [2015]). In other words, in this sample there are 364 firms with only one observation. Because with the demeaning of y this means that we subtract y from itself for that single observation of a firm, the observation does not contribute to the estimation. Although this issue does not affect the coefficient estimate, it *can* have implications for the standard errors. But, in most settings, the implications appear limited unless the number of singleton observations grows very large. In most cases with relatively standard fixed effects structures (but always check whether this is true!), it appears to be safe to retain these singletons. The benefit of doing so is that the regression tables will display the total number of observations in the sample rather than the reduced sample after singletons were dropped. As we see below, when I invoke the `keepsingletons` option when running `reghdfe`

for my data, the standard error on `post_ifrs` is exactly the same. The only thing that slightly changes is the intercept, which we typically do not interpret in these settings anyway:[9]

```
. reghdfe y post_ifrs, cluster(countryid) absorb(gvkey year) keepsin
WARNING: Singleton observations not dropped;
...
------------------------------------------------------------------------------
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
   post_ifrs |  -.0991684    .017839    -5.56   0.000    -.1363798   -.0619569
       _cons |   .2632191   .0014252   184.69   0.000     .2602462     .266192
------------------------------------------------------------------------------
```

`[help areg]` `[help reghdfe]`

4.7 Difference-in-differences (DiD)

4.7.1 Simple DiD settings

A popular research design that is often used to examine the effects of some regulatory intervention, a new accounting standard, or any other event is the difference-in-differences design (see for example Chapter 4 of Roberts and Whited [2012]). In the most basic form, this design can be implemented with a simple two-by-two matrix that displays the average values of the outcome variable for four groups split by two dimensions: (i) a treatment dimension that indicates whether the observation belongs to the treatment group affected by the shock or to the control group not affected by the shock, and (ii) a time dimension that indicates whether the observation is from the period before or after the shock. See Landsman et al. [2012, Table 3] for a nice illustration.

For example, for an international sample of firm-year observations around the mandatory adoption of IFRS in 2005, we can examine pre- and post-IFRS average stock illiquidity of firms in countries that are subject to IFRS adoption (`ifrs==1`) versus those that are not (`ifrs==0`). In the following illustration, we use the `tabulate` command for such a sample to display the average of an illiquidity variable named `zeroreturn`:[10]

[9] Based on equations (1) and (2) of Correia [2015], I compute that for this estimation, including the 364 singletons induces an extremely small downward bias in the standard error of 0.00015 *percent*. Given the standard error of 0.017839, I find that the number of singleton groups would have to increase to almost 5,000 to make the standard error drop to a rounded value of 0.017838 when singletons are included. Hence, the consequence for the standard error estimates of keeping the singletons appears extremely small, at least in my example setting.

[10] For completeness, note that the dataset used here consists of firm-years from 21 countries, of which firm-years in five countries with mandatory IFRS adoption and concurrent changes in enforcement (Christensen et al. [2013] belong to the treatment group (`ifrs==1`)).

```
. gen post=0 if year<2005
(64,098 missing values generated)

. replace post=1 if year>2005
(55,038 real changes made)

. tabulate ifrs post, summarize(zeroreturn) means

Means of zeroreturn

           |        post
      ifrs |         0          1 |     Total
-----------+----------------------+----------
         0 | .23238348  .21638861 | .22415495
         1 | .42692598  .32413876 | .38461387
-----------+----------------------+----------
     Total | .27632782  .23379685 | .25529639
```

From the values displayed in the four cells that are separated by `ifrs` and `post`, we can calculate the difference-in-difference as the change in average outcomes (i.e., the change from `post==0` to `post==1`) for the treatment (`ifrs==1`) group minus the change in average outcomes for the control group (`ifrs==0`).

Alternatively, we can also recover this difference-in-difference estimate by estimating a linear regression with interactions. Specifically, consider the following model for firm i in year t, where each firm is located in country c. We can use this model to estimate the effects of country-level IFRS adoption on some outcome variable y:

$$y_{ict} = \beta_0 + \beta_1 IFRS_c + \beta_2 POST_t + \beta_3 IFRS_c \times POST_t + \varepsilon_{ict} \tag{4.3}$$

Using the same sample of firm-year observations, the simplest way we can estimate this difference-in-difference model is by regressing the outcome variable on the two indicator variables and their interaction. We can do so by either creating the interaction variable ourselves (`post_ifrs`) or by telling Stata to create the interaction for us using "`##`" in between the two variables to be interacted:

```
. gen post_ifrs=post*ifrs

. reg zeroreturn ifrs post post_ifrs, cluster(countryid)

Linear regression                               Number of obs     =    111,301
                                                F(3, 20)          =      80.52
                                                Prob > F          =     0.0000
                                                R-squared         =     0.0779
```

```
                                                Root MSE          =      .2327

(Std. err. adjusted for 21 clusters in countryid)
------------------------------------------------------------------------------
             |               Robust
  zeroreturn | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
        ifrs |   .1945425   .0436291     4.46   0.000     .1035337    .2855513
        post |  -.0159949   .0164295    -0.97   0.342    -.0502663    .0182766
   post_ifrs |  -.0867924   .0179522    -4.83   0.000      -.12424   -.0493447
       _cons |   .2323835   .0237324     9.79   0.000     .1828786    .2818883
------------------------------------------------------------------------------

. reg zeroreturn ifrs##post, cluster(countryid)

Linear regression                               Number of obs     =    111,301
                                                F(3, 20)          =      80.52
                                                Prob > F          =     0.0000
                                                R-squared         =     0.0779
                                                Root MSE          =      .2327

(Std. err. adjusted for 21 clusters in countryid)
------------------------------------------------------------------------------
             |               Robust
  zeroreturn | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
      1.ifrs |   .1945425   .0436291     4.46   0.000     .1035337    .2855513
      1.post |  -.0159949   .0164295    -0.97   0.342    -.0502663    .0182766
             |
   ifrs#post |
       1 1   |  -.0867924   .0179522    -4.83   0.000      -.12424   -.0493447
             |
       _cons |   .2323835   .0237324     9.79   0.000     .1828786    .2818883
------------------------------------------------------------------------------
```

The results suggest that firms that were subject to the IFRS-shock experienced a change in illiquidity that was significantly lower than the change in illiquidity of the control firms. This result is consistent with results in the literature that suggest IFRS adoption was associated with improvements in stock liquidity.[11] What is important to add here is that the coefficient of -0.0868 on the interaction term is exactly the same number as we would get from calculating the difference-in-difference from the two-by-two matrix displayed above. The benefit of the regression over the tabulation is that we now also obtain an estimate of the standard error associated with the difference-in-difference estimator.

A variant of this type of regression that is often estimated in practice is one in which the

[11] More precisely, the sample of IFRS-adopting countries examined in this example consists of the five countries identified by Christensen et al. [2013] for which mandatory IFRS adoption coincided with changes in enforcement. Christensen et al. [2013] find that it is this subset of countries in which the IFRS effect on liquidity is concentrated.

main effects $IFRS$ and $POST$ are "subsumed" by unit and time fixed effects. Specifically, in the same example, the $IFRS$ indicator can be replaced either by country- of firm-fixed effects, while the $POST$ variable can be replaced by year fixed effects.[12] Using the `reghdfe` program (see Section 4.6), we can absorb both firm- and year-fixed effects as follows. Here, firms are identified by variable `gvkey` and the interaction between $IFRS$ and $POST$ is again captured by the variable `post_ifrs` that we created above:

```
.reghdfe zeroreturn ifrs post post_ifrs, cluster(countryid) absorb(gvkey year)
(dropped 364 singleton observations)
note: ifrs is probably collinear with the fixed effects (all partialled-out
>  values are close to zero; tol = 1.0e-09)
note: post is probably collinear with the fixed effects (all partialled-out
>  values are close to zero; tol = 1.0e-09)
(MWFE estimator converged in 7 iterations)
note: ifrs omitted because of collinearity
note: post omitted because of collinearity

HDFE Linear regression                            Number of obs   =    110,937
Absorbing 2 HDFE groups                           F(   1,     20) =      30.90
Statistics robust to heteroskedasticity           Prob > F        =     0.0000
                                                  R-squared       =     0.8175
                                                  Adj R-squared   =     0.7978
                                                  Within R-sq.    =     0.0260
Number of clusters (countryid) =         21       Root MSE        =     0.1089

(Std. err. adjusted for 21 clusters in countryid)
------------------------------------------------------------------------------
             |               Robust
  zeroreturn | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
        ifrs |          0  (omitted)
        post |          0  (omitted)
   post_ifrs |  -.0991684    .017839    -5.56   0.000    -.1363798   -.0619569
       _cons |   .2628359   .0014286   183.98   0.000     .2598559    .2658158
------------------------------------------------------------------------------

Absorbed degrees of freedom:
-----------------------------------------------------+
 Absorbed FE | Categories  - Redundant  = Num. Coefs |
-------------+---------------------------------------|
       gvkey |     10770       10770           0    *|
        year |        15           0          15     |
-----------------------------------------------------+
* = FE nested within cluster; treated as redundant for DoF computation
```

[12] Note that this is highly context-specific. Namely, in some research settings it might be that the $POST$ variable does not perfectly align with calendar or fiscal years, as in Christensen et al. [2013]. This means that you should always gain a deep understanding of your institutional setting first before estimating these types of regressions. It also further underscores the importance of not blindly following the examples I provide here!

This difference-in-difference coefficient of -0.0992 supports the same conclusion as before that IFRS adoption is associated with improved stock liquidity. Compared with the simple estimation performed earlier, we can see that inclusion of the firm- and year-fixed effects indeed causes the main effects of `post` and `ifrs` to drop out of the estimation, because these are perfectly collinear with the fixed effects. Also note the substantial increase in explanatory power (R^2) as a result of the inclusion of the fixed effects instead of the simple main effect variables.[13]

This type of difference-in-difference estimation, in which the main effects for the treatment and time dimensions are subsumed by fixed effects, is often labeled a "generalized" difference-in-difference estimation (e.g., Breuer and deHaan [2023]). This estimation with fixed effects can be particularly useful in settings where identification comes from the differential timing of treatment for different groups of observations. Such a "staggered" difference-in-difference design and its implementation is discussed in Section 4.7.3.

4.7.2 Testing for parallel trends

The key identifying assumption underlying the use of a difference-in-difference estimator is that the average changes in the outcome variable would have been the same for the treatment and control groups in the absence of treatment (e.g., Roberts and Whited [2012]).[14] This assumption is known as the parallel trends assumption. Unfortunately, because the parallel trends assumption focuses on unobservable counterfactual outcomes (e.g., the change in liquidity post-2005 for IFRS-adoption firms in the hypothetical case that they had not adopted IFRS), this assumption is not testable.

As Roberts and Whited [2012] explain, a solution to this problem is take a visual approach in which we compare the relative trends in the outcome variable for the treatment and control groups in the pre-treatment period. If the trends already diverge in the pre-treatment period in a way similar to the difference-in-difference effect we find, we have a problem because the parallel trends assumption is unlikely to hold (see for example Figure 2 of Roberts and Whited [2012, Chapter 4]). For the IFRS example from Section 4.7.1, we can do this by separately

[13] Additionally note that, in line with the discussion in Section 4.6, we see that because the firm fixed effects are nested within the country clusters (because a firm has its headquarters in one country in the data), they are considered "redundant" in computing the degrees of freedom. The year fixed effects are not nested within the clusters, so these *do* count in determining the degrees of freedom.

[14] See also Armstrong et al. [2022, Appendix A] for a careful discussion of the different assumptions underlying difference-in-difference estimation.

plotting the average values of the illiquidity variable for the treatment and control groups over time. Figure 4.5 presents such a plot, which was created using the following block of code:[15]

```
collapse (mean) zeroreturn, by(ifrs year)

global options "xline(2005) legend(on order(1 2) label(1 "IFRS==1") label(2
↪  "IFRS==0") cols(2) position(12))"

sort year
graph twoway connect zeroreturn year if ifrs==1, $options || ///
    connect zeroreturn year if ifrs==0, lpattern(dash) || ///
    lfit zeroreturn year if ifrs==1 & year<2005 || ///
    lfit zeroreturn year if ifrs==0 & year<2005 || ///
    lfit zeroreturn year if ifrs==1 & year>2005 || ///
    lfit zeroreturn year if ifrs==0 & year>2005
```

A comparison of the trends in Figure 4.5 suggests that the negative difference-in-difference effect found in Section 4.7.1 is unlikely to be caused by a violation of parallel trends. Before treatment year 2005, we can actually see that the trends move in opposite direction of the negative effect found in the difference-in-difference estimation. Although this result suggests that the pre-trends are not exactly parallel, the visual inspection does provide us with some confidence that the divergence in the trends after treatment year 2005 is unlikely to be the result from a continuation of the trends in pre-treatment outcomes for the treatment and control groups.

We can also test the difference in pre-trends using significance tests. For example, we can test whether the time trend in average outcomes differs significantly between the two groups in the years before 2005. The following code shows how we can assess the divergence in pre-trends statistically using interactions between the `ifrs` indicator and a continuous (categorical) `time` variable. The results indicate that we cannot rejected the null hypothesis of parallel pre-trends, as the coefficient on the interaction between `ifrs` and `time` is not statistically significant at conventional levels ($p = 0.102$):

```
. sum year

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
        year |    120,361    2004.827    4.362174       1997       2012

. gen time=1+year-r(min)

. reg zeroreturn i.ifrs##c.time if year<2005, cluster(countryid)
```

[15] See Section 4.11 for more details on how to create graphs in Stata.

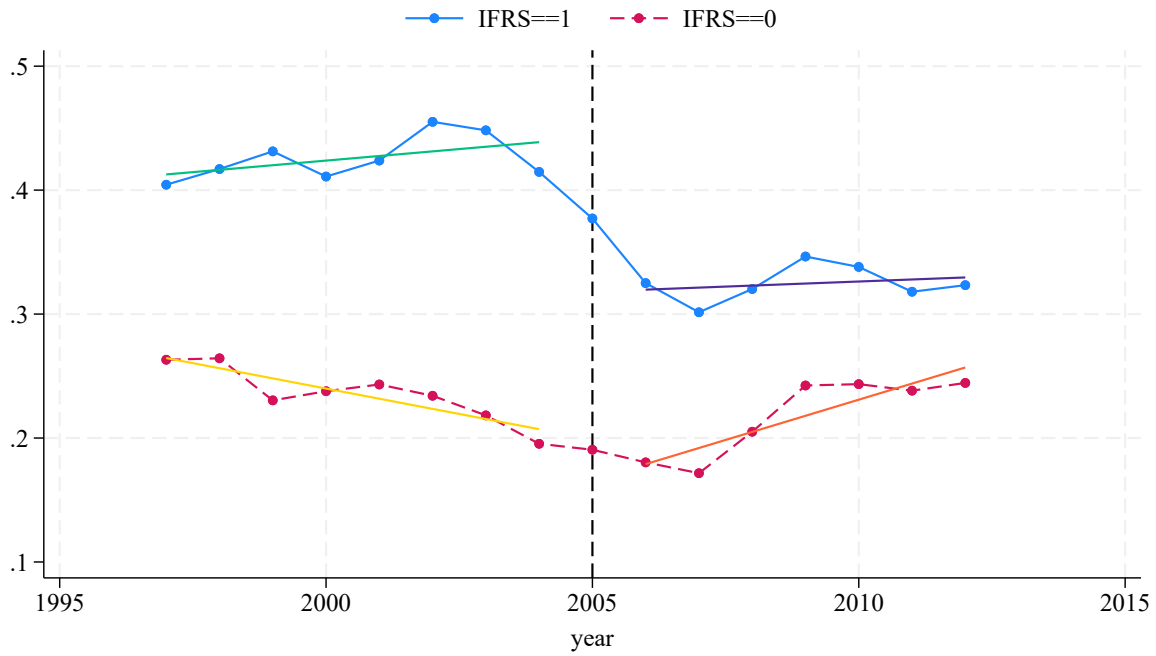


Figure 4.5: *Evaluation of trends pre- and post-treatment (IFRS example)*

```
Linear regression                               Number of obs     =     56,263
                                                F(3, 20)          =      18.01
                                                Prob > F          =     0.0000
                                                R-squared         =     0.1145
                                                Root MSE          =     .23115

(Std. err. adjusted for 21 clusters in countryid)
------------------------------------------------------------------------------
             |               Robust
  zeroreturn | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
      1.ifrs |   .1370058    .052788     2.60   0.017      .026892    .2471197
        time |  -.0083804   .0067808    -1.24   0.231    -.0225249    .0057642
             |
 ifrs#c.time |
          1  |   .0118943   .0069317     1.72   0.102    -.0025649    .0263535
             |
       _cons |    .273432   .0327904     8.34   0.000     .2050325    .3418315
------------------------------------------------------------------------------
```

Because the parallel trends assumption is not testable, an alternative approach that researchers often take is to perform placebo tests in which the treatment effect is tested around a year in which treatment should not have an effect. In the sample used for the IFRS example in Section 4.7.1, the pre-IFRS period runs from 1997–2004. Hence, we could for example split the sample into two periods 1997–2000 and 2001-2004 and pretend as if treatment would take place

at the end of the year 2000. If a violation of parallel trends (or related, any other endogeneity concern) causes the difference-in-difference estimate found earlier to be biased, we might observe a similar difference-in-difference estimate for this placebo treatment. The results below suggest this is not the case and provide some more assurance regarding the liquidity effects of IFRS documented earlier. In fact, the difference-in-difference appears to be positive instead of negative (albeit only borderline significant at $p < 0.10$):

```
. gen placebo=0 if year<2005
(64,098 missing values generated)

. replace placebo=1 if ifrs==1 & year>2000 & year<2005
(6,793 real changes made)

. reghdfe zeroreturn placebo, cluster(countryid) absorb(gvkey year)
(dropped 1296 singleton observations)
(MWFE estimator converged in 6 iterations)

HDFE Linear regression                            Number of obs   =     54,967
Absorbing 2 HDFE groups                           F(   1,     20) =       3.13
Statistics robust to heteroskedasticity           Prob > F        =     0.0922
                                                  R-squared       =     0.8714
                                                  Adj R-squared   =     0.8453
                                                  Within R-sq.    =     0.0091
Number of clusters (countryid) =         21       Root MSE        =     0.0966

(Std. err. adjusted for 21 clusters in countryid)
------------------------------------------------------------------------------
             |               Robust
  zeroreturn | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
     placebo |   .0460763   .0260493     1.77   0.092    -.0082616    .1004143
       _cons |   .2709885   .0031581    85.81   0.000     .2644007    .2775762
------------------------------------------------------------------------------
```

Finally, another common approach that is often used to address the parallel trends assumption is the following. Instead of testing the difference-in-difference using the interaction between the treatment variable and the variable indicating the post-treatment period (in the example here: `ifrs` and `post`), it is common for researchers to replace the `post` variable by separate indicators for years in the pre- and post-treatment windows. For example, the following code shows how we can obtain annual estimates of the difference in average outcomes between the treatment and control group.

```
forvalues i=2001(1)2010{
    gen p_`i'=0
    replace p_`i'=ifrs if year==`i'
```

```
}

reghdfe zeroreturn i_* if year>=2000 & year<=2010, cluster(countryid)
↪  absorb(gvkey year) noconstant
coefplot, vertical yline(0)
```

The graph produced by the program `coefplot` (available via `ssc inst coefplot`), which takes as input the results from a previous regression estimation (in this case produced by `reghdfe`), is shown in Figure 4.6.

Figure 4.6: *Dynamic estimates of treatment effects (IFRS example)*

The results displayed in Figure 4.6 suggest that a significant liquidity effect of IFRS adoption appears to materialize as of year 2007, where the 95%-confidence interval associated with the point estimate (the vertical line) no longer crosses the zero line.

It is important to note for this analysis—and any similar analysis that is used to identify "dynamic" treatment effects like these—is typically highly sensitive to the choice of the time period that is omitted to provide a baseline. In this example displayed in Figure 4.6, all coefficient estimates (the dots) and their 95% confidence intervals (the vertical lines) are obtained relative to the baseline year 2000. If we would have chosen a different baseline year, the plot would look very different. Also note that although tests like these are frequently presented in research, they are often somewhat difficult to interpret in the context of the parallel trends assumption. The reason is that while in Figure 4.5 we could clearly see *differences in average*

trends (which is what we are interested in when evaluating parallel trends), in Figure 4.6 we can only see the *trend in the average differences*.

4.7.3 "Staggered" DiD settings

The previous section illustrated the case in which, at one point in time, a subset of observations receive some treatment while others do not. It is also common for studies to exploit settings in which the treatment assignment occurs at different points in time for different groups of observations. One example is Christensen et al. [2013], which exploits the differential timing of firms' mandatory IFRS adoption that is caused by differences in fiscal-year ends. The benefit of these staggered settings is that, while in a single-DiD setting there may be arguments to suggest that treated observations are differentially affected by some confounding event when compared to control observations (for example, the IFRS countries had concurrent changes in enforcement), it is much harder to argue that such confounders had the same differential impact on the outcome variable for the treatment and control observations at each point in time that a treatment group received the treatment. Moreover, in these staggered DiD settings we can use not only the never-treated observations as a control group, but also the observations of the treatment group before they receive the treatment.

One practical challenge with these staggered settings is that we do not have a clear pre- and post-period for observations that are never subject to the treatment. In the simple IFRS setting discussed in Section 4.7.1, we could simply label `post` for control observations as 0 or 1 depending on whether the observations was from before or after 2005. But when treatment occurs at different points in time, we can no longer do this. Fortunately, the fixed effects in the generalized difference-in-difference framework solve this issue and still allow us to identify the difference-in-difference (at the same time, heading concerns with these types of designs, as discussed shortly).

Consider, for example, a dataset that is simulated following "Simulation 3" of Baker et al. [2022]. In this dataset, a treatment effect is induced in an accounting variable (`roa`) in a staggered fashion for different groups of firms at different points in time. The effect of treatment is a permanent increase in `roa` that is constant for all treatment groups.[16] Given this differ-

[16]This dataset is simulated based on a large sample of Compustat firm-years in which synthetic values of a return-on-assets variable (ROA) are generated by randomly drawing firm-fixed effects, year-fixed effects, and ROA residuals (see Baker et al. [2022, Section 3.1.1] for more details). Firm-years are then randomly assigned to 50 "states" and these states are randomly assigned to three different treatment groups that each receive a constant positive treatment (an increase in ROA) after the year of treatment.

ential timing of the treatment for the different groups, the generalized difference-in-differences framework with fixed effects for units (firms) and time (years) can help us to identify the average treatment effect that is simulated in the data. The treatment effect is identified from the coefficient on variable `post_treat`, which is an indicator variable set equal to 1 for firms after they are subject to the treatment, and 0 otherwise. Given a true effect that was seeded into the data of 0.154, we see that the twoway fixed difference-in-differences estimation nicely identifies this effect:

```
. reghdfe roa_sim post_treat, absorb(gvkey fyear) cluster(stateid)
(MWFE estimator converged in 9 iterations)

HDFE Linear regression                            Number of obs   =    176,736
Absorbing 2 HDFE groups                           F(   1,     49) =    9447.38
Statistics robust to heteroskedasticity           Prob > F        =     0.0000
                                                  R-squared       =     0.7583
                                                  Adj R-squared   =     0.7406
                                                  Within R-sq.    =     0.0456
Number of clusters (stateid) =         50         Root MSE        =     0.1713

(Std. err. adjusted for 50 clusters in stateid)
------------------------------------------------------------------------------
             |               Robust
     roa_sim | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
  post_treat |   .1528581   .0015727    97.20   0.000     .1496978    .1560185
       _cons |  -.0344601   .0007823   -44.05   0.000    -.0360321    -.032888
------------------------------------------------------------------------------
```

The twoway fixed difference-in-differences estimation works well in this example because the treatment effect is constant. As discussed by Barrios [2021], Baker et al. [2022], and others, this estimation works less well when treatment effects vary across the different groups that receive the treatment at different points in time, and/or when the treatment effects for dynamic (i.e., they are not instantaneous but take time to grow larger). To illustrate this concern, we can change the simulated sample to now follow "Simulation 6" of Baker et al. [2022]. In this alternative setup, the treatment effect is dynamic and it varies across the different treatment groups. For this sample, we can see that even though the treatment effect that is simulated in the data is positive, the difference-in-differences results suggest that the effect is, surprisingly, significantly negative:

```
. reghdfe roa_sim post_treat, absorb(gvkey fyear) cluster(stateid)
(MWFE estimator converged in 9 iterations)
```

```
HDFE Linear regression                            Number of obs   =    176,736
Absorbing 2 HDFE groups                           F(   1,     49) =      29.68
Statistics robust to heteroskedasticity           Prob > F        =     0.0000
                                                  R-squared       =     0.7557
                                                  Adj R-squared   =     0.7377
                                                  Within R-sq.    =     0.0003
Number of clusters (stateid) =         50         Root MSE        =     0.1755

(Std. err. adjusted for 50 clusters in stateid)
------------------------------------------------------------------------------
             |               Robust
     roa_sim | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
  post_treat |  -.0127942   .0023484    -5.45   0.000    -.0175136   -.0080749
       _cons |   .0369591   .0011682    31.64   0.000     .0346116    .0393066
------------------------------------------------------------------------------
```

This estimated effect is clearly biased.[17] As discussed and shown by Baker et al. [2022], there are several practical solutions available for Stata that can alleviate this bias. Two of these solutions are based on Callaway and Sant'Anna [2021] and Sun and Abraham [2021], respectively, which are available via the Stata packages `csdid` and `eventstudyinteract` (both available via Stata's SSC).

For the purpose of this guide, I will illustrate how to implement a third solution, which is the "stacked" regression estimator. The idea of this solution is to identify the different groups of observations ("cohorts") that receive treatment at a different point in time. Then for each of these cohorts we look for a clean 2×2 comparison where the treatment observations are those from the specific cohort and the control observations are those from observations from (i) other cohorts that have not yet been treated or (ii) those observations that will never be treated during the sample. If it happens to be the case that there is no group that will never get the treatment (i.e., there is no "never-treated" group), this means that we will not be able to use the cohort that gets treated last. This is because at the time of the last treatment, there are no potential control observations left unless there are observations that will never be subject to the treatment. For each treatment cohort, we create a separate dataset after which we "stack" the different datasets. Finally, the fixed effects in the generalized DiD framework are replaced by unit and time fixed effects that are specific to the cohort in the stacked dataset.

For the "Simulation 6" sample discussed above, we have a dataset that includes a firm

[17] Please refer to Barrios [2021], Baker et al. [2022], and references in these studies for further explanation on the cause of this bias. Simply put, the problem is that the generalized difference-in-difference estimation sometimes compares observations at the time of their treatment to "control" observations that were already treated at an earlier point in time.

identifier (`gvkey`), time identifier (`fyear`), a categorical variable `treatgr` that captures the treatment cohort to which the observation belongs, and the year (`treatyr`) as of which treatment takes is effective for the cohort. The treatment cohort variable `treatgr` is ranked such that value 1 is assigned to cohort that is treated first, value 2 to the cohort that is treated next, and so on. If part of the sample never receives the treatment, we assign that cohort the value of 0. In this specific sample, there are three cohorts and no never-treated observations, so `treatgr` runs from 1 through 3. In the stacking procedure that follows below, the lack of a never-treated group means that we will only stack datasets for the first two cohorts because, again, there are no suitable control observations for the last-treated cohort.

Let's first have a look at some descriptive for the different treatments:

```
. tabstat treatyr, by(treatgr) stats(N mean)

Summary for variables: treatyr
Group variable: treatgr

 treatgr |         N      Mean
---------+--------------------
       1 |     59056      1989
       2 |     63735      1998
       3 |     53945      2007
---------+--------------------
   Total |    176736   1997.74
------------------------------
```

Next, we can use the following code to confirm the number of clean 2×2 datasets that we can identify for the stacking procedure:

```
. sum treatgr

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
     treatgr |    176,736    1.971081    .7990899          1          3

. scalar cohorts=r(max)

. sum post_treat if treatgr<r(max) & fyear<treatyr

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
  post_treat |     45,780           0           0          0          0

. if (r(max)==0) {
      scalar cohorts=cohorts-1
  }
```

```
. di cohorts
2
```

In the following step, we should create two separate datasets based on the two clean 2×2 comparisons. One way we can do this is by applying procedures specific to each cohort and then saving two separate cohort-specific datasets. But there is a more straightforward solution that does not require the saving of separate datasets. Instead, we can use the `expand` command to duplicate the original dataset and then make cohort-specific adjustments to the two datasets that we have "stacked" together using this command. Specifically, we can proceed as follows (output not shown for brevity):

```
gen n=_n
expand cohorts
bysort n: gen dataset=_n

gen keep=1 if treatgr==dataset
replace keep=1 if treatgr==0
sum dataset
local k=r(max)
forvalues i=1(1)`k'{
    replace keep=1 if dataset==`i' & treatgr>`i' & fyear<treatyr
}
keep if keep==1
```

The first three lines are used to transform the original dataset into a larger one, depending on the number of cohorts we will use in the stacking procedure. In this case we duplicate the data only once because we have two cohorts. The new variable `dataset` refers to either 1 or 2, and it refers to the treatment cohort that is being used in the clean 2×2 comparison.

The next lines that include the `forvalues` loop are used to create a temporary variable that captures, for each of the datasets, whether an observation belongs to the treatment group or whether it is available as a clean control because it is not (yet) treated. Finally, we retain only the observations that are treated in the specific cohort or those that are clean controls.

Finally, we create new variables that are needed as inputs for the estimation of cohort-specific unit and time fixed effects. We do so by simply creating a new categorical variable based on the unique combinations of the dataset identifier with unit and firm identifiers:

```
egen gvkey_stacked=group(gvkey dataset)
egen fyear_stacked=group(fyear dataset)
```

After having created these variables, we can now perform the stacked regression estimation using `reghdfe` on the expanded dataset with the cohort-specific fixed effects:

```
. reghdfe roa_sim post_treat, cluster(stateid) absorb(gvkey_stacked
>fyear_stacked)
(dropped 224 singleton observations)
(MWFE estimator converged in 11 iterations)

HDFE Linear regression                            Number of obs   =    241,406
Absorbing 2 HDFE groups                           F(   1,     49) =     617.16
Statistics robust to heteroskedasticity           Prob > F        =     0.0000
                                                  R-squared       =     0.7610
                                                  Adj R-squared   =     0.7411
                                                  Within R-sq.    =     0.0065
Number of clusters (stateid) =         50         Root MSE        =     0.1728

(Std. err. adjusted for 50 clusters in stateid)
------------------------------------------------------------------------------
             |               Robust
     roa_sim | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
  post_treat |   .0771349   .0031049    24.84   0.000     .0708953    .0833745
       _cons |  -.0120592   .0009905   -12.17   0.000    -.0140497   -.0100687
------------------------------------------------------------------------------

Absorbed degrees of freedom:
-----------------------------------------------------+
   Absorbed FE | Categories  - Redundant  = Num. Coefs |
---------------+-------------------------------------|
 gvkey_stacked |     18461       18461           0    *|
 fyear_stacked |        72           0          72     |
-----------------------------------------------------+
```

In sharp contrast to the negative coefficient we found earlier based on the standard two-way fixed effects estimation, we now find the positive treatment effect that was induced into the data. This example therefore nicely illustrates the benefits of using the stacked regression estimator in the staggered DiD setting where treatment effects are heterogeneous and dynamic. Note, however, that this difference-in-difference estimate is still slightly biased due to the weighting of the stacked regression estimator. See Baker et al. [2022] for further discussion.

`[help reghdfe]` `[help csdid]` `[help eventstudyinteract]` `[help expand]`

4.8 Propensity score matching

When including control variables in a linear regression of some outcome variable on a binary treatment indicator, these control variables address endogeneity concerns only to the extent that: (a) there are no other *unobserved* factors that affect the treatment and could therefore confound

inferences; and (b) the relation between the control variable and the outcome and independent variable(s) of interest is linear. Matching methods alleviate the inferential concerns caused by the assumption of a linear functional form in standard linear regressions (see, e.g., Angrist and Pischke [2009] and Shipman et al. [2017]). They do not help control for unobserved confounding factors.[18] The most common matching methods used in the literature are currently propensity-score matching and entropy balancing. The current section focus on the former, while Section 4.9 focuses on the latter.

4.8.1 Matching without replacement

A frequently used program for propensity score matching in Stata is `psmatch2`. If not installed already, type `ssc inst psmatch2` to install it. Because there are many degrees of freedom with the implementation of propensity-score matching (see DeFond et al. [2016] and Shipman et al. [2017]), I will only illustrate how the command can be implemented. When implementing the procedure yourself, make sure to first fully understand how the method works conceptually, as well as the benefits and drawbacks of using propensity-score matching methods. Carefully evaluate the help file for all details and options.

The code below is based on Leung and Veenman [2018] and was used to create part of the contents of Table 8 in that paper. The dataset in `lvdata.dta`, which is opened in the code below, contains a `treatment` indicator variable. This variable equals 1 for all "treatment" firm-quarters, and 0 for a control group of firm-quarter observations. The goal of the matching procedure is to match each treatment firm-quarter to a control firm-quarter, and then to compare an outcome variable `cfop1` for these observations. The matching is done by minimizing the average difference in values of a set of covariates between treatment and control observations. Note that because not all of the treatment observations have a suitable match available, some observations remain unmatched.

To understand the basic idea of the propensity score, we can first estimate a logit regression in which we explain whether an observation is identified in the treatment group based on observable firm characteristics. Below is part of the output from this estimation.

```
use "$path\InFiles\lvdata.dta", clear
. logit treatment timedum_* lnta btm lnage salesgr lnsdearn spdum rnd div cfo
↪  acc expsc inta depr ng_indus
```

[18] This is a significant limitation of matching methods when compared to, for example, fixed effects estimation or difference-in-differences analyses. Also note that some scholars advocate that propensity score matching should not be used at all [e.g., King and Nielsen, 2019].

```
note: timedum_36 omitted because of collinearity.
Iteration 0:   log likelihood =  -4583.162
Iteration 1:   log likelihood = -3057.2911
Iteration 2:   log likelihood = -2753.6509
Iteration 3:   log likelihood = -2734.7023
Iteration 4:   log likelihood =  -2734.484
Iteration 5:   log likelihood = -2734.4838

Logistic regression                                     Number of obs =   9,259
                                                        LR chi2(49)   = 3697.36
                                                        Prob > chi2   =  0.0000
Log likelihood = -2734.4838                             Pseudo R2     =  0.4034

------------------------------------------------------------------------------
  treatment | Coefficient  Std. err.      z    P>|z|     [95% conf. interval]
------------+-----------------------------------------------------------------
  timedum_1 |   1.273881   .3507545     3.63   0.000      .586415    1.961347
        ...
       lnta |   .5459653   .0291749    18.71   0.000     .4887836    .6031471
        btm |   .0522964   .0812177     0.64   0.520    -.1068873    .2114801
      lnage |  -.2490797   .0569752    -4.37   0.000     -.360749   -.1374104
    salesgr |   .0606946   .0664283     0.91   0.361    -.0695025    .1908917
   lnsdearn |  -.0438052   .0370056    -1.18   0.237    -.1163348    .0287244
      spdum |   .6638392   .0746219     8.90   0.000     .5175829    .8100954
        rnd |   4.461092   1.532898     2.91   0.004     1.456668    7.465517
        div |  -22.04053   8.673314    -2.54   0.011    -39.03991   -5.041143
        cfo |   20.68008   1.212625    17.05   0.000     18.30338    23.05678
        acc |   7.285523   1.030672     7.07   0.000     5.265443    9.305602
      expsc |   80.25899   6.616544    12.13   0.000     67.29081    93.22718
       inta |   2.177683   .1764341    12.34   0.000     1.831879    2.523488
       depr |  -18.73584   4.174841    -4.49   0.000    -26.91838    -10.5533
   ng_indus |   7.235906   .2597329    27.86   0.000     6.726839    7.744974
      _cons |  -9.171973   .4257103   -21.55   0.000    -10.00635   -8.337596
------------------------------------------------------------------------------
```

The results suggest there are substantial and significant differences in characteristics between the groups. To reduce these differences, we can use propensity score matching, which is a procedure that identifies observations in the control group (`treatment==0`) that have a similar conditional probability of being treated, based on the observed covariates, as observations in the treatment group (`treatment==1`). The goal is to have a treatment sample and a control sample for which we no longer observe significant differences in the observed covariates. The only difference between the two groups should be their treatment status.

We will use `psmatch2` for the propensity score matching to balance the covariates in the treatment and control samples, but we can first compute the propensity scores ourselves using

the fitted values of the logit estimation:[19]

```
. predict fit, xb

. gen pscore=exp(fit)/(1+exp(fit))

. sum treatment pscore

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
   treatment |      9,259    .1961335    .3970921          0          1
      pscore |      9,259    .1961335    .2570519   1.47e-07   .9903858
```

We can see that the `treatment` indicator variable equals 1 for 19.6 percent of the sample, and 0 otherwise. The `pscore` variable we just created has the same mean as `treatment`, but it is a continuous variable that captures the predicted probability of treatment based on the covariates included in the logit estimation from above. When we evaluate these predicted probabilities for the treatment and control groups, we can see they clearly discriminate between observations that are actually treated versus those that are not:

```
. tabstat pscore, by(treatment)

Summary for variables: pscore
Group variable: treatment

treatment |      Mean
----------+----------
        0 |  .1135354
        1 |  .5346675
----------+----------
    Total |  .1961335
--------------------
```

To reduce the differences in covariate values between the treatment and control sample, `psmatch2` computes the propensity score in the same way (assuming we let the program use a logit estimation; the alternative is probit). The following code uses the same logit estimation as presented above and specifies that the outcome variable we examine is `cfop1`. We further specify that we match each treatment observation to its nearest neighbor in terms of the propensity score (`n(1)`), we require `common` support (more on this below), we allow a maximum `caliper` distance in propensity scores between matched pairs of 0.01, and allow each control observation to be matched with only one treatment observation (`noreplace`):

[19] The code here illustrates how to obtain the propensity scores in two steps. Alternatively, you can simply type `predict pscore, pr`.

```
. psmatch2 treatment timedum_* lnta btm lnage salesgr lnsdearn spdum rnd div
↪  cfo acc expsc  inta depr ng_indus, n(1) common caliper(0.01) logit
↪  noreplace outc(cfop1)
note: timedum_36 omitted because of collinearity.

Logistic regression                                     Number of obs =   9,259
                                                        LR chi2(49)   = 3697.36
                                                        Prob > chi2   =  0.0000
Log likelihood = -2734.4838                             Pseudo R2     =  0.4034

------------------------------------------------------------------------------
  treatment | Coefficient  Std. err.      z    P>|z|     [95% conf. interval]
------------+-----------------------------------------------------------------
  timedum_1 |   1.273881   .3507545     3.63   0.000      .586415    1.961347
        ...
       lnta |   .5459653   .0291749    18.71   0.000     .4887836    .6031471
        btm |   .0522964   .0812177     0.64   0.520    -.1068873    .2114801
      lnage |  -.2490797   .0569752    -4.37   0.000     -.360749   -.1374104
    salesgr |   .0606946   .0664283     0.91   0.361    -.0695025    .1908917
   lnsdearn |  -.0438052   .0370056    -1.18   0.237    -.1163348    .0287244
      spdum |   .6638392   .0746219     8.90   0.000     .5175829    .8100954
        rnd |   4.461092   1.532898     2.91   0.004     1.456668    7.465517
        div |  -22.04053   8.673314    -2.54   0.011    -39.03991   -5.041143
        cfo |   20.68008   1.212625    17.05   0.000     18.30338    23.05678
        acc |   7.285523   1.030672     7.07   0.000     5.265443    9.305602
      expsc |   80.25899   6.616544    12.13   0.000     67.29081    93.22718
       inta |   2.177683   .1764341    12.34   0.000     1.831879    2.523488
       depr |  -18.73584   4.174841    -4.49   0.000    -26.91838    -10.5533
   ng_indus |   7.235906   .2597329    27.86   0.000     6.726839    7.744974
      _cons |  -9.171973   .4257103   -21.55   0.000    -10.00635   -8.337596
------------------------------------------------------------------------------
------------------------------------------------------------------------------
Variable   Sample |    Treated     Controls   Difference        S.E.   T-stat
------------------+-----------------------------------------------------------
 cfop1  Unmatched | .082819289  -.090824745   .173644034  .005924133    29.31
             ATT | .077399289   .024039467   .053359822  .005203649    10.25
------------------+-----------------------------------------------------------
Note: S.E. does not take into account that the propensity score is estimated.

 psmatch2: |   psmatch2: Common
 Treatment |        support
assignment | Off suppo  On suppor |     Total
-----------+----------------------+----------
 Untreated |         0      7,443 |     7,443
   Treated |       681      1,135 |     1,816
-----------+----------------------+----------
     Total |       681      8,578 |     9,259
```

The first lines of results are the same as we had with the logit estimation. The next block of results provides us two different tests. First we see a t-test on the difference in means of `cfop1`

for the unmatched treatment and control groups. Second, we see a t-test on the difference in means of `cfop1` for the treatment and control groups after matching these groups on the covariates included in the logit estimation. The results suggest that the difference in means between the matched treatment and control groups equals 0.053 with a t-value of 10.25. This is an estimate of the average treatment effect on the observations that actually receive the treatment (the "ATT"). Importantly, note that this t-value is based on standard errors that do not (yet) account for the effects of potential clustering. In addition, as indicated by the note "S.E. does not take into account that the propensity score is estimated", the standard error might not be correct because it ignores the fact that a matching step was taken before the mean comparison between the matched treatment and control observations.

In the bottom block of results, we further see information on the "common support". The common support refers to the extent to which the predicted probabilities (as captured by the propensity scores) for the treatment observations are within the range of those in the control sample. Here, 681 of the 1816 treatment observations have propensity scores that fall outside of this range. We can see this more clearly by inspecting the propensity scores for the treatment and control observations that remain unmatched:

```
. sum pscore if treatment==1 & _weight==.

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
      pscore |        681    .8000031    .1307566   .4820849   .9893873

. sum pscore if treatment==0 & _weight==.

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
      pscore |      6,308    .0658553    .0866694   1.47e-07   .4721092
```

For the 681 for which no match was found, we can see that their propensity scores do not overlap with the distribution of propensity scores of the remaining observations in the control sample. The lowest propensity score of the unmatched treatment group is 0.482, while the highest propensity score of the unmatched control group is 0.472. Because we match without replacement, the pool of potential control observation matches shrinks every time the program's algorithm finds a match for one of the treatment observations. The matching takes place until there is no more overlap in the distributions of the propensity scores of the treatment and control groups.

When we inspect our dataset after running `psmatch2`, we can also see that several new variables were created by the program. These are:

```
. sum _pscore _treated _support _weight _cfop1 _id _n1 _nn _pdif

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
     _pscore |      9,259    .1961335    .2570519   1.47e-07   .9903858
    _treated |      9,259    .1961335    .3970921          0          1
    _support |      9,259    .9264499    .2610514          0          1
     _weight |      2,270           1           0          1          1
      _cfop1 |      1,135    .0240395    .1484834  -.6990907   .4217812
-------------+---------------------------------------------------------
         _id |      9,259        4630    2672.987          1       9259
         _n1 |      1,135    6337.086    1184.003        819       7443
         _nn |      9,259    .1225834    .3279761          0          1
       _pdif |      1,135    .0030939    .0040083   1.11e-09   .0099939
```

Of these variables, `_pscore` is exactly the same propensity score as we calculated ourselves after the logit estimation, `_treated` is equal to the `treatment` variable, and `_support` is an indicator variable that is equal to 1 for observations in the control group or treatment observations with common support. Variable `_weight` captures the number of times an observation is used for a match. For treatment observations this value is always 1, but for control observations this can be greater than 1 if we would match with replacement. Because we match without replacement replacement here, all values equal 1.

Variable `_cfop1` is a variable that, for each treatment observation, contains the value of the outcome variable of the matched control observation. Variable `_id` is a variable that uniquely identifies each observation in the sample. This variable is useful because it links to variable `_n1`, which contains the identifier of the nearest-neighbor match for a treatment observation. The variable `_nn` contains the number of observations matched to a treatment observation, which is 1 here given the choice to match only to the first nearest neighbor. Finally, for each matched treatment observation, variable `_pdif` contains the difference in propensity scores between matched treatment and control observations. For example, the highest value of 0.0099939 (which is below the caliper distance of 0.01) belongs to the treatment observation with `_id==9103`, which was matched to the control observation with `_id==7434`. The propensity scores for these observations are 0.91661393 and 0.92660782, respectively, which translates into the difference of 0.0099939.

It is important that we not only examine the differences in the means of the outcome variable between the matched treatment and control observations, but also that we assess the success of

the matching procedure in balancing the treatment and control groups in terms of the observed covariates. One simple solution is to redo the logit estimation for the treatment indicator, but now only for the matched treatment and control observations. In the output shown below, we can see that none of the covariates is significantly related to the treatment status after matching:

```
. gen treat_psm=0 if _weight!=. & treatment==0
(8,124 missing values generated)

. replace treat_psm=1 if _weight!=. & treatment==1
(1,135 real changes made)

. logit treat_psm timedum_* lnta btm lnage salesgr lnsdearn spdum rnd div cfo
↪  acc expsc inta depr ng_indus

note: timedum_36 omitted because of collinearity.
Iteration 0:   log likelihood = -1573.4441
Iteration 1:   log likelihood = -1566.2955
Iteration 2:   log likelihood =  -1566.295
Iteration 3:   log likelihood =  -1566.295

Logistic regression                                     Number of obs =  2,270
                                                        LR chi2(49)   =  14.30
                                                        Prob > chi2   = 1.0000
Log likelihood = -1566.295                              Pseudo R2     = 0.0045

------------------------------------------------------------------------------
  treat_psm | Coefficient  Std. err.      z    P>|z|     [95% conf. interval]
------------+-----------------------------------------------------------------
  timedum_1 |   .0000751    .429216     0.00   1.000    -.8411728     .841323
        ...
       lnta |   .0051809    .035437     0.15   0.884    -.0642744    .0746362
        btm |  -.0226109   .1021104    -0.22   0.825    -.2227435    .1775218
      lnage |   .0514345   .0691098     0.74   0.457    -.0840183    .1868873
    salesgr |   .0589519   .0844805     0.70   0.485    -.1066268    .2245305
   lnsdearn |   .0424737   .0439864     0.97   0.334     -.043738    .1286854
      spdum |  -.0354987   .0931525    -0.38   0.703    -.2180742    .1470768
        rnd |   1.209932   2.030382     0.60   0.551    -2.769543    5.189407
        div |  -7.231397   9.833041    -0.74   0.462     -26.5038    12.04101
        cfo |  -1.367198   1.533419    -0.89   0.373    -4.372644    1.638249
        acc |  -.3149529   1.212049    -0.26   0.795    -2.690526     2.06062
      expsc |   -5.45401   8.649604    -0.63   0.528    -22.40692     11.4989
       inta |   .0402169    .213122     0.19   0.850    -.3774945    .4579283
       depr |    1.95448   5.189355     0.38   0.706    -8.216468    12.12543
   ng_indus |  -.0606578     .31979    -0.19   0.850    -.6874347    .5661191
      _cons |   .1093349   .5219786     0.21   0.834    -.9137243    1.132394
------------------------------------------------------------------------------
```

We can also test the differences in means for each variable individually by regressing the variable on the matched treatment indicator. This also allows us to adjust the standard errors

for potential clustering, as follows:

```
. reghdfe lnta treat_psm, cluster(cik qid) noab
(MWFE estimator converged in 1 iterations)

HDFE Linear regression                            Number of obs   =      2,270
Absorbing 1 HDFE group                            F(   1,     35) =       0.00
Statistics robust to heteroskedasticity           Prob > F        =     0.9832
                                                  R-squared       =     0.0000
                                                  Adj R-squared   =    -0.0004
Number of clusters (cik)     =        932         Within R-sq.    =     0.0000
Number of clusters (qid)     =         36         Root MSE        =     1.6705

(Std. err. adjusted for 36 clusters in cik qid)
------------------------------------------------------------------------------
             |               Robust
        lnta | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
  treat_psm |   .0031065   .1464974     0.02   0.983     -.294299     .300512
      _cons |   6.367659   .1198772    53.12   0.000     6.124296    6.611023
------------------------------------------------------------------------------
```

Also useful is the post-matching command `pstest`, which provides a table of differences in means and variances after matching. Finally, to perform our actual tests of differences in the outcome variables between the matched treatment and control observations, we can estimate a simple linear regression of `cfop1` on the `treat_psm` variable. Without adjusting the standard errors, we get a coefficient of 0.0533598 with a t-value of 10.25, which is exactly the same as for the ATT shown by `psmatch2`. Alternatively, we can use the regression to adjust the standard errors and/or to add additional control variables in the mode. In the following output, we can see the difference in average future cash flows between treated and control observations reported in Table 8 of the paper (for "PSM 2") with standard errors that are clustered by firm and time:[20]

```
. reghdfe cfop1 treat_psm, cluster(cik qid) noab
(MWFE estimator converged in 1 iterations)

HDFE Linear regression                            Number of obs   =      2,270
Absorbing 1 HDFE group                            F(   1,     35) =      41.19
Statistics robust to heteroskedasticity           Prob > F        =     0.0000
                                                  R-squared       =     0.0443
                                                  Adj R-squared   =     0.0439
Number of clusters (cik)     =        932         Within R-sq.    =     0.0443
```

[20] The only difference with the results in Table 8 of Leung and Veenman [2018] lies in the standard errors and t-values. This is because here we use `reghdfe`, while in Leung and Veenman [2018] we used `cluster2` from https://www.kellogg.northwestern.edu/faculty/petersen/htm/papers/se/se_programming.htm. See Section 4.5.1 for an explanation of the difference induced by these programs.

```
Number of clusters (qid)     =         36         Root MSE        =     0.1240

(Std. err. adjusted for 36 clusters in cik qid)
------------------------------------------------------------------------------
             |               Robust
       cfop1 | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
   treat_psm |   .0533598   .0083146     6.42   0.000     .0364803    .0702394
       _cons |   .0240395   .0067449     3.56   0.001     .0103467    .0377322
------------------------------------------------------------------------------
```

Similarly, we can estimate this regression with all of the matching covariates included as well. As explained in Section 5.2 of Leung and Veenman [2018], this form of "doubly-robust" estimation helps to eliminate any residual (random) differences in covariates across the treated and control observations. The idea is that the matching is never perfect (i.e., `_pdif>0` for all observations). Below is part of the regression output (coefficient estimates for the control variables are excluded):[21]

```
. reghdfe cfop1 timedum_* indusdum_* cfo acc_gaap lnta btm lnage salesgr lnsd
>earn spdum rnd div expsc inta depr ng_indus treat_psm, cluster(cik qid) noab
(MWFE estimator converged in 1 iterations)

HDFE Linear regression                            Number of obs   =      2,270
Absorbing 1 HDFE group                            F(  98,     35) =          .
Statistics robust to heteroskedasticity           Prob > F        =          .
                                                  R-squared       =     0.3656
                                                  Adj R-squared   =     0.3369
Number of clusters (cik)     =        932         Within R-sq.    =     0.3656
Number of clusters (qid)     =         36         Root MSE        =     0.1032

(Std. err. adjusted for 36 clusters in cik qid)
------------------------------------------------------------------------------
             |               Robust
       cfop1 | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
   timedum_1 |   .0849939   .0136689     6.22   0.000     .0572446    .1127432
         ...
         cfo |   .9514314   .1520771     6.26   0.000     .6426984    1.260164
    acc_gaap |   .4985704   .0964427     5.17   0.000     .3027814    .6943595
        lnta |   .0087056   .0026818     3.25   0.003     .0032613    .0141499
         btm |  -.0083251   .0070997    -1.17   0.249    -.0227384    .0060881
       lnage |   .0059662   .0048171     1.24   0.224     -.003813    .0157454
     salesgr |  -.0197993   .0061151    -3.24   0.003    -.0322136    -.007385
    lnsdearn |  -.0066653   .0028166    -2.37   0.024    -.0123834   -.0009473
```

[21] Note that with this estimation, we still do not account for the potential estimation uncertainty that is induced by the matching step that was performed before this regression estimation. Abadie and Spiess [2022], however, show that the combination of doubly-robust estimation, as done here, when matching without replacement and with cluster-robust standard errors, leads to consistent standard error estimates when we assume the regression model is correctly specified.

```
       spdum |   .0067357   .0048564     1.39   0.174    -.0031234    .0165947
         rnd |  -1.071397   .2657889    -4.03   0.000    -1.610978   -.5318173
         div |   2.405209   .4097511     5.87   0.000      1.57337    3.237048
       expsc |   1.990613   .9415434     2.11   0.042     .0791783    3.902048
        inta |    .009229   .0150343     0.61   0.543    -.0212922    .0397503
        depr |   3.264426   .3467343     9.41   0.000     2.560518    3.968334
    ng_indus |    .046966    .030421     1.54   0.132    -.0147919     .108724
   treat_psm |   .0564843   .0066518     8.49   0.000     .0429805    .0699881
       _cons |  -.1523569   .0234643    -6.49   0.000     -.199992   -.1047218
------------------------------------------------------------------------------
```

[psmatch2]

4.8.2 Matching with replacement

The example presented above focused on matching without replacement. It can also be useful, however, to match with replacement. Matching with replacement simply means that once a control observation is used in a match, it is "thrown back" into the potential pool of control observations for potential matching with another treatment observation. This could be useful when there is little common support and/or the pool of control observations is smaller, or not much larger, than the treatment sample. A classic example is the matching of Big-4 clients to non-Big-4 clients in studies of the effects of auditor size on outcome variables such as audit fees or discretionary accruals (e.g., Lawrence et al. [2011]; DeFond et al. [2016]).

Consider the question of whether firms pay a fee premium for hiring a Big-4 auditor instead of a smaller audit firm, and what the magnitude of the fee premium is (if any). In a sample of US firms I obtained for the years 2002–2018, we can see that the pool of treatment observations (i.e., Big-4 clients) is substantially larger than the control sample:

```
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
        big4 |     36,115     .777627    .4158462         0          1
```

That is, about 78 percent of firm-year observations in this sample are treatment observations, leaving only 22 percent of observations available as potential matches in the control sample. This means that even when we do not impose restrictions such as common support or a caliper distance, the largest part of treatment sample will not be matched if we decide to match with replacement. In this example, an additional concern is that there is very limited overlap in the distributions of propensity scores for the treatment and control samples (see Shipman et al. [2017, Figure 1-A]). So this leaves even fewer control firm-observations as suitable matches.

In the example below, we first match without replacement using a simple set of firm characteristics. We can see that 8,031 firm-years in the Big-4 (treatment) sample were matched to an observation in the control sample, much less than the total number of 28,137 firm-years in the treatment sample. This is because the control sample consists of only 8,031 observations. Each of these observations is thus used in a match with a treatment observation.

```
. psmatch2 big4 lnassets aturn curr lev roa salesgr, n(1) logit noreplace
>outc(lnauditfees)

Logistic regression                                     Number of obs =   36,115
                                                        LR chi2(6)    = 11312.03
                                                        Prob > chi2   =   0.0000
Log likelihood = -13481.142                             Pseudo R2     =   0.2956

------------------------------------------------------------------------------
        big4 | Coefficient  Std. err.      z    P>|z|     [95% conf. interval]
-------------+----------------------------------------------------------------
    lnassets |   .9962023   .0126367    78.83   0.000     .9714347     1.02097
       aturn |    .156676    .021578     7.26   0.000     .1143839    .1989681
        curr |   .0536695   .0084254     6.37   0.000     .0371561     .070183
         lev |  -.7975957   .0782258   -10.20   0.000    -.9509154    -.644276
         roa |  -1.601652   .0914624   -17.51   0.000    -1.780915   -1.422389
 salesgrowth |  -.1897099   .0401083    -4.73   0.000    -.2683207   -.1110992
       _cons |  -4.454175   .0949572   -46.91   0.000    -4.640288   -4.268062
------------------------------------------------------------------------------
------------------------------------------------------------------------------
Variable     Sample |   Treated    Controls   Difference         S.E.   T-stat
-----------------+------------------------------------------------------------
lnaud...  Unmatched | 14.119882   12.659356   1.46052659   .013189094   110.74
               ATT | 13.094030   12.659356   .434674244   .014031483    30.98
-----------------+------------------------------------------------------------
Note: S.E. does not take into account that the propensity score is estimated.

 psmatch2: |   psmatch2: Common
 Treatment |        support
assignment | Off suppo  On suppor |     Total
-----------+----------------------+----------
 Untreated |         0      8,031 |     8,031
   Treated |    20,053      8,031 |    28,084
-----------+----------------------+----------
     Total |    20,053     16,062 |    36,115
```

Importantly, further inspection suggests that the matching is extremely poor. For example, we can see that although the pre-matching difference in propensity scores between the treatment and control groups is reduced after the matching, the average propensity scores of 62.3 and 53.1 percent for the matched treatment and control observations are still far apart:

```
. // Propensity scores before matching:
```

```
. tabstat _pscore, by(big4)

Summary for variables: _pscore
Group variable: big4

    big4 |      Mean
---------+----------
       0 |    .53097
       1 |  .8481619
---------+----------
   Total |   .777627
--------------------

. // Propensity scores after matching:
. tabstat _pscore if _weight!=., by(big4)

Summary for variables: _pscore
Group variable: big4

    big4 |      Mean
---------+----------
       0 |    .53097
       1 |  .6232409
---------+----------
   Total |  .5771055
```

Another important issue we observe when comparing the mean propensity scores of the treatment group before and after matching, is that the mean propensity score of the treatment observations drops from 84.8 to 62.3 percent after matching. This means that out of all the treatment observations in the original sample, the matched sample contains only those observations that had a relatively low conditional probability of hiring a Big-4 auditor. Given the importance of client size in explaining the likelihood of having a Big-4 auditor (see the logit results of `psmatch2` above), this means that the matched treatment firms are much smaller than the unmatched treatment firms (variable `lnassets` is the natural logarithm of firms' total assets, a measure of firm size):

```
. gen big4_matched=0 if big4==1
(8,031 missing values generated)

. replace big4_matched=1 if big4==1 & _weight!=.
(8,031 real changes made)

. tabstat lnassets, by(big4_matched)

Summary for variables: lnassets
Group variable: big4_matched
```

```
big4_matched |      Mean
-------------+----------
           0 |   7.84284
           1 |  4.995169
-------------+----------
       Total |   7.02851
-------------------------
```

The problem of the remaining difference in average propensity scores after matching can be addressed by imposing the caliper restriction (i.e., a maximum distance in propensity scores between matched pairs). Doing so helps balance the average propensity scores and remove the significant differences in firm characteristics between the matched treatment and control samples, but this also further reduces the number of treatment observations that are matched to 6,011 (see the number of observation in the cell "Treated / On suppor[t]"):[22]

```
. psmatch2 big4 lnassets aturn curr lev roa salesgr, n(1) common caliper(0.01)
>logit noreplace outc(lnauditfees)

...
------------------------------------------------------------------------------
Variable     Sample |   Treated    Controls   Difference         S.E.   T-stat
--------------------+---------------------------------------------------------
lnaud...  Unmatched | 14.119882   12.659356   1.46052659   .013189094   110.74
                ATT | 13.106878   12.898236   .208642045   .016916551    12.33
------------------+---------------------------------------------------------
Note: S.E. does not take into account that the propensity score is estimated.

 psmatch2: |   psmatch2: Common
 Treatment |        support
assignment | Off suppo  On suppor |     Total
-----------+----------------------+----------
 Untreated |         0      8,031 |     8,031
   Treated |    22,073      6,011 |    28,084
-----------+----------------------+----------
     Total |    22,073     14,042 |    36,115

.     tabstat _pscore if _weight!=., by(big4)

Summary for variables: _pscore
Group variable: big4

    big4 |      Mean
---------+----------
       0 |  .6192445
       1 |  .6130136
```

[22] Note that the choice here to impose a caliper distance of 0.01 is a bit arbitrary and should not be followed automatically in an actual study without careful thought. See Roberts and Whited [2012] and Shipman et al. [2017] for more guidance on the choice of caliper distance.

```
---------+----------
   Total |  .6161291
-------------------
```

What happens when we match with replacement by removing the `noreplace` option? We now see that 27,902 of the treatment observations get matched, with only 182 remaining unmatched:

```
. psmatch2 big4 lnassets aturn curr lev roa salesgr, n(1) common caliper(0.01)
>logit outc(lnauditfees)

...
------------------------------------------------------------------------------
Variable     Sample |   Treated    Controls   Difference         S.E.   T-stat
-------------------+----------------------------------------------------------
lnaud...  Unmatched | 14.119882   12.659356   1.46052659   .013189094   110.74
                ATT |  14.10841   13.726584   .381833746    .03721675    10.26
---------------------------+--------------------------------------------------
Note: S.E. does not take into account that the propensity score is estimated.

 psmatch2: |   psmatch2: Common
 Treatment |        support
assignment | Off suppo  On suppor |     Total
-----------+----------------------+----------
 Untreated |         0      8,031 |     8,031
   Treated |       182     27,902 |    28,084
-----------+----------------------+----------
     Total |       182     35,933 |    36,115
```

We can see that the matching with replacement produces only a very small difference in average propensity scores between the matched treatment and control samples. Importantly, we now also see that the mean propensity score of the treatment observations is now much higher than it was when matching without replacement:

```
. tabstat _pscore [aw=_weight], by(big4)

Summary for variables: _pscore
Group variable: big4

    big4 |      Mean
---------+----------
       0 |  .8471856
       1 |  .8471862
---------+----------
   Total |  .8471859
-------------------
```

In tabulating these means using `tabstat`, the weights in variable `_weights` were used to obtain weighted means. This is because the `_weights` variable contains information about how

often a particular control observation is used as a match with a treatment observation. Recall that for treatment observations the weight is always equal to 1, and that it is equal to 1 for all observations when matching without replacement. After matching with replacement, we see a very different pattern:

```
. sum _weight if big4==0, d

psmatch2: weight of matched controls
-------------------------------------------------------------
      Percentiles      Smallest
 1%            1              1
 5%            1              1
10%            1              1       Obs                4,672
25%            1              1       Sum of wgt.        4,672

50%            2                      Mean           5.972175
                      Largest         Std. dev.      14.92343
75%            5            203
90%           12            243       Variance       222.7087
95%           22            314       Skewness       12.55932
99%           60            464       Kurtosis       272.3791
```

The 27,902 treatment observations were matched to (only!) 4,672 control observations. Also, we observe huge variation in the weights. While a bit more than half of these control observations are used in a match only once or twice, several observations receive weights that are extremely large. One control observation is used even 464 times in a match with a treatment observation. Despite the benefits of matching with replacement, this extreme weighting is a clear drawback of matching with replacement because it makes the inferences much more dependent on a single observation (this issue is similar to that for entropy balancing, discussed in Section 4.9, see McMullin and Schonberger [2022]).

Thus far we have not interpreted the results with respect to the outcome variable `lnauditfee` yet. The `psmatch2` results without replacement suggested a difference of 0.2086 with a t-value of 12.33. This translates into a fee premium of more than 23 percent for hiring a Big-4 auditor ($e^{0.2086} - 1 = 0.2320$). The difference is much higher at 0.3818 with a t-value of 10.26 when matching with replacement, which highlights that the inferences here are very sensitive to the implementation of the matching procedure. There appears to be a clear "Big-4 premium", but its magnitude is highly uncertain given the sensitivity to different design choices. Given the focus here is on the implementation of `psmatch2` in Stata, the remaining discussion will refrain from actually attempting to answer the research question.

As before, we can also combine the matching with regression and compute cluster-robust standard errors. For comparison purposes, below are the results after matching without and with replacement, respectively. The following regression results were obtained after matching without replacement:

```
. reg lnauditfee big4 lnassets aturn curr lev roa salesgr if _weight!=.,
>cluster(gvkey)

Linear regression                               Number of obs     =     12,022
                                                F(7, 3507)        =     651.05
                                                Prob > F          =     0.0000
                                                R-squared         =     0.5259
                                                Root MSE          =     .64279

(Std. err. adjusted for 3,508 clusters in gvkey)
------------------------------------------------------------------------------
             |               Robust
 lnauditfees | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
        big4 |   .2397458   .0203788    11.76   0.000     .1997903    .2797013
    lnassets |   .5667943   .0095985    59.05   0.000     .5479751    .5856135
       aturn |   .1197562   .0144379     8.29   0.000     .0914486    .1480637
        curr |  -.0099505   .0049328    -2.02   0.044    -.0196219   -.0002791
         lev |  -.3848752   .0470814    -8.17   0.000     -.477185   -.2925654
         roa |  -1.012373   .0479707   -21.10   0.000    -1.106426   -.9183193
 salesgrowth |  -.0421716    .015676    -2.69   0.007    -.0729065   -.0114367
       _cons |   9.965836   .0591581   168.46   0.000     9.849849    10.08182
------------------------------------------------------------------------------
```

The following regression results were obtained after matching with replacement:

```
. reg lnauditfee big4 lnassets aturn curr lev roa salesgr [aw=_weight],
>cluster(gvkey)
(sum of wgt is 55,804)

Linear regression                               Number of obs     =     32,574
                                                F(7, 5001)        =     460.90
                                                Prob > F          =     0.0000
                                                R-squared         =     0.5419
                                                Root MSE          =     .70625

(Std. err. adjusted for 5,002 clusters in gvkey)
------------------------------------------------------------------------------
             |               Robust
 lnauditfees | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
        big4 |   .3693434   .0622546     5.93   0.000     .2472971    .4913898
    lnassets |   .4771622   .0270295    17.65   0.000     .4241725    .5301519
       aturn |   .1399533   .0376796     3.71   0.000     .0660849    .2138218
        curr |   .0078425   .0106401     0.74   0.461    -.0130167    .0287017
```

```
         lev |  -.1971171   .1258805    -1.57   0.117    -.4438981    .0496639
         roa |  -.7540524   .0969409    -7.78   0.000    -.9440991   -.5640057
 salesgrowth |  -.2346081   .0890705    -2.63   0.008    -.4092254   -.0599908
       _cons |   10.54304   .2020616    52.18   0.000     10.14691    10.93917
------------------------------------------------------------------------------
```

We can see for these two estimations that after controlling for other covariates, the estimates of the Big-4 premiums are 0.240 and 0.369, respectively. Interestingly, we also see that, despite the substantially larger sample when matching with replacement, the standard error (t-value) is larger (smaller) for the `big4` coefficient in the second estimation (after matching with replacement) than in the first estimation (after matching without replacement). This is an outcome of the duplication of information due to the reuse of control observations in the matching with replacement. This duplication, or in other words, the fact that some control observations have extreme weights in determining the results, leads to a substantial reduction in the precision (an increase in the variance) of the estimated treatment effect.

In sum, this section illustrates how `psmatch2` can be used for the purpose of propensity score matching. Keep in mind, however, that the program should be used with considerable caution. In many settings, the results from propensity score matching can be highly sensitive to the many design choices available.

`[psmatch2]`

4.9 Entropy balancing

An alternative to propensity score matching that is frequently used in the literature is entropy balancing. This section illustrates entropy balancing in the Big-4 audit premium setting from Section 4.8 using the program `ebalance` (available via `ssc inst ebalance`). To understand how entropy balancing works, see for example Hainmueller [2012] and McMullin and Schonberger [2022].

An important advantage of entropy balancing over propensity-score matching is that it has fewer parameters to choose from, so inferences will be less dependent on researcher discretion. In addition, the idea of entropy balancing is to essentially eliminate *all* differences in covariates between the treatment and control observations, while with propensity-score matching random differences will remain. Similar to matching with replacement, control observations receive weights that can differ from 1 (all treatment observations receive weights of 1). Different from matching with replacement is that the weights of the control observations are not integers and

they can also be between 0 and 1. Entropy balancing is a procedure that optimizes weights such that an exact balance is achieved between the treatment and control sample in terms of the covariates specified by the researcher. Consider the following example for the same dataset as in Section 4.8:

```
. ebalance big4 lnassets aturn curr lev roa salesgr, targets(1)

Data Setup
Treatment variable:   big4
Covariate adjustment: lnassets aturn curr lev roa salesgrowth

Optimizing...
Iteration 1: Max Difference = 45530.5589
...
Iteration 14: Max Difference = .000991439
maximum difference smaller than the tolerance level; convergence achieved

Treated units: 28084   total of weights: 28084
Control units: 8031    total of weights: 28084

Before: without weighting

           |              Treat               |             Control
           |      mean   variance   skewness |      mean   variance   skewness
-----------+----------------------------------+----------------------------------
  lnassets |     7.029      2.898     -.1436 |     4.668      2.095      .5108
     aturn |     1.107      .5965      1.396 |     1.219      .6632      1.133
      curr |     2.237      2.714      2.824 |     2.506      4.207      2.643
       lev |     .2814     .04153      .7513 |     .2303     .04086      1.056
       roa |   .003831     .02196     -2.835 |   -.05669     .04157     -1.998
salesgrowth|     1.108     .09521      3.796 |     1.136      .1744      3.437

After:  _webal as the weighting variable

           |              Treat               |             Control
           |      mean   variance   skewness |      mean   variance   skewness
-----------+----------------------------------+----------------------------------
  lnassets |     7.029      2.898     -.1436 |     7.028      2.875    -.05005
     aturn |     1.107      .5965      1.396 |     1.107      .6706      1.189
      curr |     2.237      2.714      2.824 |     2.237      3.298      2.849
       lev |     .2814     .04153      .7513 |     .2814     .03968      .5958
       roa |   .003831     .02196     -2.835 |   .003821     .01478     -3.051
salesgrowth|     1.108     .09521      3.796 |     1.108      .1146      2.032
```

The output shows us that after the entropy balancing procedure is applied, the means of the treatment and (weighted) control groups are virtually identical. We only see a very minor difference (7.029 vs. 7.028) for the `lnassets` variable. Similar to matching with replacement, we can now use the weights produced by the entropy balancing procedure (stored in variable

_webal) to examine the differences in averages of the outcome variable for the treatment and control group after balancing on the observable covariates:

```
. reg lnauditfee big4 lnassets aturn curr lev roa salesgr [aw=_webal],
>cluster(gvkey)
(sum of wgt is 56,168)

Linear regression                               Number of obs     =     36,115
                                                F(7, 5221)        =     385.92
                                                Prob > F          =     0.0000
                                                R-squared         =     0.5415
                                                Root MSE          =     .71017

(Std. err. adjusted for 5,222 clusters in gvkey)
------------------------------------------------------------------------------
             |               Robust
 lnauditfees | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
        big4 |    .367632   .0642078     5.73   0.000     .2417578    .4935062
    lnassets |   .4758699   .0269708    17.64   0.000     .4229958    .5287439
       aturn |   .1396813   .0364761     3.83   0.000     .0681728    .2111897
        curr |    .008929   .0105429     0.85   0.397    -.0117396    .0295975
         lev |  -.2912367   .1204249    -2.42   0.016    -.5273199   -.0551536
         roa |  -.7417336   .0956106    -7.76   0.000    -.9291704   -.5542968
 salesgrowth |  -.1013017   .0575894    -1.76   0.079     -.214201    .0115976
       _cons |   10.43007   .1699488    61.37   0.000      10.0969    10.76324
------------------------------------------------------------------------------
```

These results are very close, in terms of both coefficients and t-values, to those obtained using matching with replacement in Section 4.8. This is not surprising, because there a strong overlap in the weights computed in the propensity-score matching and entropy-balancing procedures. Setting the weights of unmatched control observations in the propensity-score matching procedure to zero (analogously, entropy balancing sets these weights to values very close to zero), we can see a high correlation between the weights:

```
. pwcorr _webal _weight if big4==0

             |   _webal  _weight
-------------+------------------
      _webal |   1.0000
     _weight |   0.7245   1.0000
```

Different from propensity score matching, entropy balancing allows us to balance distributions of the treatment and control variables on more than just the first moment (i.e., the means). It also allows us to balance the variance (second moment) and even skewness (third moment) of the distributions. In the first example above, we only balanced the first moment by specifying

the option `targets(1)`. But we can also specify `targets(2)` and `targets(3)` to additionally balance on the second and third moments, respectively. For example, using `targets(3)` we get an almost perfect overlap in the distributions in terms of means, standard deviations, and skewness:

```
. ebalance big4 lnassets aturn curr lev roa salesgr, targets(3)

Data Setup
Treatment variable:   big4
Covariate adjustment: lnassets aturn curr lev roa salesgrowth (1st order).
lnassets aturn curr lev roa salesgrowth (2nd order).
lnassets aturn curr lev roa salesgrowth (3rd order).


Optimizing...
Iteration 1: Max Difference = 212468.448
...
Iteration 15: Max Difference = .000423625
maximum difference smaller than the tolerance level; convergence achieved


Treated units: 28084   total of weights: 28084
Control units: 8031    total of weights: 28084


Before: without weighting
...

After:  _webal as the weighting variable

           |               Treat               |             Control
           |      mean   variance   skewness |      mean   variance   skewness
-----------+---------------------------------+---------------------------------
  lnassets |     7.029      2.898     -.1436 |     7.028      2.898     -.1436
     aturn |     1.107      .5965      1.396 |     1.107      .5965      1.396
      curr |     2.237      2.714      2.824 |     2.237      2.714      2.824
       lev |     .2814     .04153      .7513 |     .2814     .04153      .7513
       roa |   .003831     .02196     -2.835 |   .003829     .02196     -2.835
salesgrowth|     1.108     .09521      3.796 |     1.108     .09521      3.796
```

The regression results using the weight variable indicate that the fee premium drops a little (from 0.368 to 0.336) when additionally balancing on the higher moments, which highlights the potential importance of taking these higher moments into consideration:

```
. reg lnauditfee big4 lnassets aturn curr lev roa salesgr [aw=_webal],
>cluster(gvkey)
(sum of wgt is 56,168)

Linear regression                               Number of obs     =     36,115
```

```
                                                F(7, 5221)        =     334.65
                                                Prob > F          =     0.0000
                                                R-squared         =     0.5340
                                                Root MSE          =     .71822

(Std. err. adjusted for 5,222 clusters in gvkey)
------------------------------------------------------------------------------
             |               Robust
 lnauditfees | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
        big4 |   .3355869   .0674788     4.97   0.000     .2033001    .4678736
    lnassets |   .4755423   .0292884    16.24   0.000     .4181247    .5329599
       aturn |   .1222458   .0396551     3.08   0.002     .0445052    .1999864
        curr |   .0091996   .0109847     0.84   0.402     -.012335    .0307343
         lev |   -.307755   .1316488    -2.34   0.019    -.5658417   -.0496684
         roa |  -.7425553   .1141851    -6.50   0.000     -.966406   -.5187047
 salesgrowth |  -.0959353   .0380029    -2.52   0.012    -.1704369   -.0214338
       _cons |   10.48181   .1666737    62.89   0.000     10.15506    10.80856
------------------------------------------------------------------------------
```

Similar to the discussion on matching with replacement in Section 4.8, it is important to assess the impact of the weights produced by the entropy balancing procedure. Here, we similarly see that some observations receive extreme weights, which is consistent with the concerns expressed by McMullin and Schonberger [2022]. The weights obtained from balancing on the first three moments have the following distribution:

```
. sum _webal if big4==0, d

entropy balancing weights
-------------------------------------------------------------
      Percentiles      Smallest
 1%     .0740297       .0068324
 5%     .1628202       .0125047
10%     .2329358       .0139524       Obs                8,031
25%     .4426855       .0142769       Sum of wgt.        8,031

50%     1.002785                      Mean           3.496949
                       Largest        Std. dev.      10.87473
75%     2.651635       139.3798
90%     7.129325       172.8756       Variance       118.2598
95%     13.67728       188.3195       Skewness       19.37855
99%     42.89001        536.318       Kurtosis       764.8857
```

All observations receive nonzero weights, but many receive weights close to zero. Others receive extremely high weights, with one control observation receiving a weight as high as 536.318! This is similar to matching a control-group observation to 536 different treatment observations. This result suggests that inferences become more dependent on a few specific

observations. Please see McMullin and Schonberger [2022] for more discussion on this issue and for potential ways to address it.

Overall, entropy balancing is an alternative to propensity-score matching that achieve a (much) more exact balance in terms of covariates between a treatment and control sample, it allows for balancing on higher moments as well, and it has fewer research degrees of freedom (which improves reproducibility of research). At the same time, it is important to keep in mind that entropy balancing has many parallels to matching with replacement. The major difference lies in the mechanics of how the weights are computed. With both approaches, control observations can receive extremely high weights, which makes inferences more sensitive to specific observations and increases the variance (i.e., decreases the precision) of the estimated treatment effect.

Lastly, coming back to the opening discussion of Section 4.8, keep in mind that the main benefit of using entropy balancing (or matching) is that it relaxes the assumption of a linear functional form. It does not "solve endogeneity" or "address selection bias" when the treatment of interest is determined by factors that are not observed by us as researchers.

`[ebalance]`

4.10 Exporting results

One way we can export statistics and regression results to other programs, such as Word or Excel, is to simply copy the output from Stata's **Output/results window**. As explained with the `tabstat` command in Section 4.1, however, it sometimes helps to right-mouseclick the selected output and choose `Copy Table` to have the data structured in a way that Word or Excel recognizes as a table (and therefore separates the statistics into different cells). However, this option does not work well in all situations. For instance, it is difficult to directly copy and paste regression output into a Word or Excel file, even in table format. The good news is that there are several programs that help in exporting statistics and regression output into well-structured tables. Have a look at the programs `tabout`, `outreg2`, and `estout` and their associated help files. All programs are available via the SSC.

Consider for example `outreg2`, which helps create text files that are easily loaded in Excel and can then be used to produce nicely formatted tables. The following code underlies part of the regression results reported in Table 6 (Panel B) of Schafhäutle and Veenman [2023]:

```
reghdfe bhar01 post qes_ibes qss_ibes qsize qbtm qbharpre_* qlnn,
↪  cluster(gvkey qid) absorb(gvkey qid) keepsin
outreg2 using "$path\Tables\did_earet.xls", replace stats(coef tstat)
↪  asterisk(tstat) tdec(2) bdec(3) adjr2 par

reghdfe bhar_miss post qes_ibes qss_ibes qsize qbtm qbharpre_* qlnn,
↪  cluster(gvkey qid) absorb(gvkey qid) keepsin
outreg2 using "$path\Tables\did_earet.xls", append stats(coef tstat)
↪  asterisk(tstat) tdec(2) bdec(3) adjr2 par
```

The `outreg2` command is very flexible and can help you export any part of the regression output. For example, we can add regression columns by using the `append` option, we can let Stata export the t-statistics using parentheses instead of brackets using the `par` option, we can export the adjusted R^2 instead of the normal R^2 using `adjr2`, we can indicate that the stars for significance levels should be placed with the t-statistic instead of the coefficient using `asterisk(tstat)`, and we can specify the number of decimals to be included for the coefficients and t-statistics. We can additionally tell Stata to ignore some of the variables in exporting the results, using the `drop()` option.

The different programs available to export results have many more options that can be tailored to your needs. For example, you can also use these programs to directly export the results to LaTeX files. See the programs' help files for more details and examples.

`[help tabout] [help outreg2] [help estout]`

4.11 Creating graphs

Stata also offers many options to create and export graphs. In my recent papers, I always use Stata to create these graphs and export them to pdf-files, which can then be loaded into a LaTeX editor used to write the paper. Because the creation and formatting of these graphs is always very specific to the research setting, it is impossible to provide an exhaustive overview of how to create publication-ready graphs in Stata. Instead, I provide a few examples of code used to create such graphs below. All graphs are created using `graph twoway` (see `help graph twoway`), which can be abbreviated with the command `twoway`.

First consider Figure 2 of Veenman and Verwijmeren [2022, p.3144], which I reproduce in Figure 4.7 below. This figure contains two overlapping frequency distributions, based on kernel density plots, with different colors and formatting, and nicely labeled axes and a legend.[23]

[23] Note that the appearance of the graph might look slightly different in older versions of Stata (I used Stata

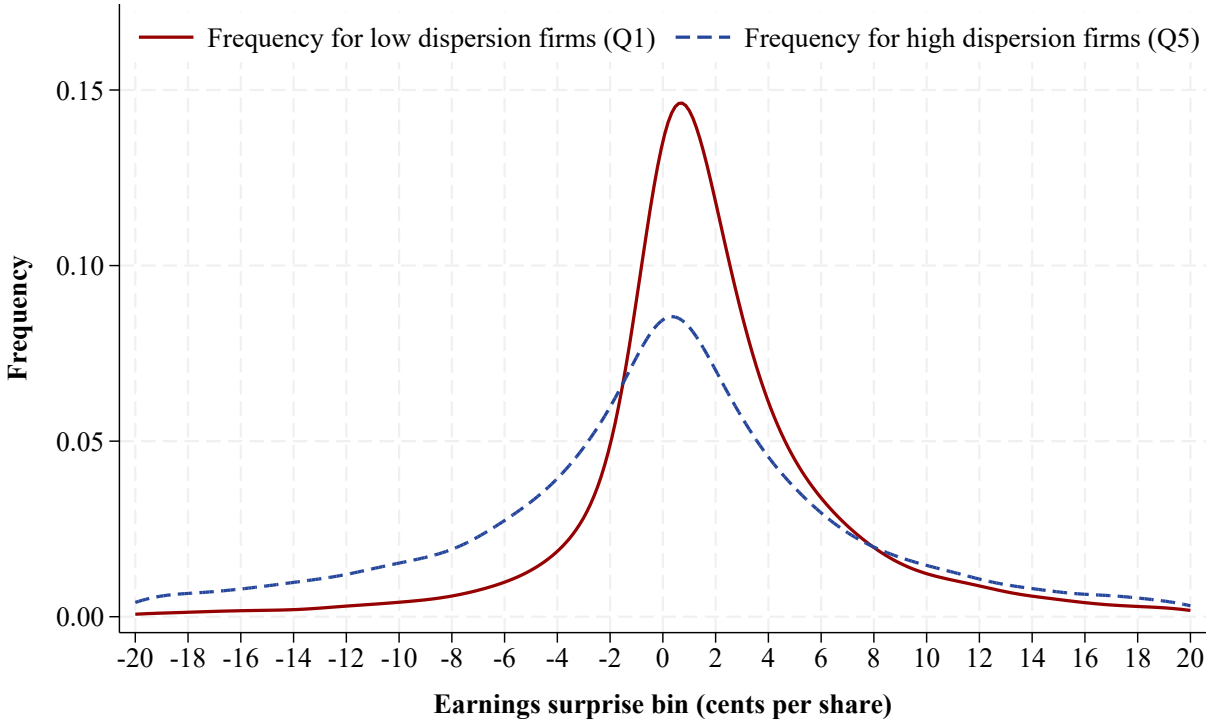


Figure 4.7: *Replication of Figure 2 from Veenman and Verwijmeren [2022]*

The code underlying this figure is shown below. Because this is not the default setting in Stata, I tell Stata to format the text in font Times New Roman. In addition, I tell Stata to export the graph to both a png-file and a pdf-file. The input data contain two variables: `surp_ibes` is the earnings surprise in cents per share; `quintile1` is a categorical variable ranging from 1 to 5 for quintile portfolios formed based on another variable (here, analyst forecast dispersion). The three forward slashes (`///`) are used to split the code over multiple lines. This is not required, but is helpful for readability purposes because the commands used to create graphs like this can become lengthy. Similarly, the indents are used to further enhance the readability of the code:

```
graph set window fontface "Times New Roman"

twoway ///
    (kdensity surp_ibes if abs(surp_ibes)<=20 & quintile1==1, ///
        lcolor("150 1 1") ///
        lwidth(medthick) ///
        lpattern(solid) ///
        kernel(gaussian) ///
        bwidth(1)) ||  ///
    (kdensity surp_ibes if abs(surp_ibes)<=20 & quintile1==5, ///
        lcolor("38 70 156") ///
        lwidth(medthick) ///
        lpattern(dash) ///
```

18 for the graphs presented in this section).

```
        kernel(gaussian) ///
        bwidth(1)), ///
    xlabel(-20(2)20, labsize(medium)) ///
    yscale(range(.00 .17)) ///
    ylabel(0(0.05).15, labsize(medium) format(%6.2fc)) ///
    legend(on order(1 2) ///
        label(1 "Frequency for low dispersion firms (Q1)") ///
        label(2 "Frequency for high dispersion firms (Q5)") ///
        size(medium) ///
        region(lcolor(white)) ///
        ring(0) ///
        bplacement(north) ///
        cols(2)) ///
    graphregion(color(white) margin(zero)) ///
    bgcolor(white) ///
    xtitle("{bf:Earnings surprise bin (cents per share)}", ///
        size(medium) height(5)) ///
    ytitle("{bf:Frequency}", ///
        size(medium) height(5))

set printcolor asis
graph export "$path\Doc\fig_vv.png", replace
graph export "$path\Doc\fig_vv.pdf", replace
```

The next example is from Gassen and Veenman [2023], where we plot two graphs side-by-side. The figure is reproduced as Figure 4.8 below. In both graphs, we overlay a histogram with a normal density line and use the scatterplot option to present the means of a variable for each bin on the horizontal axis. Because the graphs are based on simulated data, for completeness I also include the lines of code that create the data. I split the code below in three parts. I first present the code used to create the first graph. Next, I present the code used to create the second graph. Finally, I present the lines of code used to combine the graphs. For the purpose of combining the graphs, the options `saving(one, replace)` and `saving(two, replace)` were added when creating the graphs.

The first block of code, including the simulation commands, looks as follows:

```
//Create mixture distribution of firms in different ``life cycles'':
set seed 1234
clear

set obs 1000000
gen lc=ceil(_n/(_N/3))
gen lc1=0
replace lc1=1 if lc==1

gen double roa=.
replace roa=.0055+rnormal()*.117 if lc==1
```

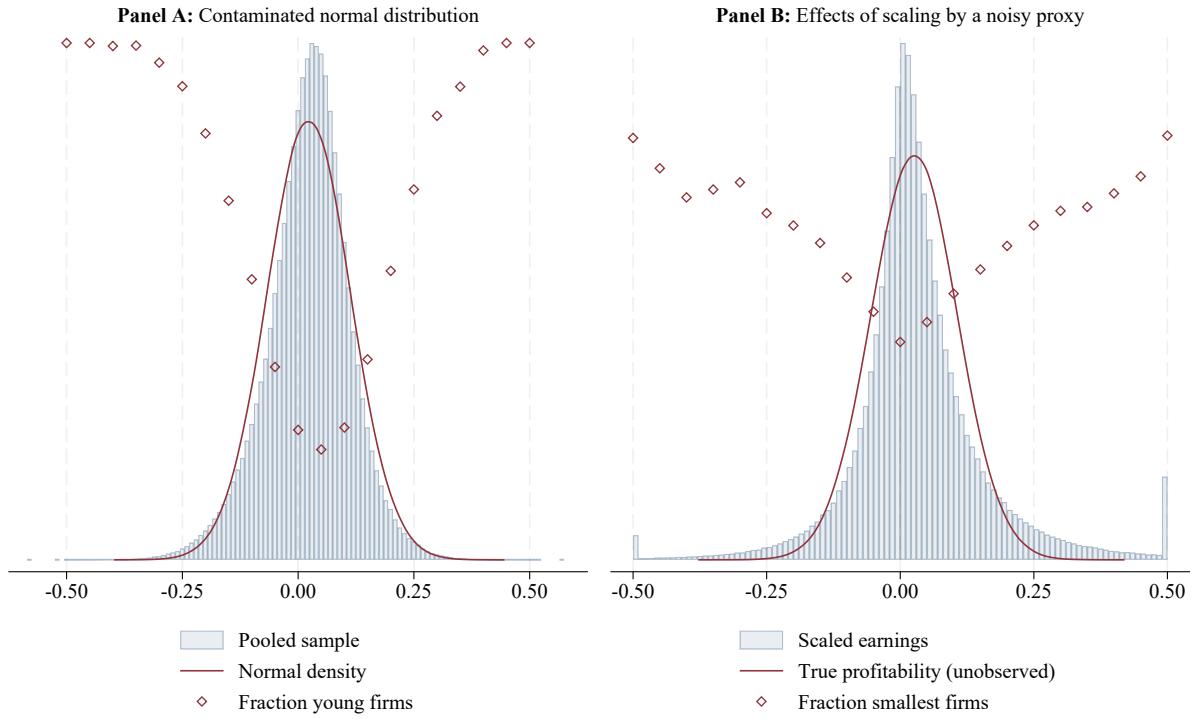


Figure 4.8: *Replication of Figure 2 from Gassen and Veenman [2023]*

```
replace roa=.0241+rnormal()*.085 if lc==2
replace roa=.0396+rnormal()*.052 if lc==3
sum roa, d
gen roanormal=r(mean)+r(sd)*rnormal()

gen double bin=round(roa*20)/20 if abs(roa)<.5
sum bin, d
replace bin=-0.5 if roa<-0.5
replace bin=0.5 if roa>0.5
egen meanlc1=mean(lc1), by(bin)
bysort bin: gen n=_n

//Create first figure:
twoway ///
    hist roa, dens color(navy%10) lcolor(navy%30) ///
        lwidth(thin) width(0.01) yaxis(1) || ///
    kdensity roanormal, bw(.01) lcolor(maroon) ///
        yaxis(1) yscale(lstyle(none) axis(1)) || ///
    scatter meanlc1 bin if n==1, msymbol(D) msize(medsmall) mlcolor(maroon)
    ↪  mcolor(none) mlwidth(medthin) ///
        yaxis(2) yscale(range(0 1) lstyle(none) axis(2)) ///
    title("{bf:Panel A:} Contaminated normal distribution", size(medium)) ///
    graphregion(color(white) margin(zero)) ///
    bgcolor(white) ///
    ylabel("", axis(1)) ///
    ylabel("", axis(2)) ///
    xlabel(-.5(.25).5, labsize(medium) format(%6.2fc)) ///
    xscale(range(-.6 .6)) ///
```

```
    legend(on order(1 2 3) ///
        label(1 "Pooled sample") ///
        label(2 "Normal density") ///
        label(3 "Fraction young firms") ///
        size(medium) ///
        region(lcolor(white)) ///
        cols(1) ///
        position(6)) ///
    ytitle("", axis(1)) ///
    ytitle("", axis(2)) ///
    xtitle("") ///
saving(one, replace)
```

The second block of code looks as follows:

```
// Simulate data to demonstrate effects of using a noisy scale proxy
set seed 1234
clear

matrix C=(1,.95\.95,1)
corr2data lnscale0 lnscale, corr(C) n(1000000) double
replace lnscale0=5.5+lnscale0*2.3
replace lnscale=5.5+lnscale*2.3

gen double roa=0.026+rnormal()*0.080
gen double scale0=exp(lnscale0)
gen double ni=scale0*roa
gen double scale=exp(lnscale)
gen double roa2=ni/scale
replace roa2=-.5 if roa2<-.5
replace roa2=.5 if roa2>.5

gen double bin=round(roa2*20)/20
xtile size=scale, nq(10)
gen small=0
replace small=1 if size==1
egen meansmall=mean(small), by(bin)
bysort bin: gen n=_n

//Create second figure:
twoway ///
    hist roa2 if abs(roa2)<=1, dens color(navy%10) ///
        lcolor(navy%30) lwidth(thin) width(0.01) || ///
    kdensity roa, bw(.01) lcolor(maroon) ///
        yaxis(1) yscale(lstyle(none) axis(1)) || ///
    scatter meansmall bin if n==1, msymbol(D) msize(medsmall) mlcolor(maroon)
    ↪ mcolor(none) mlwidth(medthin) ///
        yaxis(2) yscale(range(0 .2) lstyle(none) axis(2)) ///
    ylabel("", axis(1)) ///
    ylabel("", axis(2)) ///
    title("{bf:Panel B:} Effects of scaling by a noisy proxy", size(medium))
    ↪ ///
```

```
    graphregion(color(white) margin(zero)) ///
    bgcolor(white) ///
    xscale(range(-.52 .52)) ///
    xlabel(-.5(.25).5, labsize(medium) format(%6.2fc)) ///
    legend(on order(1 2 3) ///
        label(1 "Scaled earnings") ///
        label(2 "True profitability (unobserved)") ///
        label(3 "Fraction smallest firms") ///
        size(medium) ///
        region(lcolor(white)) ///
        cols(1) ///
        position(6)) ///
    ytitle("", axis(1)) ///
    ytitle("", axis(2)) ///
    xtitle("") ///
saving(two, replace)
```

Finally, the graphs are combined and exported as follows:

```
graph combine one.gph two.gph, graphregion(color(white) margin(zero)
↪  lpattern(none))

set printcolor asis
graph export "$path\Doc\gv_sim_combined.png", replace
graph export "$path\Doc\gv_sim_combined.pdf", replace
```

The last example is from Schafhäutle and Veenman [2023], where in Figure 3 we combine four plots of the "walkdown" in earnings and revenue forecasts over the period before earnings announcements. Although these are four plots, we actually create two figures with each having the two graphs presented side-by-side. Each graph contains two lines that capture the means of the variables that are plotted, as well as a 95-percent confidence interval around those means. Because the only differences between the four graphs are the input data and the labeling, I only present the code for the first graph below. The input data contains variable `d` (the day relative to the earnings announcement date) and variables `mean1`, `low1`, and `high1` (`mean2`, `low2`, and `high2`) that capture the mean, lower bound, and upper bound of the 95 percent confidence interval of the outcome variable based on the analyst (Estimize) consensus for each day.

```
format low1 mean1 high1 low2 mean2 high2 %6.2f
graph set window fontface "Times New Roman"

twoway ///
    rarea high1 low1 d, color("150 1 1 %15") lcolor(%0) || ///
    line mean1 d, lcolor("150 1 1 %80") || ///
    rarea high2 low2 d, fcolor("38 70 156%10") lcolor("38 70 156 %75")
    ↪  lpattern(shortdash) || ///
    line mean2 d, lpattern(shortdash) lcolor("38 70 156 %80") ///
```

```
    xlabel(-180(30)-30, labsize(medlarge)) ///
    ylabel(0.3(0.1)0.6,labsize(medlarge)) ///
    legend(on order(4 2) label(2 "Sell-side consensus") ///
        size(medlarge) label(4 "Crowdsourced consensus") ///
        region(lcolor(white)) ring(0) bplacement(swest) cols(1)) ///
        plotregion(m(zero)) ///
    graphregion(color(white) margin(zero)) ///
    xtitle("{bf:Calendar days relative to earnings announcement date}",
    ↪  size(medlarge) height(5)) ///
    ytitle("{bf:Fraction of day {it:t} consensus optimistic vs.
    ↪  pessimistic}", size(medlarge) height(5)) ///
    bgcolor(white) ///
    yscale(range(0.27 0.615)) ///
    title("{bf:Panel A:} Crowd vs. sell-side (EPS)", ring(0) size(vlarge))
    ↪  ///
saving(one, replace)
```

After having created the other three graphs in a similar way and having labeled them as `two`, `three`, and `four`, we can combine the graphs into two separate figures as follows to create Figure 4.9:

```
graph combine one.gph two.gph, graphregion(color(white) margin(zero)
↪  lpattern(none)) ysize(2.5)
set printcolor asis
graph export "$path\Doc\fig_walkdown_stata_combined.pdf", replace

graph combine three.gph four.gph, graphregion(color(white) margin(zero)
↪  lpattern(none)) ysize(2.5)
set printcolor asis
graph export "$path\Doc\fig_walkdown_stata_combined_sales.pdf", replace
```

These examples illustrate how Stata can be used to create nicely formatted figures. I have deliberately not explained each and every option that is used in the construction of these graphs, because each setting and each study will require different sets of options (and each researcher might have different preferences for formatting). Ultimately, the Stata help files will be your best guide in crafting figures for your own study.

`[help graph twoway]`

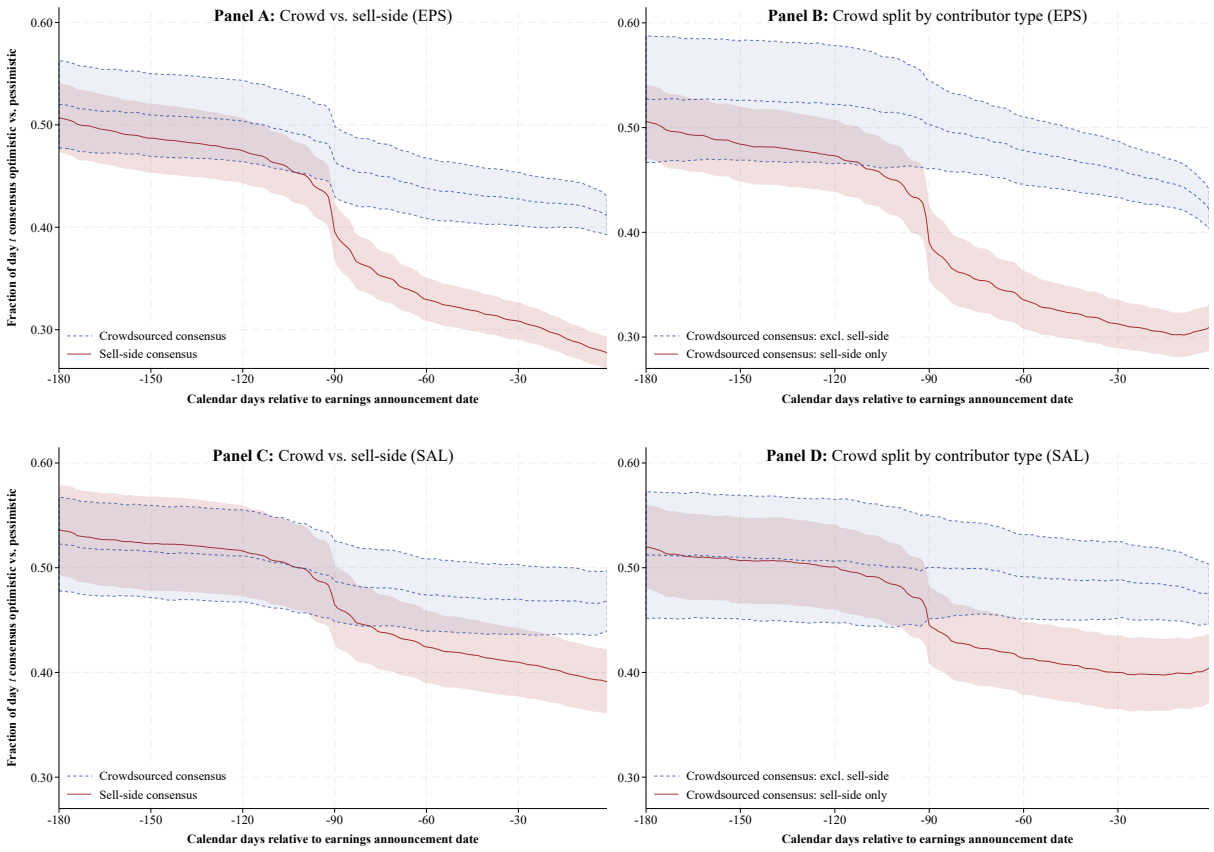


Figure 4.9: *Replication of Figure 3 from Schafhäutle and Veenman [2023]*

Chapter 5

Simulation and programming

5.1 Illustration of using Stata for simulation analysis

Besides using Stata for data management and statistical analysis, it can also be used as an effective tool for simulations. This is particularly helpful to learn to better understand the sources and consequences of specific econometric problems. For example, in Gassen et al. [2020] and Gassen and Veenman [2023] we rely on simulations to verify and quantify the econometric predictions we make, using hypothetical effects seeded in the data. Similarly, Petersen [2009] and Gow et al. [2010] use simulations to quantify to effects of dependence in regression errors on standard error estimates and to assess the effectiveness of several available remedies. Baker et al. [2022] use simulations to determine the conditions under which difference-in-difference estimates with staggered treatment assignment are biased.

The key feature of Stata (like any other statistical program) that helps us perform simulation analyses is its ability generate random numbers. We can can use these random numbers to generate specific distributions of variables, to generate relations between variables with a specific amount of noise, or to simply sort our data in a random way. For example, the `runiform()` function fills a variable with random numbers between 0 and 1 (`[help runiform()]`). That said, however, it is important to note that these numbers are not completely random. They are "pseudo-random", in the sense that they are based on deterministic functions that generate numbers *as if* they are random.[1] The good news is that because we can treat these observations as if they are truly random, we can also keep our simulations in complete control and allow for full replication. To achieve this, we can use the `set seed` command at the start of our code to

[1] You can verify this by creating a random variable based on `runiform()`, writing down the first value from the new variable, restarting Stata, and doing the same. You will see that this first value is exactly the same.

determine where it is that Stata starts in its pseudo-random number generation. The number provided to the `set seed` command can be any number between 0 and $2^{31} - 1$.

5.1.1 Example: quantifying omitted variable bias

Below is a simple example of a simulation that quantifies the problem of omitted variables bias. We create a new random variable z with values that are drawn from a standard normal distribution. We next create variable $x = 0.5z + \varepsilon$, where values for ε are drawn from a standard normal distribution as well. Variable $y = 0.5x + 0.5z + \upsilon$, where values for υ are again drawn from a standard normal distribution.

```
set seed 1234
clear
set obs 5000
gen double z=rnormal()
gen double x=.5*z+rnormal()
gen double y=.5*x+.5*z+rnormal()
```

While the random variables we draw have standard normal distributions (i.e., mean of 0 and standard deviation of 1), we can easily modify this. For example, the following adjustment to the code gives variable `z` an expected mean and standard deviation of 0.1 and 0.3, respectively:

```
. gen double z_alt=.1+.3*rnormal()

. sum z_alt

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
       z_alt |      5,000     .097052    .2991142   -.952991   1.263909
```

Although we know that the true relation between y and x is such that an increase of 1 in x should lead to an increase of 0.5 in y, the following regression estimates based on these simulated data points illustrate that the coefficient in a simple linear regression of y on x is biased. Specifically, when we do not control for variable `z`, the regression coefficient on `x` is biased upwards:

```
. reg y x

      Source |       SS           df       MS      Number of obs   =     5,000
-------------+----------------------------------   F(1, 4998)      =   2791.24
       Model |  3260.46281         1  3260.46281   Prob > F        =    0.0000
    Residual |  5838.20049     4,998  1.16810734   R-squared       =    0.3583
-------------+----------------------------------   Adj R-squared   =    0.3582
       Total |   9098.6633     4,999  1.82009668   Root MSE        =    1.0808

------------------------------------------------------------------------------
```

```
           y | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   .7168712   .0135688    52.83   0.000     .6902703    .7434721
       _cons |  -.0556353    .015285    -3.64   0.000    -.0856007   -.0256699
------------------------------------------------------------------------------
```

When we do control for `z`, we see that the regression coefficient on `x` is very close to the true value 0.5, and not significantly different from 0.5:

```
. reg y x z

      Source |       SS           df       MS      Number of obs   =     5,000
-------------+----------------------------------   F(2, 4997)      =   2161.03
       Model |  4219.83852         2  2109.91926   Prob > F        =    0.0000
    Residual |  4878.82478     4,997   .976350767   R-squared       =    0.4638
-------------+----------------------------------   Adj R-squared   =    0.4636
       Total |   9098.6633     4,999  1.82009668   Root MSE        =     .9881

------------------------------------------------------------------------------
           y | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   .5229924   .0138616    37.73   0.000     .4958176    .5501672
           z |   .4875544   .0155536    31.35   0.000     .4570625    .5180464
       _cons |  -.0466927   .0139771    -3.34   0.001     -.074094   -.0192913
------------------------------------------------------------------------------
```

This example may appear trivial, but it illustrates the foundations of a simulation analysis. The next step is to perform the same process of creating and analyzing data for a large number of iterations and to average the resulting estimates across the iterations. This is also known as Monte Carlo simulation. In the following code, we store the coefficients for `x` in variables `b1` and `b2` for the regressions without and with control for `z`, respectively.

```
set seed 1234
clear
local n=5000
set obs `n'
gen n=_n
gen b1=.
gen b2=.
forvalues i=1(1)`n'{
    qui gen double z=rnormal()
    qui gen double x=.5*z+rnormal()
    qui gen double y=.5*x+.5*z+rnormal()
    qui reg y x
    qui replace b1=_b[x] if n==`i'
    qui reg y x z
    qui replace b2=_b[x] if n==`i'
    qui keep n b1 b2
```

```
    di `i' " / " `n'
}
```

When we summarize the coefficient variables that were filled by this loop, we obtain a statistical quantification of the omitted variables bias that is caused by excluding the confounding variable from the estimation:

```
. sum b1 b2

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
          b1 |      5,000    .6999475    .0139404    .6570377   .7572708
          b2 |      5,000    .5001474     .014336    .4485804   .5555762
```

And simple t-tests (`ttest`) allow us to assess the statistical significance of the difference between the coefficient estimates and the value of 0.5:

```
. ttest b1=.5

One-sample t test
------------------------------------------------------------------------------
Variable |     Obs        Mean    Std. err.   Std. dev.   [95% conf. interval]
---------+--------------------------------------------------------------------
      b1 |   5,000    .6999475    .0001971    .0139404     .699561     .700334
------------------------------------------------------------------------------
    mean = mean(b1)                                               t =  1.0e+03
H0: mean = .5                                  Degrees of freedom =     4999

   Ha: mean < .5               Ha: mean != .5                 Ha: mean > .5
Pr(T < t) = 1.0000         Pr(|T| > |t|) = 0.0000          Pr(T > t) = 0.0000

. ttest b2=.5

One-sample t test
------------------------------------------------------------------------------
Variable |     Obs        Mean    Std. err.   Std. dev.   [95% conf. interval]
---------+--------------------------------------------------------------------
      b2 |   5,000    .5001474    .0002027     .014336      .49975    .5005449
------------------------------------------------------------------------------
    mean = mean(b2)                                               t =   0.7272
H0: mean = .5                                  Degrees of freedom =     4999

   Ha: mean < .5               Ha: mean != .5                 Ha: mean > .5
Pr(T < t) = 0.7664         Pr(|T| > |t|) = 0.4671          Pr(T > t) = 0.2336
```

We can also assess the bias more formally be computing type 1 error rates. When the model is correctly specified, in this case when it includes the confounding variable z, we should expect that in five percent of our simulations we would falsely reject the (true) null hypothesis that

$\beta = 0.5$ at a significance level of 0.05. We can test this by modifying the code to store whether or not $H_0 : \beta = 0.5$ was rejected, for each of the samples that was generated in this procedure:

```
set seed 1234
clear
local n=5000
set obs `n'
gen n=_n
gen b1=.
gen b2=.
gen sig1=0 if n<=`n'
gen sig2=0 if n<=`n'
forvalues i=1(1)`n'{
    qui gen double z=rnormal()
    qui gen double x=.5*z+rnormal()
    qui gen double y=.5*x+.5*z+rnormal()
    qui reg y x
    qui replace b1=_b[x] if n==`i'
    qui local t=(_b[x]-0.5)/_se[x]
    qui local p=2*ttail(e(df_r), abs(`t'))
    qui replace sig1=1 if `p'<=0.05 & n==`i'
    qui reg y x z
    qui replace b2=_b[x] if n==`i'
    qui local t=(_b[x]-0.5)/_se[x]
    qui local p=2*ttail(e(df_r), abs(`t'))
    qui replace sig2=1 if `p'<=0.05 & n==`i'
    qui keep n b1 b2 sig1 sig2
    di `i' " / " `n'
}
```

Inspection of the summary statistics of the two new variable we added, `sig1` and `sig2`, suggests that the type 1 error rate is very close to 0.05 for the correctly specified model that includes the additional control variable. The (true) null hypothesis that the coefficient equals 0.5 is falsely rejected in 5.36 percent of the simulations, which is not statistically significantly different from 0.05. Instead, for the misspecified model that excludes the confounding factor, we observe a type 1 error rate of 100 percent!

```
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
        sig1 |      5,000           1           0         1          1
        sig2 |      5,000       .0536    .2252492         0          1
```

[help runiform()] [help rnormal()] [help set seed]

5.1.2 Example: quantifying bias in standard error estimates

A fundamental assumption underlying the us of OLS for linear regression estimation is homoskedasticity, which means that the variance of the (unobservable) error term of the regression is constant across observations. When this is not the case, and the variance varies predictably with the our independent variable(s), the errors are heteroskedastic and we face the problem that basic standard error estimates are biased. In the simulation below, we can visualize and quantify this bias. We can also assess what happens when we apply the solution to this problem by estimating heteroskedasticity-robust standard errors instead.

In the block of code presented below, we generate a sample in which the true relation between y and x is such that each unit increase in x should lead to a unit increase in y plus noise. The noise is modeled such that the variance of this noise increases when the absolute value of x increases. This setup captures the heteroskedasticity problem described above. In addition to storing the coefficients and rejections rates in the example simulation presented in the previous section, we now additionally store the standard error estimates. In each iteration of the simulation exercise, we estimate the regression of y on x two times, one without and one with correction for heteroskedasticity in the calculation of the standard errors.

```
set seed 1234
clear
local n=5000
set obs `n'
gen n=_n
gen b1=.
gen b2=.
gen se1=.
gen se2=.
gen sig1=0 if n<=`n'
gen sig2=0 if n<=`n'
forvalues i=1(1)`n'{
    qui gen double x=rnormal()
    qui gen double y=x+abs(x)*rnormal()
    qui reg y x
    qui replace b1=_b[x] if n==`i'
    qui replace se1=_se[x] if n==`i'
    qui local t=(_b[x]-1)/_se[x]
    qui local p=2*ttail(e(df_r), abs(`t'))
    qui replace sig1=1 if `p'<=0.05 & n==`i'
    qui reg y x, r
    qui replace b2=_b[x] if n==`i'
    qui replace se2=_se[x] if n==`i'
    qui local t=(_b[x]-1)/_se[x]
    qui local p=2*ttail(e(df_r), abs(`t'))
```

```
    qui replace sig2=1 if `p'<=0.05 & n==`i'
    qui keep n b1 b2 se1 se2 sig1 sig2
    di `i' " / " `n'
}
```

The summary statistics obtained after generating the 5,000 samples, presented below, show that the coefficient estimates obtained when not adjusting and adjusting the standard error estimates for heteroskedasticity, respectively, are obviously the same. This is because all that we changed in the second estimation was to compute the standard errors differently, by invoking the `, r` option.

```
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
          b1 |      5,000    1.000118    .0244742    .9160714   1.083236
          b2 |      5,000    1.000118    .0244742    .9160714   1.083236
         se1 |      5,000    .0141347    .0002455    .0132134   .0150223
         se2 |      5,000    .0244537    .0008664    .0216258   .0292998
        sig1 |      5,000        .264    .4408434           0          1
-------------+---------------------------------------------------------
        sig2 |      5,000       .0508     .219611           0          1
```

More interesting is the standard deviation of the coefficient of 0.02447, which approximates the true standard error of the coefficient (i.e., the one that we try to approximate with the standard error *estimate* for each sample). When we inspect the mean values of the two standard error variables `se1` and `se2`, it becomes clear that the basic standard error is substantially biased (0.01413) compared to the true standard error of 0.02447. The heteroskedasticity-robust standard errors are unbiased, as the mean estimate of 0.02445 is very close to our estimate of the true standard error. The difference in bias also becomes clear from inspecting the type 1 error rates: due to the downward bias, basic standard errors cause a substantial overrejection of the (true) null hypothesis that the coefficient equals 1 (0.264). On the other hand, the type 1 error rate based on robust standard errors almost exactly equal to 0.05 (0.0508), as it should be.

The example in this section demonstrates how we can use simulations to learn to better understand an econometric problem and its solution. The example can easily be modified to introduce other problems, such as dependence in the regression errors and the consequences of using cluster-robust standard errors. The example presented above can also easily be adjusted to assess the effects of using robust standard errors when they are actually not needed, by generating y as `y=x+rnormal()`. Try this yourself. You will see that the consequence of using

robust standard errors when they are not needed is that the standard error estimates become noisier (they have higher variance) compared to the basic standard errors.

5.1.3 Example: bootstrapping standard errors

Bootstrapping is another form of simulation analysis, which relies on the random resampling of observations from a dataset. Bootstrapping can be helpful in situations where the asymptotic (i.e., large sample) properties of an estimator are unknown or are difficult to derive analytically. As an example, Armstrong et al. [2012, Tables 5-6] use a bootstrap procedure on the Kolmogorov-Smirnov statistic to derive a p-value associated with the difference in medians across two groups. Section 4.3.3 shows an example of a bootstrap procedure applied to a robust regression estimator. For other examples, see Cameron and Trivedi [2010].

To illustrate the idea of the bootstrap, consider the hypothetical situation in which we would not know the right formula to compute standard errors that account for heteroskedasticity (see the previous section). Instead of relying on a formula to derive the standard errors, we can use a non-parametric bootstrap approach to "bootstrap the standard errors." The idea is to treat the sample we have as if it is the population, and then draw observations from this sample with replacement to derive a new bootstrap sample. This sample will be different from the original sample due to the drawing of observations with replacement, but it will similarly contain the heteroskedastic nature of the data.

This procedure of creating a new bootstrap sample from the original data is then repeated a specific number of times. Cameron and Trivedi [2010] advocate the use of 400 bootstrap samples, although more samples are often feasible and result in estimates that are more stable and precise. Let's assume we use 1,000 bootstrap replications. Then based on the 1,000 independent bootstrap samples, we can estimate our regression of interest, say y on x, for each sample and obtain 1,000 estimates of the coefficient on x (β^{B1}, β^{B2}, and so on). By taking the standard deviation of these 1,000 estimates, we obtain the bootstrap estimate of the standard error (se^{boot}) of the coefficient that we obtained with the regression estimated on our original sample ($\hat{\beta}$). Because the bootstrapped coefficient estimates follow a normal distribution, we can then create a test statistic z defined as $(\hat{\beta} - \beta_0)/se^{boot}$, where β_0 is the value of the coefficient specified under the null hypothesis of our test.

Because each of the bootstrap samples contains the same heteroskedastic nature of the regression errors as the original sample, the standard deviation of our 1,000 bootstrap-sample

coefficients will capture the consequences of this heteroskedasticity (note that the consequence of heteroskedasticity is an increase in the variance of the estimates, which reduces the efficiency of OLS).

Now let's see how we can implement this in Stata. The following block of code shows how to create one random sample of data and then apply the bootstrap to obtain an estimate of the standard error. The number of bootstrap replications used here is 1,000. Also note that because the bootstrap procedure relies on random resampling from the data, it is essential that we set the seed for replication purposes. In this procedure, we will make use of the `bsample` command to randomly draw observations from the sample that is currently in Stata's memory. Because this changes (i.e., overwrites) the data currently in Stata's memory, it is important to also use the `preserve` and `restore` commands to ensure that we can go back to the original data:

```
set seed 3
clear
local n=1000
set obs `n'
gen n=_n

gen double x=rnormal()
gen double y=abs(x)*rnormal()
reg y x
scalar b0=_b[x]
scalar t0=r(table)[3,1]

gen bboot=.
forvalues i=1(1)1000{
    qui preserve
    qui bsample
    qui reg y x
    qui restore
    qui replace bboot=_b[x] if n==`i'
    di `i'
}

sum bboot
scalar seboot=r(sd)
scalar zboot=b0/seboot
di "Original t-value: " t0
di "Z-stat based on bootstrapped standard error: " zboot
```

The first five lines set the seed and create a dataset of 1,000 observations and one variable that contains the row number. The second set of five lines creates two variables y and x that are unrelated, but where the errors are heteroskedastic (y and x are unrelated because the

conditional expectation of y does not depend on x, only its variance depends on x) and stores the coefficient estimate and t-value from a simple regression of y on x. The results of the regression for this random sample are as follows. They give us the false impression of a nonzero relation between y and x:

```
. reg y x

      Source |       SS           df       MS      Number of obs   =     1,000
-------------+----------------------------------   F(1, 998)       =      6.97
       Model |  7.02432439         1  7.02432439   Prob > F        =    0.0084
    Residual |    1006.493       998  1.00851002   R-squared       =    0.0069
-------------+----------------------------------   Adj R-squared   =    0.0059
       Total |  1013.51733       999  1.01453186   Root MSE        =    1.0042

------------------------------------------------------------------------------
           y | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |  -.0844067   .0319827    -2.64   0.008    -.1471677   -.0216457
       _cons |   -.001202   .0317636    -0.04   0.970    -.0635331    .0611291
------------------------------------------------------------------------------
```

The next set of nine lines contains the meat of the exercise, the bootstrap procedure. By applying the `bsample` command, we create a new sample by redrawing cases from the original sample. The sample size will be the same. However, the composition of the sample changes because some observations are drawn multiple times, while others are not drawn at all. After estimating the regression for the bootstrap sample, we `restore` the data to the sample that was in memory at the time we applied the `preserve` command, and then store the coefficient in variable `bboot`. This sequence of steps is then repeated another 999 times.

In the final set of five lines, we obtain the standard error of the original coefficient estimate (stored in scalar `b0`) from the standard deviation of the 1,000 coefficient estimates stored in variable `bboot`. Next, we can compute the z-statistic and compare it to the t-value that was obtained from the initial regression and stored in scalar `t0`. The output of these steps looks as follows:

```
. sum bboot

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
       bboot |      1,000   -.0827179    .0510604  -.2406291   .0998563

. scalar seboot=r(sd)

. scalar zboot=b0/seboot
```

```
. di "Original t-value: " t0
Original t-value: -2.6391384

. di "Z-stat based on bootstrapped standard error: " zboot
Z-stat based on bootstrapped standard error: -1.6530748
```

The output clearly shows that when we compute the standard errors using the bootstrap procedure, inferences change quite drastically. While the initial t-value of 2.64 was well above the standard critical value of 1.96, the bootstrap z-statistics of 1.65 is well below 1.96. Which one should we trust? Well, here we know that heteroskedasticity-robust standard errors would give us unbiased estimates (remember that, for the purpose of this illustration, we assumed in this exercise that we did not know how to calculate those). Applying heteroskedasticity-robust standard errors via the `, r` option provides us with standard errors that are close to those based on the bootstrapped standard errors:

```
. reg y x, r

Linear regression                               Number of obs     =      1,000
                                                F(1, 998)         =       2.69
                                                Prob > F          =     0.1010
                                                R-squared         =     0.0069
                                                Root MSE          =     1.0042

------------------------------------------------------------------------------
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |  -.0844067   .0514239    -1.64   0.101     -.185318    .0165047
       _cons |   -.001202   .0317675    -0.04   0.970    -.0635407    .0611367
------------------------------------------------------------------------------
```

This example illustrates the basics of a bootstrap procedure for the purpose of hypothesis testing. Although this specific bootstrap is not very useful by itself, because we can simply rely on the analytical heteroskedasticity-robust standard errors available via option `, r`, the goal was to introduce the basics of the bootstrap. In practice, bootstrap procedures that have added value over and above analytical options are much more complex than this example (for example, the bootstrap procedure in program `boottest`, see Roodman et al. [2019] and Section 4.5.2). Also note that we did not actually have to write a program ourselves to implement the bootstrap since it is readily available in Stata for several estimators. Of course that is not relevant, since the point here was to provide an understanding of how the bootstrap works and

how Stata can be used to create such a procedure. For completeness, below are the command and results from using Stata's built-in bootstrap option:

```
. reg y x, vce(bootstrap, reps(1000))
(running regress on estimation sample)

Bootstrap replications (1,000): .........10.........   ...   ........1,000 done

Linear regression                                       Number of obs =  1,000
                                                        Replications  =  1,000
                                                        Wald chi2(1)  =   2.48
                                                        Prob > chi2   = 0.1155
                                                        R-squared     = 0.0069
                                                        Adj R-squared = 0.0059
                                                        Root MSE      = 1.0042

------------------------------------------------------------------------------
             |   Observed   Bootstrap                         Normal-based
           y | coefficient  std. err.      z    P>|z|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |  -.0844067   .0536335    -1.57   0.116    -.1895264     .020713
       _cons |   -.001202   .0314408    -0.04   0.970    -.0628248    .0604208
------------------------------------------------------------------------------
```

Note that `bootstrap, reps(1000): reg y x` would have given the same results. Also keep in mind again that with these types of procedures, we should always set the seed at the beginning. Although this may not be clear from the code presented above, I ran the above bootstrap procedure at the end of my do-file (which started with the line `set seed 3`), such that the final results are again perfectly replicable when I fully rerun the complete do-file. Another option could be to include `seed(...)` as an option for the bootstrap after `reps(1000)`.

In a similar vein, we can use a bootstrap procedure to obtain standard errors that are cluster-robust. The process is similar, except that we now invoke the `bsample` command with the additional option `, cluster()`. The difference implied by this additional option is that now we perform a "block" or "pairs-cluster" bootstrap by resampling not individual observations, but entire clusters of observations. For example, if would want to cluster by firm in a panel dataset of firms and years, we would draw the entire time-series of yearly observations of a firm and then draw the entire time-series of another firm in the data. The consequence of drawing entire clusters instead of individual observations is that we do not break the clustering in the data, which means that the bootstrap coefficient estimates will capture the variation that is caused by the clustering problem (recall that the problem with clustering in the data is that an individual observation in a cluster does not provide unique information for the estimation,

which increases the variance of the estimator).[2]

The following block of code provides an example. First, a dataset is created in which y and x are again independent, but both x and the unexplained part of y (the error) contain a firm-specific component that causes basic standard errors in a regression of y on x to be biased. The remainder of the code is similar to that used in the earlier example in this section, except that we now use `bsample, cluster(firm)`:

```
set seed 3
clear
local n=1000
set obs `n'
gen n=_n
gen firm=ceil(_n/10)
bysort firm: gen year=_n

gen x=rnormal() if year==1
replace x=x[_n-1] if year>1
replace x=x+rnormal()
gen e=rnormal() if year==1
replace e=e[_n-1] if year>1
replace e=e+rnormal()
gen y=e

reg y x
scalar b0=_b[x]
scalar t0=r(table)[3,1]

gen bboot=.
forvalues i=1(1)1000{
    qui preserve
    qui bsample, cluster(firm)
    qui reg y x
    qui restore
    qui replace bboot=_b[x] if n==`i'
    di `i'
}

sum bboot
scalar seboot=r(sd)
scalar zboot=b0/seboot
di "Original t-value: " t0
di "Z-stat based on cluster-bootstrapped standard error: " zboot
```

The results from this exercise reveal that the bootstrap approach provides us with standard errors (t-values) that are substantially larger (smaller) than those based on a regression

[2] Another consequence of drawing entire clusters is that the size of the bootstrap samples will not always be exactly equal to the size of the original sample when that sample is an unbalanced panel, but this is not a problem since the average size of the bootstrap sample will approximate the original sample size.

estimated without cluster-robust standard errors:

```
. di "Original t-value: " t0
Original t-value: -2.1683071

. di "Z-stat based on cluster-bootstrapped standard error: " zboot
Z-stat based on cluster-bootstrapped standard error: -1.1666433
```

The estimated t-value of 1.17 is very close to the one we would get by estimating a regression with standard errors clustered by `firm` (1.21) and to that obtained by invoking Stata's built-in bootstrap procedure (1.20):[3]

```
. reg y x, cluster(firm)

Linear regression                               Number of obs     =      1,000
                                                F(1, 99)          =       1.45
                                                Prob > F          =     0.2310
                                                R-squared         =     0.0047
                                                Root MSE          =     1.4835

(Std. err. adjusted for 100 clusters in firm)
------------------------------------------------------------------------------
             |               Robust
           y | Coefficient  std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |  -.0759898   .0630495    -1.21   0.231    -.2010937     .049114
       _cons |   .1829946   .1146466     1.60   0.114    -.0444891    .4104783
------------------------------------------------------------------------------

. reg y x, vce(bootstrap, reps(1000) cluster(firm))
(running regress on estimation sample)

Bootstrap replications (1,000): .........10........   ...   ........1,000 done

Linear regression                                     Number of obs =  1,000
                                                      Replications  =  1,000
                                                      Wald chi2(1)  =   1.45
                                                      Prob > chi2   = 0.2290
                                                      R-squared     = 0.0047
                                                      Adj R-squared = 0.0037
                                                      Root MSE      = 1.4835

(Replications based on 100 clusters in firm)
------------------------------------------------------------------------------
             |   Observed   Bootstrap                         Normal-based
```

[3] As a more technical matter, our implementation of the pairs-cluster bootstrap above and that in Stata produces a z-statistic under the assumption that the bootstrap estimates will follow a normal distribution. In samples with relatively few clusters, however, it may be safer to evaluate the statistic as a t-value and compare it to a critical value from a t-distribution that is obtained based on setting the number of degrees of freedom equal to $G - 1$, where G is the number of clusters (Cameron et al. [2008] even propose using $G - 2$). Cameron and Miller [2015] further advocate the use of the same finite-sample correction as is applied when obtaining clustered standard errors from the analytical cluster-robust variance estimator.

```
           y | coefficient  std. err.      z    P>|z|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |  -.0759898   .0631762    -1.20   0.229    -.1998129    .0478332
       _cons |   .1829946   .1156431     1.58   0.114    -.0436617    .4096509
------------------------------------------------------------------------------
```

Again, because the bootstrap is readily available in Stata and the basic bootstrapping of standard errors (whether cluster-robust or not) is asymptotically not better than the analytical cluster-robust estimates (Cameron and Miller [2015]), remember that the purpose of this exercise is purely to demonstrate the idea of how we can build such a procedure in Stata. A procedure that works better is the restricted wild cluster bootstrap discussed in Section 4.5.2, which is based on the bootstrapping of t-values instead of the coefficient estimates and which provides "asymptotic refinement" (Cameron et al. [2008]).

`[help bsample]` `[help bootstrap]`

5.2 Creating and using programs in Stata

In Section 5.1.3 we created a procedure to bootstrap standard errors. We also created a random sample in which a clustering problem was introduced (i.e., a lack of independence in variable x and the regression errors). If we would want to incorporate these steps into a simulation exercise as described in Section 5.1, our code could become a bit messy. The solution to this issue is to define separate programs that (a) generate the panel dataset with the dependence structure and (b) contain the bootstrap procedure we wrote. We can then "call" these programs in each iteration of our simulation exercise.

The following code block illustrates how we can package the lines that create the panel dataset into a simple program named "panelframe". Because the contents of the program will remain in memory until we exit Stata, it is often useful to add the first line starting with `capture ...` to ensure the program is recompiled every time we redo our do-file (otherwise we would obtain an error message that the program named "panelframe" already exists):

```
capture program drop panelframe
program define panelframe
    qui gen x=rnormal() if year==1
    qui replace x=x[_n-1] if year>1
    qui replace x=x+rnormal()
    qui gen e=rnormal() if year==1
    qui replace e=e[_n-1] if year>1
    qui replace e=e+rnormal()
```

```
    qui gen y=e
end
```

We can similarly package our lines for the bootstrapping procedure in a program called "boot-program":

```
capture program drop bootprogram
program define bootprogram
syntax, nb(integer)
tempvar bboot
qui gen `bboot'=.
forvalues i=1(1)`nb'{
    qui preserve
    qui bsample, cluster(firm)
    qui reg y x
    qui restore
    qui replace `bboot'=_b[x] if n==`i'
}
qui sum `bboot'
scalar seboot=r(sd)
end
```

And the following block of code captures the full contents of a do-file in which we perform 1,000 simulations, where in each simulation we generate a panel dataset with 1,000 observations and we estimate the standard errors using no correction for clustering, the standard adjustment for clustering, and the bootstrap approach to account for clustering. Because this example embeds a bootstrap procedure within a simulation procedure, I reduced the number of bootstrap replications from 1,000 to 100 for speed considerations. This should not be a concern because any noise that is introduced by this lower number will be canceled out across the simulations. For a single sample, however, keep in mind that higher numbers of bootstrap replications are advised.

```
set seed 1234
clear
local n=1000
local nboot=100
local sims=1000 // Note: should not be bigger than n here
set obs `n'
gen n=_n
gen firm=ceil(_n/10)
bysort firm: gen year=_n

capture program drop panelframe
program define panelframe
    qui gen x=rnormal() if year==1
    qui replace x=x[_n-1] if year>1
    qui replace x=x+rnormal()
```

```
    qui gen e=rnormal() if year==1
    qui replace e=e[_n-1] if year>1
    qui replace e=e+rnormal()
    qui gen y=e
end

capture program drop bootprogram
program define bootprogram
    syntax, nb(integer)
    tempvar bboot
    qui gen `bboot'=.
    forvalues i=1(1)`nb'{
        qui preserve
        qui bsample, cluster(firm)
        qui reg y x
        qui restore
        qui replace `bboot'=_b[x] if n==`i'
    }
    qui sum `bboot'
    scalar seboot=r(sd)
end

gen b=.
gen se_1=.
gen se_2=.
gen se_3=.
gen sig_1=0 if n<=`sims'
gen sig_2=0 if n<=`sims'
gen sig_3=0 if n<=`sims'
forvalues j=1(1)`sims'{
    // Create dataset:
        panelframe
    // Basic regression:
        qui reg y x
        scalar b0=_b[x]
        qui replace b=_b[x] if n==`j'
        qui replace se_1=_se[x] if n==`j'
        qui replace sig_1=1 if r(table)[4,1]<=0.05 & n==`j'
    // Basic regression with clustered SEs:
        qui reg y x, cluster(firm)
        qui replace se_2=_se[x] if n==`j'
        qui replace sig_2=1 if r(table)[4,1]<=0.05 & n==`j'
        local df_r=e(df_r)
    // Bootstrapped clustered SEs:
        bootprogram, nb(`nboot')
        qui replace se_3=seboot if n==`j'
        local t=b0/seboot
        local p=2*ttail(`df_r', abs(`t'))
        qui replace sig_3=1 if `p'<=0.05 & n==`j'
        drop y x e
    di `j'
```

```
}
```

The results that we obtain from running this do-file illustrate how, again, the standard clustered standard errors and the bootstrapped clustered standard errors are close to the true standard error in the data (as captured by the standard deviation of the coefficient estimates). As a result, we observe type 1 error rates that are close to 0.05:

```
. sum b se_* sig_*

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
           b |      1,000    .0006283    .0562508  -.1367834    .198946
        se_1 |      1,000    .0316782    .0017903   .0266349   .0371801
        se_2 |      1,000    .0558253    .0064997     .03763   .0778802
        se_3 |      1,000    .0554843    .0075367   .0350296   .0866635
       sig_1 |      1,000        .276    .4472405          0          1
-------------+---------------------------------------------------------
       sig_2 |      1,000        .048    .2138732          0          1
       sig_3 |      1,000        .051    .2201078          0          1
```

5.3 Creating ado-programs and using Mata

The next step in programming in Stata is to create full ado-file programs that can be run from the command line, instead of including small programs in our do-files. Examples of such programs are `reghdfe` and `robreg`, which were discussed earlier in this guide and that can be downloaded from Stata's SSC. But we can also create our own programs that we store in our local Stata folder or that we can even share with the outside world. While it is impossible for me to provide a complete description of how to create useful and well-structured ado-file programs in Stata (for much better and more complete guidance, see Baum [2015]), the purpose of this section is to use two illustrations to showcase some of the different aspects of ado-file programming. The first illustration is relatively basic, while the second is more complex and additionally showcases the use of the powerful Mata programming environment.

5.3.1 Simple example

This section provides an example of an ado-program that is created to obtain two-way clustered standard errors. The program can be viewed as an adjusted version of the program `cluster2` of Petersen [2009].[4] The difference with that program is that now, similar to `reghdfe`,

[4] See http://www.kellogg.northwestern.edu/faculty/petersen/htm/papers/se/cluster2.ado.

we set the degrees of freedom equal to the number of clusters in the smallest clustering dimension minus one, and we apply a different finite-sample correction (see Section 4.5.1 for more discussion on these concepts).

Let's first have a look at the output from running this new program, named "cluster2alt", on a simulated panel data sample:[5]

```
. cluster2alt y x x2 x3, cluster(firm year)
Linear regression with two-way cluster-robust standard errors

                                                Number of obs.:      1,000
                                                R-squared:          0.4828
                                                Adj. R-squared:     0.4813
Number of clusters in dimension firm: 100
Number of clusters in dimension year: 10
Degrees of freedom: 9
------------------------------------------------------------------------------
             | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   1.007756   .0592429    17.01   0.000     .8737391    1.141773
          x2 |  -.0401548   .0526722    -0.76   0.465    -.1593076    .0789981
          x3 |    .060455    .034816     1.74   0.117    -.0183043    .1392143
       _cons |   .1507776   .1142741     1.32   0.220    -.1077283    .4092835
------------------------------------------------------------------------------
```

The `cluster2alt` program was written in Stata's do-file editor and saved as "cluster2alt.ado" in the "c" subfolder of Stata's "PLUS" folder (the location of this folder can be found by using the `sysdir` command; see also Section 2.16). Before we go through the code step-by-step, I first copy the full contents of the ado-file below.

```
*! version 1.0 20230808 DVeenman

program define cluster2alt, eclass sortpreserve
    syntax varlist(numeric) [in] [if], cluster(varlist)

    marksample touse
    markout `touse' `varlist'
    tokenize `varlist'
    local depv `"`1'"'
    _fv_check_depvar `depv'
    macro shift 1
    local indepv "`*'"

    // Ensure exactly two variables included in cluster():
    local nc: word count `cluster'
    if (`nc'!=2){
```

[5] The results of this estimation are identical to those obtained when estimated using `reghdfe`.

```
    di as text "ERROR: cluster() should include two clustering
    ↪  dimensions"
    exit
}
local clusterdim1: word 1 of `cluster'
local clusterdim2: word 2 of `cluster'
if ("`clusterdim1'"=="`clusterdim2'"){
   di as text "ERROR: cluster() should include two unique clustering
   ↪  dimensions"
    exit
}
tempvar intersection
qui egen `intersection'=group(`clusterdim1' `clusterdim2') if `touse'
qui sum `intersection'
local nclusterdim3=r(max)

// Store results from estimation in first clustering dimension:
qui reg `depv' `indepv' if `touse', cluster(`clusterdim1')
scalar e_N=e(N)
scalar e_r2=e(r2)
scalar e_r2_a=e(r2_a)
local nclusterdim1=e(N_clust)
matrix b=e(b)
matrix V1=e(V)

// Store results from estimation in second clustering dimension:
qui reg `depv' `indepv' if `touse', cluster(`clusterdim2')
local nclusterdim2=e(N_clust)
matrix V2=e(V)

// Store results from estimation in intersection of clustering
↪  dimensions:
qui reg `depv' `indepv' if `touse', cluster(`intersection')
matrix V3=e(V)

// Get the right degrees of freedom based on smallest clustering
↪  dimension:
if (`nclusterdim1'<`nclusterdim2') {
    scalar e_df_r=`nclusterdim1'-1
}
else {
    scalar e_df_r=`nclusterdim2'-1
}

// Compute finite-sample corrections for variance matrices:
local N=e_N
local k=rowsof(V1)
local factor1=(`nclusterdim1'/(`nclusterdim1'-1))*((`N'-1)/(`N'-`k'))
local factor2=(`nclusterdim2'/(`nclusterdim2'-1))*((`N'-1)/(`N'-`k'))
local factor3=(`nclusterdim3'/(`nclusterdim3'-1))*((`N'-1)/(`N'-`k'))
if `nclusterdim1'<`nclusterdim2'{
```

```
        local factormin=`factor1'
    }
    else{
        local factormin=`factor2'
    }

    // Unadjust variance matrices for finite-sample corrections:
    matrix V1=V1/`factor1'
    matrix V2=V2/`factor2'
    matrix V3=V3/`factor3'

    // Create combined variance matrix with finite-sample correction based on
    ↪  smallest clustering dimension:
    matrix V=V1+V2-V3
    matrix V=`factormin'*V

    // Export and display results:
    local indepvnames "`indepv' _cons"
    matrix colnames b =`indepvnames'
    matrix colnames V =`indepvnames'
    matrix rownames V =`indepvnames'
    ereturn clear
    ereturn post b V
    ereturn scalar N=e_N
    ereturn scalar r2=e_r2
    ereturn scalar r2_a=e_r2_a
    ereturn scalar df_r=e_df_r
    ereturn local cmd "cluster2alt"

    di as text "Linear regression with two-way cluster-robust standard
    ↪  errors"
    di " "
    di _column(53) as text "Number of obs.: " %10.0fc as result e(N)
    di _column(53) as text "R-squared:         " %7.4f as result e(r2)
    di _column(53) as text "Adj. R-squared:    " %7.4f as result e(r2_a)
    di as text "Number of clusters in dimension " as result "`clusterdim1'"
    ↪  as text ": " as result "`nclusterdim1'"
    di as text "Number of clusters in dimension " as result "`clusterdim2'"
    ↪  as text ": " as result "`nclusterdim2'"
    di as text "Degrees of freedom: " as result e_df_r
    ereturn display
end
```

The first lines of the file contain important information about the program. The line starting with `*!` provides information on the version, date, and the author of the program. Then the line starting with `program` defines the program, where the name specified should be identical to the name of the ado-file. The option `eclass` determines that this is an e-class program, because it is an estimation command similar to `regress`. The key alternative is an r-class function such

as `summarize`. The additional option `sortpreserve` ensures that when any of the procedures included in the program causes the data to be sorted, this option ensures that the data will be restored to the order in which the data were sorted before the program was run.

```
*! version 1.0 20230808 DVeenman

program define cluster2alt, eclass sortpreserve
syntax varlist(numeric) [in] [if], cluster(varlist)
```

The line starting with `syntax` provides the program syntax and indicates that the program allows as input a list of names of numeric variables. It also allows for optional `in` and `if` qualifiers as conditional statements (note that the square brackets denote the program's options that are optional). Finally, the option `cluster(varlist)` is a required option that specifies the variables that capture the two clustering dimensions.

The following set of lines parses the input provided to the program on the command line. The `marksample touse` command creates a temporary indicator variable named "touse", where values are set to 1 based on the input provided in the `in` and/or `if` qualifiers. If `in` and `if` are not specified, all values are set equal to 1. The line with `markout `touse' `varlist'` is used to set `touse` equal to 0 for those observations that have missing values for any of the variables specified in `varlist` (i.e., they are "marked out" of the sample). Note that the local macro `varlist` contains the string of variable names that we include on the command line. As with `regress`, in our program this string should start with the name of the dependent variable and then be followed by the names of the independent variables.

```
marksample touse
markout `touse' `varlist'
tokenize `varlist'
local depv `"`1'"'
_fv_check_depvar `depv'
macro shift 1
local indepv "`*'"
```

The `tokenize` command divides `varlist` into tokens that are stored in local macros '1', '2', and so on. We can use this to extract the dependent variable as the first token of `varlist`, which we do in the following line with `local depv `"`1'"'`. In the line starting with `_fv_check_depvar `depv'`, we use a Stata tool that checks to ensure that the dependent variable we include is not specified as a factor variable (e.g., we should not start the command with `i.year`). Next, we use `macro shift 1` to drop the first element from the list

of tokens created by `tokenize`, such that we can store the list of independent variables in a separate local macro called "indepv" using `local indepv "`*'"`.

The next set of lines is used to do some housekeeping. We first let the program ensure that the `cluster()` option contains two separate clustering dimensions specified by the researcher. Next, we create a temporary variable that captures the intersection of the two clustering dimensions, since this is needed for the calculation of the two-way cluster robust variance estimator (e.g., Cameron et al. [2011]).

```
local nc: word count `cluster'
if (`nc'!=2){
    di as text "ERROR: cluster() should include two clustering dimensions"
    exit
}
local clusterdim1: word 1 of `cluster'
local clusterdim2: word 2 of `cluster'
if ("`clusterdim1'"=="`clusterdim2'"){
    di as text "ERROR: cluster() should include two unique clustering
    ↪  dimensions"
    exit
}
tempvar intersection
qui egen `intersection'=group(`clusterdim1' `clusterdim2') if `touse'
qui sum `intersection'
local nclusterdim3=r(max)
```

Before moving to the key parts of the program, let's first briefly think about what it is that the program is supposed to do. As shown by Cameron et al. [2011] and Thompson [2011], the two-way cluster robust variance estimator, from which we obtain the cluster-robust standard errors, can be derived using three separate matrices. The first matrix ($V1$) is the variance matrix obtained from a regression with standard errors clustered by the first dimension. The second matrix ($V2$) is the variance matrix obtained from a regression with standard errors clustered by the second dimension. The third matrix ($V3$) is the variance matrix obtained from a regression with standard errors clustered by the intersection of the two clustering dimensions. Given these three matrices, the cluster-robust variance estimator is then derived as $V1 + V2 - V3$.

Given this background information, our program should estimate separate regressions and store the variance matrices for each of these three dimensions. This is what the following block of code does. Along the way, we also store some additional relevant information, such as the sample size, explanatory power, and the numbers of clusters in the first two dimensions:

```
// Store results from estimation in first clustering dimension:
qui reg `depv' `indepv' if `touse', cluster(`clusterdim1')
```

```
scalar e_N=e(N)
scalar e_r2=e(r2)
scalar e_r2_a=e(r2_a)
local nclusterdim1=e(N_clust)
matrix b=e(b)
matrix V1=e(V)

// Store results from estimation in second clustering dimension:
qui reg `depv' `indepv' if `touse', cluster(`clusterdim2')
local nclusterdim2=e(N_clust)
matrix V2=e(V)

// Store results from estimation in intersection of clustering dimensions:
qui reg `depv' `indepv' if `touse', cluster(`intersection')
matrix V3=e(V)
```

The next part of the code is what makes the output identical to that of `reghdfe`, and different from the original program `cluster2`. We first determine the degrees of freedom as equal to the number of clusters in the smallest clustering dimension minus one. Next, we reconstruct the finite-sample corrections that were used to transform the variance matrices we stored as `V1`, `V2`, and `V3`. We do so because we will first re-transform these matrices based on these corrections, because we want to apply such a finite-sample correction to the final variance matrix that is constructed from the three individual matrices. The latter is done in the final lines of the code shown below.

```
// Get the right degrees of freedom based on smallest clustering dimension:
if (`nclusterdim1'<`nclusterdim2') {
    scalar e_df_r=`nclusterdim1'-1
}
else {
    scalar e_df_r=`nclusterdim2'-1
}

// Compute finite-sample corrections for variance matrices:
local N=e_N
local k=rowsof(V1)
local factor1=(`nclusterdim1'/(`nclusterdim1'-1))*((`N'-1)/(`N'-`k'))
local factor2=(`nclusterdim2'/(`nclusterdim2'-1))*((`N'-1)/(`N'-`k'))
local factor3=(`nclusterdim3'/(`nclusterdim3'-1))*((`N'-1)/(`N'-`k'))
if `nclusterdim1'<`nclusterdim2'{
    local factormin=`factor1'
}
else{
    local factormin=`factor2'
}

// Unadjust variance matrices for finite-sample corrections:
matrix V1=V1/`factor1'
```

```
matrix V2=V2/`factor2'
matrix V3=V3/`factor3'

// Create combined variance matrix with finite-sample correction based on
↪  smallest clustering dimension:
matrix V=V1+V2-V3
matrix V=`factormin'*V
```

And finally, the lines at the bottom of the program before the `end` statement reflect the commands that are needed to export the estimation results stored in matrices `b` and `V` and to display the results in a way we are familiar with as in programs such as `regress`:

```
// Export and display results:
local indepvnames "`indepv' _cons"
matrix colnames b =`indepvnames'
matrix colnames V =`indepvnames'
matrix rownames V =`indepvnames'
ereturn clear
ereturn post b V
ereturn scalar N=e_N
ereturn scalar r2=e_r2
ereturn scalar r2_a=e_r2_a
ereturn scalar df_r=e_df_r
ereturn local cmd "cluster2alt"

di as text "Linear regression with two-way cluster-robust standard errors"
di " "
di _column(53) as text "Number of obs.: " %10.0fc as result e(N)
di _column(53) as text "R-squared:          " %7.4f as result e(r2)
di _column(53) as text "Adj. R-squared:    " %7.4f as result e(r2_a)
di as text "Number of clusters in dimension " as result "`clusterdim1'" as
↪  text ": " as result "`nclusterdim1'"
di as text "Number of clusters in dimension " as result "`clusterdim2'" as
↪  text ": " as result "`nclusterdim2'"
di as text "Degrees of freedom: " as result e_df_r
ereturn display
```

5.3.2 Example including Mata

The example presented in this section is a bit more complex, because it additionally makes use of Stata's separate Mata programming environment. Mata is an extremely versatile and efficient programming environment that can significantly speed up computations in Stata programs. It is impossible and beyond the scope of this guide to provide a comprehensive overview of the possibilities of Mata here, so please consider this as one example application.[6] If you

[6]For another example, see Chapter A.8.

want to learn more about Mata programming, see Chapters 13 and 14 of Baum [2015] for an introduction, or Gould [2018] for a more complete account.

Below you can find the output that is obtained after running the ado-program `bootfast` that performs a pairs-cluster bootstrap procedure similar to Stata's built-in procedure or the program we created earlier in Section 5.1.3. The key differences are that (a) we use Mata for the estimations and (b) we exploit the fact that in the clustering setting we randomly redraw entire clusters of data instead of individual observations in the bootstrap. This creates the possibility to write programs that are computationally extremely efficient when the number of clusters is not extremely large (see MacKinnon [2023]). The following output is obtained by running the estimation command on a simulated panel of firms and years where we cluster by firm:

```
. bootfast y x x2 x3, cluster(firm) nboot(9999)

Status of bootstrap procedure: 0%....20%....40%....60%....80%....100%

Regression with cluster-robust bootstrapped standard errors
Number of bootstrap resamples: 9999
Cluster dimension: firm
Number of clusters: 100

                                                Number of obs.:      1,000
------------------------------------------------------------------------------
             | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
           x |   1.007756    .059271    17.00   0.000      .8901495    1.125362
          x2 |  -.0401548    .055124    -0.73   0.468     -.1495327    .0692232
          x3 |    .060455   .0373214     1.62   0.108     -.0135989    .1345088
       _cons |   .1507776   .1121606     1.34   0.182     -.0717733    .3733285
------------------------------------------------------------------------------
```

The program included in ado-file `bootfast` consists of two parts. The first part contains a program that is comparable to that shown in the previous section. Because of the focus on Mata here, I will not discuss the contents in detail. The most important part of this first block of code is the line `mata: _bootfast(`nboot')`. In this line, we tell Stata to move to the Mata environment and to run the Mata function `_bootfast` with as input the desired number of bootstrap replications stored in local macro `nboot`. With this command-line use of Mata, we immediately return to the Stata environment after the execution of function `_bootfast` in Mata has finished, and we can use the information that was generated in the Mata environment and that was "pushed back" into the Stata environment.

```
*! version 1.0 20230808 DVeenman
```

```
program define bootfast, eclass sortpreserve
    syntax varlist(numeric) [in] [if], cluster(varlist) nboot(integer)

    tokenize `varlist'
    marksample touse
    markout `touse' `varlist'
    local depv `"`1'"'
    // Ensure dv is not a factor variable:
    _fv_check_depvar `depv'
    macro shift 1
    local indepv "`*'"

    // Ensure only one variable included in cluster():
    local nc: word count `cluster'
    if (`nc'!=1){
        di as text "ERROR: You can define only one cluster dimension"
        exit
    }

    // Create temporary variable and local lists for variables to be read in
    ↪  Mata:
    tempvar clusterid
    egen `clusterid'=group(`cluster') if `touse'
    local y "`depv'"
    local xlist "`indepv'"
    local xlistname "`indepv' _cons"
    local cvar "`clusterid'"

    // Mata's panelsetup requires sorting on the cluster variable:
    sort `clusterid'

    // Run Mata program (defined at the bottom):
    di ""
    mata: _bootfast(`nboot')
    matrix colnames b =`xlistname'
    matrix colnames V =`xlistname'
    matrix rownames V =`xlistname'

    // Apply finite-sample correction following Cameron/Miller (2015, p. 12)
    local k=rowsof(V)
    local N=e_N
    local c=(nc/(nc-1))*((`N'-1)/(`N'-`k'))
    matrix V=`c'*V

    // Export and display results:
    ereturn clear
    ereturn post b V
    ereturn scalar df_r=e_df_r
    ereturn scalar N=e_N
    di " "
    di as text "Regression with cluster-robust bootstrapped standard errors"
```

```
    di as text "Number of bootstrap resamples: " as result `nboot'
    di as text "Cluster dimension: " as result "`cluster'"
    di as text "Number of clusters: " as result nc
    di " "
    di _column(53) as text "Number of obs.: " %10.0fc as result e(N)
    ereturn display
end
```

The second part of the ado-file, included at the bottom of the program, consists of the relevant Mata code. This Mata code is itself split up into two separate parts, because it defines two separate Mata functions. The first function is labeled `_bootfast()`. The second function is labeled `_bootcounter()`, which is simply a helper function that we use to display the status of the bootstrap procedure. The `_bootcounter()` function is used *within* the `_bootfast()` function, but it is defined *outside* of this function in the Mata environment:

```
mata:
    mata clear

    void _bootfast(real scalar B){
        // Declare variables:
        real vector y, cvar, b, coef, bsample_c, Xyb, V
        real matrix X, XXinv, info, xg, yg, beta, XXb, XXbinv
        real scalar n, nc, dfr, k, cluster
        pointer(real matrix) rowvector xxp, xyp

        // Load the data:
        y=st_data(., st_local("y"), st_local("touse"))
        X=st_data(., tokens(st_local("xlist")), st_local("touse"))
        cvar=st_data(., st_local("cvar"), st_local("touse"))

        // Obtain coefficients and info for the full sample:
        n=rows(X)
        X=(X,J(n,1,1))
        XXinv=invsym(cross(X,X))
        b=XXinv*cross(X,y)
        coef=b'
        info=panelsetup(cvar, 1)
        nc=rows(info)
        dfr=nc-1
        k=cols(X)

        // Obtain the relevant information from the clusters and store in
        ↪  lower-dimensional matrices;
        // These lower-dimensional matrices are stored using pointers in
        ↪  pointer-vectors xxp and xyp:
        xxp=J(1, nc, NULL)
        xyp=J(1, nc, NULL)
        for(i=1; i<=nc; i++) {
```

```
            xg=panelsubmatrix(X,i,info)
            xxp[i]=&(cross(xg,xg))
            yg=panelsubmatrix(y,i,info)
            xyp[i]=&(cross(xg,yg))
        }

        // Obtain and store coefficients for each bootstrap sample:
        beta=J(0,k,0)
        bcounter=0
        bcounter2=0
        pct=0
        for(b=1; b<=B; b++) {
            bsample_c=mm_sample(nc,nc)
            XXb=J(k,k,0)
            Xyb=J(k,1,0)
            for(i=1; i<=nc; i++) {
                cluster=bsample_c[i]
                XXb=XXb+*xxp[cluster]
                Xyb=Xyb+*xyp[cluster]
            }
            XXbinv=invsym(XXb)
            beta=(beta \ cross(XXbinv,Xyb)')

            // Display status of bootstrap:
            _bootcounter(b, B, bcounter, bcounter2,
             ↪  pct)
        }

        // Store variance of each coefficient and push info back to Stata
         ↪  matrices:
        V=variance(beta)
        st_matrix("b", coef)
        st_matrix("V", V)
        st_numscalar("e_N", n)
        st_numscalar("e_df_r", dfr)
        st_numscalar("nc", nc)
    }

    void _bootcounter(real scalar b, B, bcounter, bcounter2, pct){
        if (b==1) {
            printf("   Status of bootstrap procedure: 0%%")
            displayflush()
        }
        bcounter++
        bcounter2++
        if (floor(5*bcounter/B)==1) {
            pct=pct+20
            printf("%f", pct)
            printf("%%")
            displayflush()
            bcounter=0
```

```
            bcounter2=0
        }
        else{
            if (b==B) {
                printf("100%%")
                displayflush()
            }
            if (floor(25*bcounter2/B)==1) {
                printf(".")
                displayflush()
                bcounter2=0
            }
        }
        if (b==B) {
            printf("\n")
            displayflush()
        }
    }
end
```

Let's briefly discuss what this long block of Mata code embedded in the ado-file does for us. First note that the code for the two Mata functions is embedded between the following lines:

```
mata:
mata clear
...
end
```

The line with `mata:` makes Stata switch to the Mata environment. You can try this yourself by typing and executing this line in the Stata command window. Until the `end` command is executed, all commands provided to Stata will be interpreted in the Mata environment. For example, `mata clear` resets Mata's memory to ensure any previously created (and interpreted) variables and functions will be deleted before we create our new Mata variables and functions.

Next, we can see that within the Mata section of the ado-file, the two functions we create are structured as follows. I will not discuss the `_bootcounter()` function here since it is just a support function that helps display the status of the bootstrapping procedure in Stata's output window. When we focus on the key function of interest instead, i.e. `_bootfast()`, we can see that this is a function that takes as input a scalar with real numeric values. This input number is then given the name `B` within the Mata function. The definition of the function starts with the command `void`, which is Mata language for saying that the function does not need to return a specific object as output.

```
void _bootfast(real scalar B){
    ...
```

```
}

void _bootcounter(real scalar b, B, bcounter, bcounter2, pct){
        ...
}
```

Now let's look at the contents of the `_bootfast()` function. The first lines declare all the relevant variables and their types that we will be using in the function. This step is not required but is good practice. In this example, we can see that we have vectors, matrices, and scalars that will each contain real numerical values. We also see the line starting with `pointer`, which requires a bit more explanation. The "pointer" in Mata is a variable that stores the memory address of another variable or object. Pointers can be useful because they can help us store a long sequence of objects (e.g., matrices) into another object that we reference elsewhere in our program. In the specific example of the program we create here, what we will do is create separate matrices for the different clusters in our data, and then store the addresses of the matrices in a "pointer vector" so that we can easily access the contents of these matrices by de-referencing the pointers. For a more precise description of pointers in Mata, please see Baum [2015] and Gould [2018]. Here, `pointer(real matrix) rowvector xxp, xyp` means that we create two vectors of pointers called `xxp` and `xyp` that will contain the addresses of real-valued matrices.

```
real vector y, cvar, b, coef, bsample_c, Xyb, V
real matrix X, XXinv, info, xg, yg, beta, XXb, XXbinv
real scalar n, nc, dfr, k, cluster
pointer(real matrix) rowvector xxp, xyp
```

After having declared the relevant variables and their types, the next step is to load the relevant data from Stata into the Mata environment. We do so using the `st_data()` command. In the following lines, we create a Mata vector `y` that contains the values of the dependent variable of our regression, we store the values of the independent variables in matrix `X`, and we store the values of the clustering variable in vector `cvar`. The inputs `st_local("y")`, `tokens(st_local("xlist"))`, and `st_local("cvar")` rely on the local macros that defined in the first (non-Mata) part of the ado program. In addition, the `st_local("touse")` in each of the lines refers to the `touse` variable and tells Mata to store values for the vectors and matrix only for observations in the Stata dataset for which `touse` is equal to 1.

```
y=st_data(., st_local("y"), st_local("touse"))
X=st_data(., tokens(st_local("xlist")), st_local("touse"))
cvar=st_data(., st_local("cvar"), st_local("touse"))
```

Before going into the real action of this Mata function, the next set of lines is used to obtain relevant information about the regression for which we will apply the bootstrapping. Given the information on the dependent variable and independent variables stored in vector `y` and matrix `X`, respectively, we can use Mata's powerful matrix functions to compute the OLS coefficients for the regression using `XXinv=invsym(cross(X,X))` and `b=XXinv*cross(X,y)`. Note that in step `n=rows(X)`, we simply store the number of observations in the data in scalar `n`, while with `X=(X,J(n,1,1))` we add a column of ones to the `X` matrix because we want the regression to be estimated with an intercept. The line `coef=b'` tells Mata to transpose the vector of coefficients into a new vector valled `coef`.

Next, we create a new matrix called `info` that contains information derived from the function `panelsetup()`. The first argument refers to the object that contains information about the different panels (clusters) in the dataset and the second argument refers to the column of this object that contains the relevant information (in this case, we just have one column because `cvar` is a vector). The information stored in `info` can then be used for other functions that are useful in the processing of panel data. For example, we will shortly see how `panelsubmatrix()` can be used to extract the information from matrix `X` for observations that belong to a particular panel (i.e., cluster). Next, `nc=rows(info)` stores the number of clusters in the data into scalar `nc`, and `dfr=nc-1` determines the degrees of freedom based on this number, which we need for statistical inference. Lastly, with `k=cols(X)` we store the number of independent variables (including the intercept) in matrix `X`, a number we will also need in the subsequent steps.

```
n=rows(X)
X=(X,J(n,1,1))
XXinv=invsym(cross(X,X))
b=XXinv*cross(X,y)
coef=b'
info=panelsetup(cvar, 1)
nc=rows(info)
dfr=nc-1
k=cols(X)
```

The next lines are the most critical ones in making this such a computationally efficient program for the bootstrap. The first two lines create the two vectors in which we will be storing our pointers (i.e., the addresses to matrices). With `J(1, nc, NULL)` we create an empty matrix with one row and number of columns equal to `nc`. The first column will contain the address of the first matrix, second column the address of the second matrix, and so on. We can then easily store (or refer to) the address of the matrix by referring the specific column elements of

these vectors.

The following lines contain Mata's `for` loop, which we use here to iterate over the different clusters in our data. The syntax of this loop is `for(i=1; i<=nc; i++){...}`, where we iterate over index `i` which runs from a value of 1 to the value stored in scalar `nc` (the number of clusters in the data). The first line in the loop, `xg=panelsubmatrix(X,i,info)`, creates a submatrix called `xg` that contains the rows of matrix `X` that correspond to all observations that belong to cluster number `i`. For example, in the first iteration of the loop we create a submatrix for the first cluster of observations. In the third line of the loop, we do the same but then for the contents of the `y` vector, i.e., the values of the dependent variable of the regression.

In the second line of the loop, we do two things. First, we perform a matrix multiplication using the submatrix that was constructed in the first line (`cross(xg,xg)`). This multiplication is $X'X$, which is a key part of the calculation of the OLS coefficients. This multiplication changes the matrix dimensions from the original $N_c \times k$, where N_c is the number of observations in cluster `c` and `k` is the number of independent variables (including the intercept), to a dimension that is only $k \times k$. In other words, in this part of the program we transform all of the information from the original data matrix X, which can be huge, to much smaller matrices. As MacKinnon [2023] shows, this does not result in a loss of information, because the OLS estimates can still be obtained by stacking all these small submatrices. The benefit of this procedure will become evident when we start applying the bootstrap, because instead of having to redraw all observations from each cluster, we will only have to redraw the much smaller matrices we just created.[7]

Second, after having created the $k \times k$ matrices for the different clusters, we now take advantage of the concept of pointers by storing the address of each matrix we create in the loop with `cross(xg,xg)` in the pointer vector `xxp`. We do this by placing the `&` sign before the matrix. Specifically, `xxp[i]=&(cross(xg,xg))` will store the address of the matrix for cluster `i` in column `i` of vector `xxp`. Note that with this use of pointers we never have to give each matrix a specific name, as we will only have to refer to its address.

In the third line of the loop, we similarly use `panelsubmatrix()` on the `y` vector. Next, in

[7] To illustrate the efficiency gains, consider a dataset with 100,000 observations where observations belong to 100 unique clusters. Also assume we estimate a regression with an intercept and four independent variables. In this procedure, we create 100 unique matrices with dimension 5×5 that will we draw from when performing the bootstrap replication. That is, instead of having the draw 100 matrices of dimension 1000×5, we will only have to draw 100 matrices of dimension 5×5 in each bootstrap replication. Again see MacKinnon [2023] for more information.

the fourth line we create the $k \times 1$ matrix $X'y$ for each cluster and store its address in pointer vector `xyp`.

```
xxp=J(1, nc, NULL)
xyp=J(1, nc, NULL)
for(i=1; i<=nc; i++) {
    xg=panelsubmatrix(X,i,info)
    xxp[i]=&(cross(xg,xg))
    yg=panelsubmatrix(y,i,info)
    xyp[i]=&(cross(xg,yg))
}
```

The subsequent set of lines contains the bootstrap procedure. We first create an empty vector `beta` using `J(0,k,0)`. This might seem odd, since we create a matrix that has zero rows, but we will start adding rows to this matrix for each iteration of our bootstrap procedure. That is, in each iteration we will add a $1 \times k$ row vector to the matrix such that, once we have gone through the `B` bootstrap replications, we will have a $B \times k$ matrix of coefficients. In other words, we will have a matrix that contains all coefficients we have estimated for each of `B` bootstrap samples. The other scalars `bcounter`, `bcounter2`, and `pct` are created to facilitate the function `_bootcounter`.

The bootstrap action is in the `for` loop, which runs from 1 to our specified number of bootstrap replications contained in scalar `B`. In the first line of the loop, we use the `mm_sample()` function.[8] By specifying `mm_sample(nc,nc)`, we create a new vector of `nc` observations. Each value in this vector represents a random draw (with replacement) of the integers in the list `1..nc`. For example, `mm_sample(4,4)` might give us the sequence 2, 1, 1, 4. Now this vector of `nc` values between 1 and `nc` can be used as the list of clusters of data we draw for a bootstrap sample in the (pairs-cluster) bootstrap procedure.

For each cluster drawn and stored in vector `bsample_c`, we now start extracting the submatrices we created earlier from our pointer vectors to perform the calculations needed for the bootstrap coefficient estimates. Specifically, we take the following steps. First, we create a new `for` loop (which will be nested within the earlier `for` loop). Second, for each cluster listed in vector `bsample_c`, we create a temporary scalar `cluster` that contains the cluster number. Third, we use this cluster number to refer to the relevant column in the pointer vector in which the address of the submatrices for this specific cluster were stored. We do this with `*xxp[cluster]` and `*xyp[cluster]`, where the `*` sign helps us to extract the contents of the

[8] Note that this is a function that is available only through an upgrade of Mata functions available via the `moremata` package (`ssc inst moremata`). See https://github.com/benjann/moremata.

address stored in the pointer vector.

As MacKinnon [2023] shows, the coefficient estimates for the bootstrap sample can easily be obtained by constructing the relevant matrices $X'X$ and $X'y$ as the sums of the submatrices for the different clusters drawn in the bootstrap sample. Specifically, $X'X = \sum_{g=1}^{G} X_g'X_g$ and $X'y = \sum_{g=1}^{G} X_g'y_g$, where g refers to a cluster, G is the total number of clusters, and X_g and y_g refer to the submatrices for `X` and `y`, respectively. This is what we do with the matrices `XXb` and `Xyb`, respectively. We first set the values of these $k \times k$ and $k \times 1$ matrices to zero, and then when iterating through the list of clusters drawn in the bootstrap, we fill $X'X$ and $X'y$ as the sums of the submatrices for the different clusters that were drawn in the bootstrap sample.

After having gone through this procedure, we can easily obtain the OLS coefficients for the bootstrap sample and store these estimates in the matrix `beta`. Specifically, we first compute $(X'X)^{-1}$ using `XXbinv=invsym(XXb)`, and then we create the vector of coefficients for the sample using `cross(XXbinv,Xyb)`. With `beta=(beta \ cross(XXbinv,Xyb)')` we then fill the `beta` matrix row by row to store the bootstrap sample estimates.

```
beta=J(0,k,0)
bcounter=0
bcounter2=0
pct=0
for(b=1; b<=B; b++) {
    bsample_c=mm_sample(nc,nc)
    XXb=J(k,k,0)
    Xyb=J(k,1,0)
    for(i=1; i<=nc; i++) {
        cluster=bsample_c[i]
        XXb=XXb+*xxp[cluster]
        Xyb=Xyb+*xyp[cluster]
    }
    XXbinv=invsym(XXb)
    beta=(beta \ cross(XXbinv,Xyb)')
...
}
```

We are almost there, since all we need to do is use the variance of the B estimates of each coefficients as inputs for the variance matrix of the regression with bootstrapped standard errors. Recall that, in the bootstrapping of standard errors, we use the standard deviation of the coefficients as our estimate of the standard errors and that the standard errors are derived as the square roots of the diagonal entries of the variance matrix. Hence, we store the variance of the coefficients with `V=variance(beta)`. Finally, we "push" the relevant information, i.e. the baseline coefficients, bootstrapped variance estimates, and other information such as sample

size, degrees of freedom, and number of clusters back to Stata using `st_matrix` and `st_scalar`, at the end of the Mata function `_bootfast()`.

Overall, although this example is clearly very specific, its many elements help illustrate the power of Mata in programming and ado-files. Combined with theory for efficient computation (here, MacKinnon [2023]), Mata can be very helpful in creating extremely powerful and fast programs. Consistent with the objective of this exercise to create a program that is much faster than when we use Stata's built-in procedure to apply the pairs-cluster bootstrap (or the program we created earlier in Section 5.1.3), the `bootfast` program ran in only 0.9 second using 9,999 bootstrap replications on this sample of 1,000 observations, while it took 41 seconds using Stata's built-in bootstrap procedure.[9]

[9]For another illustration, see https://github.com/dveenman/bootstep, which uses Mata for the calculation of bootstrapped standard errors in two-step estimations (Chen et al. [2023]).

Appendix A

Applications and examples

The remainder of this guide provides examples of procedures and calculations that can be useful for several accounting and finance research settings. The goal of these examples is to illustrate the power of Stata in solving specific problems, as well as to illustrate the relatively simple nature of the code that is needed to come to these solutions. With all of the examples, it is essential to first carefully read the relevant background literature and to understand the specific problem that is being solved. **Do not** simply copy and paste the code for your specific research application. Without a proper understanding of what it is the code is supposed to do, it is very likely that tiny mistakes in the code will lead to major errors in the data generated for subsequent analysis. At the same time, these examples assume a solid understanding of all the topics that have been discussed in the previous chapters.

A.1 Using Stata to obtain data from WRDS

A commonly used source of archival data for accounting and finance research is the Wharton Research Data Services (WRDS) platform. Depending on your university's subscriptions, this platform provides access to a wide range of relevant databases. A simple method to download data from WRDS is to use a web query on the platform. Traditionally, researchers have also often used SQL queries in the statistical program SAS for this purpose, since WRDS provides all of the data in SAS format. The use of SAS/SQL also allows a researcher to perform intensive calculations on the WRDS server before downloading the resulting dataset. Similar procedures can now be applied in R/Rstudio in combination with the `dbplyr` package.

Although (currently) not designed to perform computations on the WRDS server, Stata can now also be used as a replacement for WRDS web queries.[1] An important benefit associated with the use of Stata to download data from WRDS is that it can improve documentation and increase transparency in the research process. By starting a project with a new do-file that contains all queries used to download WRDS data (I often call this file "00_wrds_data.do" to indicate that it forms the basis all the follow-up steps), as researcher we keep track of the sources and names of variables that we use in our study. This is useful for the transparency of the research process and has practical benefits: it is much easier to check the choices we made when we have to describe the sample selection in the paper. It also becomes easier to update a dataset without having to manually re-select the specific variables on the WRDS website, and we do not have to switch between Stata and other programs such as SAS or R/Rstudio.

To be able to access and download WRDS data via Stata on your computer, you should first prepare your computer and Stata for this. See this page on WRDS for instructions how to do this in Windows. Because the page is sufficiently informative in explaining how to set this up, I will not repeat the information here and instead assume you are able to get things to work (also note that you should understand how to work with the two-factor authentication introduced by WRDS in 2023). Given the default data source name `wrds-pgdata-64` I used for the ODBC connection, in line with the WRDS instructions, the first thing we can do is to make a list of the different data sources that are available on the platform:[2]

[1] Note that everything described below was tested only on a Windows machine. WRDS currently only explains how to connect Stata to the WRDS server via an ODBC connection in Windows and it provides no information how to set this up on mac/OS.

[2] See this page on WRDS for more examples. Note that the arrow means that the command continues on the following line. When applying this command in Stata, you should ignore the arrow and instead write "wrds_lib..." on the same line as "from".

```
odbc load, exec("select distinct frname from
↪  wrds_lib_internal.friendly_schema_mapping;") dsn("wrds-pgdata-64") clear
list
```

The result is a tabulation of all the distinct names of the data sources included on the WRDS platform (for example, we see "comp" for Compustat, "crsp" for CRSP, and "ibes" for IBES). Next, we can identify the specific databases, or "tables", included on the platforms for each of these data sources. For example, to obtain a list of all tables included in IBES, we can type the following:

```
odbc load, exec("select distinct table_name from information_schema.columns
↪  where table_schema='ibes' order by table_name;") dsn("wrds-pgdata-64")
↪  clear
list
```

The first lines of the output I obtain look as follows:

```
.         list

   +-------------------+
   |        table_name |
   |-------------------|
1. |        act_epsint |
2. |         act_epsus |
3. |       act_xepsint |
4. |        act_xepsus |
...
```

For example, the table `act_epsus` contains the split-adjusted reported EPS actuals for US firms in IBES (the equivalent split-*unadjusted* data are available in table `actu_epsus`). As another example, the unadjusted monthly consensus EPS forecast data for US firms are available in table `statsumu_epsus`. For non-US firms, this information is available in `statsumu_epsint` and, for non-EPS metrics of US firms, this information is available in table `statsumu_xepsus`. We obtain similar information on the tables of other specific data sources by simply replacing "ibes" in the `odbc load` function with the name of the data source we want to use (for example, "comp" for Compustat).

Getting to know these table names and their contents requires some researching on your part (which is good, because you will need to obtain a deep understanding of your data sources anyway). The best way to find out which table you need for your research is to go to the WRDS page where you would normally perform a web query. On this page, click on the "Variable Descriptions" tab and then click on the name listed after "Product". For example,

for IBES this is called `tr_ibes` and for Compustat North America (daily updates) this is called `comp_na_daily_all`. The product page provides an overview of the different tables available for the data source and the connection between the web query and the specific table. By clicking on the table name, you can view the different variables and their descriptions included in the table. Alternatively, you can also explore the variable search page. From there you can search by data source, click on a specific table name, and observe its contents.

Now let's see how we can download data from WRDS using Stata. For example, let's say we want to download the company information table from Compustat North America and save the dataset in a local folder called "InFiles", which is part of our project folder:

```
odbc load, exec("select * from comp.company") noquote dsn("wrds-pgdata-64")
↪  clear
save "$path\InFiles\company.dta"
```

In the `exec(...)` part of the command, we send a string with an SQL query to the WRDS server similar to what researchers do when using SAS, since this is the language used on the WRDS platform. In this case, we download the full table of information. Alternatively, we can also adjust the string in such a way that we obtain only a specific subset of the data for specific variables of interest. For example, assume we would only need data on total assets from the Compustat North America Fundamentals Annual table (`funda`) for the years 2001-2020. In this case, we would use:

```
odbc load, exec("select at from comp.funda where fyear>='2001' and
↪  fyear<='2020'") noquote dsn("wrds-pgdata-64") clear
```

The result of this query is a simple dataset with one variable named "at" (Compustat's data item for total assets) for all firm-years in Compustat Fundamentals Annual between 2001 and 2020. In practice, of course, we might need more than one variable and prefer to impose some more restrictions. Because doing so requires us to write a relatively long string that includes the query, it can be useful to include a separate step in which we store the specific query into a global macro. The code below prepares a query that accesses data in the Fundamentals Annual table in Compustat Global:

```
global query ///
    select gvkey, fyear, fic, datadate, fyr, at, ib, xrd, ///
        sale, oancf, indfmt, consol, popsrc, datafmt, sich, ///
        capx, ceq, xsga, intan, ppent, ppegt ///
    from comp.g_funda ///
    where datadate>'20201231' ///
```

```
        and datadate<'20230101' ///
        and at>0
```

What we do here is simply store the SQL query starting with "select ..." into a global macro called "query". The spaces and three forward slashes (///) are not required, but they help to make the statement easier to read and to make it easier to spot any mistakes. The use of the three forward slashes is a useful tool in Stata that allows a command in a do-file to span multiple lines. After having assigned this information to global macro "query", we can now simply execute the following:

```
odbc load, exec("$query") noquote dsn("wrds-pgdata-64") clear
save "$path\InFiles\g_funda.dta"
```

As a final example, let's assume we want to download stock price data for a large sample of firms and countries from Compustat Global. Because the overall dataset is too large to download all data at once, we may want to download the data for each country separately. Assume we have already downloaded firm-year data from the Fundamentals Annual table of Compustat Global, as we have done above, and we now want to download the stock price data for all firms in the countries in our sample. To do so, we can open this dataset and use the `levelsof` function to create a local macro that includes a list of country identifiers (variable `fic`) for the countries in the sample. Next, for each of these countries we download the stock price data and save the data in a separate Stata file for each country. Assume we only want to examine countries with at least 100 firms in a given year and that we only need the stock price data for the month of January 2022:

```
use "$path\InFiles\g_funda.dta", clear
egen count=count(fyear), by(fic fyear)
sum count, d
drop if count<100

levelsof fic, clean
local countrylist=r(levels)

local j=1
foreach cntry in `countrylist'{
    di `j' ": `cntry'"
        global query ///
            select gvkey, iid, datadate, prccd ///
            from comp_global_daily.g_secd ///
            where datadate>='20220101' ///
                and datadate<='20220131' ///
                and fic in ('`cntry'')
```

```
    odbc load, exec("$query") noquote dsn("wrds-pgdata-64") clear
    save "$path\InFiles\g_secd_`cntry'.dta", replace
    local `j++'
}
```

After executing this block of code on my computer, the data sets are saved as new Stata files in the "InFiles" folder with names including the country labels. The first lines of output in Stata look as follows:

```
1: ARE
(file C:\home\dv\PROJECTS\STATA\InFiles\g_secd_ARE.dta not found)
file C:\home\dv\PROJECTS\STATA\InFiles\g_secd_ARE.dta saved
2: AUS
(file C:\home\dv\PROJECTS\STATA\InFiles\g_secd_AUS.dta not found)
...
```

Depending on the research objectives, we can now manage the data in each of these separate datasets and perform relevant calculations. Alternatively, we can combine the files into one large dataset using the following block of code (the first lines are the same as we used before to ensure the string of country codes is included in the local macro `countrylist`):

```
use "$path\InFiles\g_funda.dta", clear
egen count=count(fyear), by(fic fyear)
sum count, d
drop if count<100

levelsof fic, clean
local countrylist=r(levels)

local j=1
foreach cntry in `countrylist'{
    di `j' ": `cntry'"
    if (`j'==1) {
        use "$path\InFiles\g_secd_`cntry'.dta", clear
    }
    else {
        append using "$path\InFiles\g_secd_`cntry'.dta"
    }
    local `j++'
}
```

Based on the 50 countries included in my dataset, I obtain 1,064,163 unique daily closing stock price observations during the month of January 2022:

```
. sum prccd

    Variable |        Obs        Mean    Std. dev.       Min        Max
```

```
-------------+---------------------------------------------------------
       prccd |  1,064,163    3617.501    112139.4      .0001   1.65e+07
```

A.2 Estimating discretionary accruals

This chapter describes the use of Stata for the estimation of discretionary (or "abnormal" or "unexpected") accrual variables. While its focus is on discretionary accruals, the code can easily be modified to measure other constructs based on the residual from some prediction model, such as "excess" compensation (Core et al. [2008]), "abnormal" audit fees (Hribar et al. [2014]), or "abnormal" discretionary expenditures (Roychowdhury [2006]). The accrual model used here, which describes the expected level of accruals given a firm's fundamentals, is estimated for every industry group with at least 20 firms in a given year. Industry groups are defined by two-digit SIC codes. However, note that any group of firms can be used for the estimation (see, e.g., Ecker et al. [2013]).

Before illustrating how this works, please keep in mind that the purpose of this illustration is *only* to show how Stata can be used to compute this frequently used measure. The illustration does not imply that the resulting measure is a good proxy for constructs such as earnings management, earnings quality, audit quality, etc. There are major problems with discretionary accrual model estimations, including the limitations of the right-hand side variables and linear modeling, issues with measurement error, and the problems of using different sets of covariates in the first- and second-stage estimations (see Hribar and Nichols [2007], Gerakos [2012], and Chen et al. [2018]).

The model used here follows the commonly estimated modified-Jones model (all variables are scaled by lagged total assets):

$$TA_{it} = \beta_0 + \beta_1(1/ASSETS_{it-1}) + \beta_2(\Delta REV_{it} - \Delta REC_{it}) + \beta_3 PPE_{it} + \varepsilon_{it} \tag{A.1}$$

The sample used for this example is based on annual accounting data from the Compustat Fundamentals Annual table for US listed firms available in the CRSP database and for fiscal years in the period 2000–2021. Input variables are as listed below:

Variable	Description
gvkey	Compustat company identifier
fyear	Fiscal year
datadate	Fiscal year end date
fic	Foreign incorporation code
at	Total assets
ibc	Income before extraordinary items (from cash flow statement)
oancf	Cash flow from operations
ppegt	Gross property plant and equipment
sale	Sales revenue
rect	Accounts receivables[3]
sic	SIC industry code
sich	SIC industry code (historical)

Approach 1: Estimating regressions by group in a forvalues loop

The standard approach is to estimate the parameters in equation (A.1) using OLS for each group with a loop that runs over the different groups of observations. In this case, the groups are the two-digit SIC industry-year groups. To follow the approach described below, open the do-file editor in Stata, start a clean do-file, and save the empty do-file to a location and name of your preference. Use this do-file to store all the programming code below. Also ensure that the `global` macro `path` refers to the location of the project folder (see Section 2.15).

The first step is to open and organize the raw data. We eliminate observations that have missing or non-positive values for total assets and firms that are not incorporated in the country of interest (US in this case). Because we need the SIC code for the industry-level estimation, it is useful here to replace the `sic` variable by the historical SIC code in the `sich` variable if this variable is non-missing. Also, as is typical in studies that use accrual data, we eliminate financial firms (SIC codes 6000-6999).

```
use "$path\InFiles\ccm_annual.dta", clear
destring gvkey, replace
ren lpermno permno
drop if at==. | at<=0
keep if fic=="USA"
destring sic, replace
replace sic=sich if sich!=.
drop if sic==.
drop if sic>5999 & sic<7000
```

The following step ensures that all duplicate firm-year observations are eliminated before we let Stata learn about the panel structure of the data. We need this to be able to compute

[3]While this example uses balance sheet data to compute the change in accounts receivable (ΔREC), some studies such as Hribar and Nichols [2007] use the change in accounts receivables available as reported in the cash flow statement. Because the Compustat variable for this change (`recch`) is typically less populated in international samples, I compute changes in receivables from the balance sheet variable in this example. Also note that when using `recch`, negative (positive) values for this variable represent increases (decreases) in receivables, consistent with the presentation in the cash flow statement.

lagged values of the variables. Because the reason for duplicate observations is setting- and database-specific, I do not discuss this issue here. As discussed in Section 2.6, keep in mind that it is important to understand why duplicates exist and which of the duplicate observations should be kept or dropped, instead of just blindly eliminating them from the sample.

```
duplicates report gvkey fyear
gsort gvkey fyear -datadate
duplicates drop gvkey fyear, force
tsset gvkey fyear
```

Next, we generate the input variables for the accrual model and drop observations that have missing values. Note that in this step, we lose one year of data from the sample. Because lagged data are required in the computation of some variables, the sample now runs from 2001 through 2021 instead of from 2000 through 2021.

```
gen lagta=l.at
gen tacc=(ibc-oancf)/lagta
gen drev=(d.sale-d.rect)/lagta
gen inverse_a=1/lagta
gen ppe=ppegt/lagta
drop if tacc==.
drop if drev==.
drop if ppe==.
```

The next step is to create a variable that captures a firm's two-digit SIC industry membership. We first transform the `sic` variable back from numeric format to string format. To ensure the resulting variable has four digits (e.g., change SIC code 100 into 0100), we use `tostring sic, format(%04.0f) replace`. Based on the updated variable, we then generate a new variable that is equal to the first two digits of `sic` using the `substring` command and use that to create a variable that contains a unique value for each group of two-digit industry and year with `egen`. Following common practice, we drop cases where fewer than 20 firms are available in an industry-year group. After dropping these cases, we reconstruct the variable that contains a unique value for each group of two-digit industry and year to ensure that the loop we will use runs from 1 to N with increments of 1.

```
tostring sic, format(%04.0f) replace
gen sic2=substr(sic,1,2)
egen sic2id=group(sic2 fyear)
sort sic2id
egen count=count(sic2id), by(sic2id)
drop if count<20
drop count sic2id
egen sic2id=group(sic2 fyear)
```

Because we have gone through all the data elimination steps and there are no further steps to be taken before we estimate the accrual model regressions, now is the time to correct the input variables for outliers (as in, e.g., Francis et al. [2005]). In this case, we winsorize all variables at the 1st and 99th percentiles of their distributions. Note that `inverse_assets` and `ppe` should only be winsorized at the 99th percentile, because these variables are bounded below by zero. Also note that it is essential to not take this step without carefully inspecting your data. It could be that, even after winsorization, the distribution is still extremely skewed on one or both sides (e.g., Cascino et al. [2023]).

```
sum tacc,d
replace tacc=r(p1) if tacc<r(p1)
replace tacc=r(p99) if tacc>r(p99) & tacc!=.
sum drev,d
replace drev=r(p1) if drev<r(p1)
replace drev=r(p99) if drev>r(p99) & drev!=.
sum inverse_a,d
replace inverse_a=r(p99) if inverse_a>r(p99) & inverse_a!=.
sum ppe,d
replace ppe=r(p99) if ppe>r(p99) & ppe!=.
```

In the next step, we generate two new numeric variables that will be filled with post-regression information: `dac` for the regression residuals (discretionary accruals) and optionally the variable `adjr2` that can be used to inspect the average explanatory power of the individual estimations. We also generate three new numeric variables that will contain the coefficient estimates for each group.

```
gen dac=.
gen r2a=.
gen b1=.
gen b2=.
gen b3=.
```

Now we create a forvalues loop that runs the accrual model estimations for the `k` combinations of industry and year (as stored in variable `sic2id`). The output of each estimation is stored in the variables that we generated above. In my example, the loop performs the estimations for 610 unique industry-year groups.

```
sum sic2id
local k=r(max)
forvalues i=1(1)`k'{
    qui reg tacc inverse_a drev ppe if sic2id==`i'
    qui predict res if sic2id==`i', res
    qui replace dac=res if sic2id==`i'
```

```
    qui replace r2a=e(r2_a) if sic2id==`i'
    qui replace b1=_b[inverse_a] if sic2id==`i'
    qui replace b2=_b[drev] if sic2id==`i'
    qui replace b3=_b[ppe] if sic2id==`i'
    qui drop res
    di `i' " / " `k'
}
```

We have now constructed a discretionary accrual variable by estimating a large number of regressions for industry-year groups for a large sample of firms. The discretionary accrual estimates are stored in variable `dac`. We can finish this exercise by inspecting the data and by saving the data to a new file. Inspection of the data and output is important because this can tell us more about the validity of the accrual-prediction model. For example, the average adjusted R^2 provides some information on the relevance of the predictor variables and the average coefficient estimates provide insights on whether the estimation results are consistent with expectations (e.g., the coefficient on the change in revenues should be positive, while that on PP&E should be negative). For my sample, I obtain the following statistics from the regression estimates. Note that `dac`, which is the measure of discretionary accruals, is unique for every observation since it is a regression residual (also, by construction, the mean of `dac` is zero). The average adjusted R^2 in this example equals about 0.10.

```
. tabstat r2a dac b1 b2 b3, stats(N mean p25 median p75) columns(statistics)

    Variable |         N      Mean       p25       p50       p75
-------------+--------------------------------------------------
         r2a |     58291  .1014665   .027475  .0695237  .1392917
         dac |     58291 -1.13e-13 -.0362742  .0095249  .0513281
          b1 |     58291 -.7609023 -1.438533 -.7182475 -.1805506
          b2 |     58291  .0103932 -.0391848  .0171269  .0750589
          b3 |     58291 -.0226127 -.0503221  -.023768   .006462
----------------------------------------------------------------
```

As we will see below with **Approach 2**, the estimation procedure of running a loop 610 and estimating a regression in each iteration of the loop is not very efficient. Fortunately, there are tools available that can do this (much) more quickly. Consider for example the program `rangestat` (available via SSC, `ssc inst rangestat`), which can be used for different purposes. With a specific implementation of this program, we obtain the industry-year specific coefficient estimates that we can use to compute exactly the same coefficient estimates as before. Instead of the 12 seconds it took on my computer to run the 610 loops, it took only 0.18 seconds to get the same estimates using `rangestat`:

```
. rangestat (reg) tacc inverse_a drev ppe, interval(fyear 0 .) by(sic2id)

. gen dac_rangestat=tacc-b_cons-(b_inverse_a*inverse_a)-(b_drev*drev)-
↪  (b_ppe*ppe)

. sum dac dac_rangestat

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
         dac |     58,291   -1.13e-13    .1193236   -.713851    .707473
dac_ranges~t |     58,291   -1.13e-13    .1193236   -.713851    .707473
```

Approach 2: One-step estimation using regdhfe

The previous illustration is useful because it does the trick of computing discretionary accrual measures and it illustrates how to compute regression residuals by group. However, it is not the most efficient way of doing so. The reason is that the estimation of the regressions parameters, and hence the residuals, by group can equally be done in a one-step estimation where the right-hand side variables of equation (A.1) are interacted with indicator variables for the groups (e.g., Chen et al. [2018]). Doing so allows for the regression intercept and slope coefficients to vary by group, similar to the by-group estimations. Accordingly, for the J industry-year groups we can also estimate the following linear regression model, where $\mathbb{1}_j$ refers to an indicator variable that is set equal to 1 for observations in industry-year group j, and 0 otherwise:

$$TA_{it} = \sum_{j=1}^{J} \mathbb{1}_j \left(\beta_{0j} + \beta_{1j}(1/ASSETS_{it-1}) + \beta_{2j}(\Delta REV_{it} - \Delta REC_{it}) + \beta_{3j} PPE_{it} \right) + \varepsilon_{it} \quad \text{(A.2)}$$

When we estimate this equation with the interaction terms in one step, the residuals in ε_{it} are equivalent to those computed from the by-group estimation illustrated in **Approach 1**. One way to perform this estimation is by creating new variables for the interactions with the right-hand side variables of equation (A.1), but this is very inefficient. For example, with my 610 industry-year groups we would have to create and include 2440 separate variables ($1 + 3$ for each group for the separate intercept and three slope coefficients). Fortunately, this is not necessary because the objective is to compute the residuals, not the parameters of the model. We can therefore rely on `reghdfe` (see Section 4.6), which can absorb the indicator variables and their interactions with the right-hand variables of equation (A.1):

```
. reghdfe tacc, absorb(sic2id##c.(inverse_a drev ppe)) resid noconst
(MWFE estimator converged in 1 iterations)
```

```
HDFE Linear regression                            Number of obs   =     58,291
Absorbing 1 HDFE group                            F(   0,  55851) =          .
                                                  Prob > F        =          .
                                                  R-squared       =     0.1834
                                                  Adj R-squared   =     0.1477
                                                  Within R-sq.    =     0.0000
                                                  Root MSE        =     0.1219

Absorbed degrees of freedom:
-----------------------------------------------------------+
        Absorbed FE | Categories  - Redundant  = Num. Coefs |
--------------------+--------------------------------------|
              sic2id |       610           0         610     |
 sic2id#c.inverse_a |       610           0         610    ?|
      sic2id#c.drev |       610           0         610    ?|
       sic2id#c.ppe |       610           0         610    ?|
-----------------------------------------------------------+
? = number of redundant parameters may be higher
```

This output might appear a bit strange at first sight, because we do not see any coefficient estimates and standard errors. But this is because we told `reghdfe` to only adjust the `tacc` variable for (i.e., "absorb") the effects of all the indicator variables, accrual model variables, and their interactions, and to not even estimate the coefficient for the (global) intercept since we also omitted that from equation (A.2). The latter was specified in the option `noconst`, although it is important to note that doing so does not have any effect on the estimated residuals (i.e., we could have omitted this option to get the same results).

The most important action is in the `absorb()` option, which contains `sic2id##c.()`. What does this all mean? Recall that `sic2id` is the numeric variable that contains the observation's group (industry-year) number, while `inverse_a`, `drev`, and `ppe` are the continuous variables from equation (A.1). By specifying `sic2id##c.()` we tell the program to absorb all indicator variables based on the `sic2id` group variable and their interactions with the continuous (hence the prefix `c.`) variables. Finally, the `resid` option is required to be able to use the post-estimation command `predict`. As we can see from the output below, this procedure using `reghdfe` gives us exactly the same residuals as we got from **Approach 1** in which we estimated the regressions separately by group. Together with the `reghdfe` estimation, it took only 0.06 seconds on my computer to create the discretionary accrual variable (compared to 12 seconds with the loop):

```
. predict dac_reghdfe, res
```

```
. sum dac dac_reghdfe

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
         dac |     58,291   -1.13e-13    .1193236   -.713851    .707473
 dac_reghdfe |     58,291    5.91e-19    .1193236   -.713851    .707473

.          pwcorr dac dac_reghdfe

             |      dac dac_re~e
-------------+------------------
         dac |   1.0000
 dac_reghdfe |   1.0000   1.0000
```

As you can see, the means of the two variables are not completely identical. This is because we created variable `dac` using `gen dac=.` before, which stores this variable as a floating point numeric variable. Instead, use of the `predict` command after `regdhfe ..., resid` causes the new variable that contains the regression residuals to automatically be stored using double precision (more precise). However, these differences are so extremely small that they are very unlikely to affect inferences. For example, the correlation between the two variables remains 1.0 despite the difference in storage precision.

Also note that this approach can easily be adapted to test whether discretionary accruals are higher or lower for some subset of observations by including an indicator variable for these observations in a one-step estimation. For example, Chen et al. [2018] and Leone et al. [2019] use such a one-step estimation with a Big-N indicator variable as the test variable.[4] In Schafhäutle and Veenman [2023], we examine whether firms that meet or beat earnings expectations are associated with higher discretionary accruals using such a one-step estimation. Assuming the partitioning variable used to test some hypothesis is called `part`, we can use `reghdfe` as follows to test whether observations for which `part==1` have higher discretionary accruals then those for which `part==0`. In my example, this indicator variable is completely randomly assigned to observations, so obviously we do not find any difference in average discretionary accruals across the groups:

[4] As a side-note, besides the valid empirical concerns related to these types of estimations, it is important to think of the conceptual logic behind such an analysis. For example, one might question whether regressing accruals on determinant variables and a Big-N auditor indicator for a panel of data with multiple years of data per firm is sensible, since it might not be reasonable to expect that clients of Big-N auditors have persistently high or low discretionary accruals during the full sample period. The reason is that we should expect income-increasing or income-decreasing accruals used for earnings management purposes to reverse in the next period. Instead, it might be more reasonable to examine whether the clients of Big-N auditors have higher or lower discretionary accruals in specific periods in which they have more likely engaged in earnings management. Thus, it is essential that you understand exactly what it is that you are trying to measure before you implement these methods.

```
. set seed 1234

. gen part=0

. replace part=1 if runiform()<.25
(14,516 real changes made)

. reghdfe tacc part, absorb(sic2id##c.(inverse_a drev ppe)) resid noconst
(MWFE estimator converged in 1 iterations)

HDFE Linear regression                            Number of obs   =     58,291
Absorbing 1 HDFE group                            F(   1,  55850) =       1.25
                                                  Prob > F        =     0.2630
                                                  R-squared       =     0.1834
                                                  Adj R-squared   =     0.1478
                                                  Within R-sq.    =     0.0000
                                                  Root MSE        =     0.1219

------------------------------------------------------------------------------
        tacc | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
        part |  -.0013349   .0011926    -1.12   0.263    -.0036724    .0010027
------------------------------------------------------------------------------

Absorbed degrees of freedom:
-----------------------------------------------------------+
        Absorbed FE | Categories  - Redundant  = Num. Coefs |
--------------------+--------------------------------------|
             sic2id |       610           0         610     |
 sic2id#c.inverse_a |       610           0         610    ?|
      sic2id#c.drev |       610           0         610    ?|
       sic2id#c.ppe |       610           0         610    ?|
-----------------------------------------------------------+
? = number of redundant parameters may be higher
```

A.3 Estimating Dechow-Dichev's accrual quality measure

This chapter describes the calculation of another variable that is frequently use to measure constructs related to financial reporting quality. Similar concerns and warnings apply as with the computation of discretionary accruals described in chapter A.2. These measures can be very imprecise and do not necessarily capture the underlying construct of financial reporting quality. Sometimes this noise is correlated with variables of interest, which leads to biased inferences (see, e.g., Hribar and Nichols [2007], Owens et al. [2017], and Cascino et al. [2023]). Therefore, please only consider the below as an illustration of how to implement specific procedures in Stata, rather than the right way to measure a financial reporting quality construct.

The estimation of accrual estimation errors following Dechow and Dichev [2002] and McNichols [2002] consists of two parts. The first part, the estimation of regression residuals, closely follows the estimation of discretionary accruals in chapter A.2. The only difference is the regression model that is estimated (and note that all variables are scaled by the average of total assets between year *t-1* and *t*):

$$WCA_{it} = \beta_0 + \beta_1 CFO_{it-1} + \beta_2 CFO_{it} + \beta_3 CFO_{it+1} + \varepsilon_{it} \tag{A.3}$$

The second part consists of estimating the standard deviation of residuals over time. While Dechow and Dichev [2002] estimate this standard deviation over 5 years, I will follow Ashbaugh-Skaife et al. [2008] and Veenman [2012] here and use a less restrictive period of 3–5 years. The input data are the same as in chapter A.2. The specific input variables used here are as follows:

Variable	Description
gvkey	Compustat company identifier
fyear	Fiscal year
datadate	Fiscal year end date
fic	Foreign incorporation code
at	Total assets
ibc	Income before extraordinary items (from cash flow statement)
oancf	Cash flow from operations
dpc	Depreciation (from cash flow statement)
sale	Sales revenue
ppegt	Gross property plant and equipment
sic	SIC industry code
sich	SIC industry code (historical)

Open the do-file editor, start a clean do-file, and save the empty do-file to a location and name of preference. Use this do-file to store all the programming code below. Also ensure that the `global` macro `path` refers to the location of the project folder (see Section 2.15). Generate

input variables and estimate the industry-year regressions for the Dechow and Dichev [2002] model in the same way as for the accrual model in chapter A.2.[5]

```
use "$path\InFiles\ccm_annual.dta", clear
destring gvkey, replace
ren lpermno permno
drop if at==. | at<=0
keep if fic=="USA"
destring sic, replace
replace sic=sich if sich!=.
drop if sic==.
drop if sic>5999 & sic<7000

// This requires more careful inspection:
duplicates report gvkey fyear
gsort gvkey fyear -datadate
duplicates drop gvkey fyear, force
tsset gvkey fyear

// Compute variables:
gen avta=(l.at+at)/2
gen cfom1=l.oancf/avta
gen cfo=oancf/avta
gen cfop1=f.oancf/avta
gen wca=(ibc-oancf+dpc)/avta
gen dsal=(d.sale)/avta
gen ppeg=ppegt/avta
gen sales=sale/avta
drop if wca==.
drop if cfom1==.
drop if cfo==.
drop if cfop1==.
drop if dsal==.
drop if ppeg==.
drop if sales==.

tostring sic, format(%04.0f) replace
gen sic2=substr(sic,1,2)
egen sic2id=group(sic2 fyear)
sort sic2id
egen count=count(sic2id), by(sic2id)
drop if count<20
drop count sic2id
egen sic2id=group(sic2 fyear)

sum wca,d
replace wca=r(p1) if wca<r(p1)
replace wca=r(p99) if wca>r(p99) & wca!=.
sum cfom1,d
replace cfom1=r(p1) if cfom1<r(p1)
```

[5] As before, the by-group regressions can be done more efficiently using a one-step estimation with `reghdfe`.

```
replace cfom1=r(p99) if cfom1>r(p99) & cfom1!=.
sum cfo,d
replace cfo=r(p1) if cfo<r(p1)
replace cfo=r(p99) if cfo>r(p99) & cfo!=.
sum cfop1,d
replace cfop1=r(p1) if cfop1<r(p1)
replace cfop1=r(p99) if cfop1>r(p99) & cfop1!=.
sum dsal,d
replace dsal=r(p1) if dsal<r(p1)
replace dsal=r(p99) if dsal>r(p99) & dsal!=.
sum ppeg,d
replace ppeg=r(p1) if ppeg<r(p1)
replace ppeg=r(p99) if ppeg>r(p99) & ppeg!=.
sum sales,d
replace sales=r(p1) if sales<r(p1)
replace sales=r(p99) if sales>r(p99) & sales!=.

gen residual=.
gen adjr2=.
gen b0=.
gen b1=.
gen b2=.
gen b3=.
gen b4=.
gen b5=.

// Step 1: compute residuals by industry-year group:
sum sic2id
local k=r(max)
forvalues i=1(1)`k'{
    qui reg wca cfom1 cfo cfop1 dsal ppeg if sic2id==`i'
    qui predict res if sic2id==`i', res
    qui replace residual=res if sic2id==`i'
    qui replace adjr2=e(r2_a) if sic2id==`i'
    qui replace b0=_b[_cons] if sic2id==`i'
    qui replace b1=_b[cfom1] if sic2id==`i'
    qui replace b2=_b[cfo] if sic2id==`i'
    qui replace b3=_b[cfop1] if sic2id==`i'
    qui replace b4=_b[dsal] if sic2id==`i'
    qui replace b5=_b[ppeg] if sic2id==`i'
    qui drop res
    di `i' " / " `k'
}
sort gvkey fyear
tabstat adjr2 residual b0 b1 b2 b3 b4 b5, stats(N mean p25 median p75)
↪  columns(statistics)
save "$path\OutFiles\ccm_annual_dd.dta", replace
```

For my sample, I obtain the following statistics from the regression estimates. Note that `residual`, which is the measure of unexplained accruals, is unique for every observation since

it is a regression residual (also, by construction, the mean of `residual` should be zero). The average adjusted R^2 in this example equals about 0.24. Have a look at the average values of coefficients `b1`, `b2`, and `b3` and compare them with Dechow and Dichev [2002]. Are the signs of the coefficients consistent with their results, and what would be expected based on your understanding of the role of accruals in accounting?

```
. tabstat adjr2 residual b0 b1 b2 b3 b4 b5, stats(N mean p25 median p75)
↪  columns(statistics)

    Variable |         N      Mean       p25       p50       p75
-------------+--------------------------------------------------
       adjr2 |     50354  .2431014   .129068  .1914054  .3091589
    residual |     50354 -1.35e-11 -.0258081  .0070188   .039355
          b0 |     50354 -.0367597 -.0563055 -.0367512 -.0129553
          b1 |     50354   .232314  .1314869  .2239075  .3178582
          b2 |     50354 -.3837566  -.524746 -.3494783 -.2337921
          b3 |     50354  .2057558  .1267915  .1923093  .2843632
          b4 |     50354  .0768101  .0287723   .084711  .1237326
          b5 |     50354  .0167706 -.0029726  .0176844  .0377658
----------------------------------------------------------------
```

Now open the data file that was created in the previous steps and generate two new numeric variables: `sdresidual` which will contain the 3–5 year standard deviation of the Dechow and Dichev [2002] residuals and optionally the `countobs` variable in which we will store the number of observations used to calculate the standard deviation (minimum: three, maximum: five).

```
use "$path\OutFiles\ccm_annual_dd.dta", clear
gen sdresidual=.
gen countobs=.
```

Next, calculate the standard deviation of residuals for every firm-year that has at least three years (current year plus previous two years) of residual data, for the maximum of five years (current year plus previous four years). This requires a loop *within* a loop, as we should first determine for each firm how many years of data are available, and then run the main loop over those firm-years (warning: this might take a while!):

```
sort gvkey fyear
egen nr=group(fyear)
egen minnr=min(nr), by(gvkey)
replace nr=nr-min+1
egen firmid=group(gvkey)
sum firmid
local m=r(max)
forvalues i=1(1)`m'{
    qui sum nr if firmid==`i'
```

```
    local n=r(min)+2
    local p=r(max)
    if `p'>2 {
        forvalues j=`n'(1)`p'{
            qui gen hulpvar=1 if firmid==`i' & nr<=`j' & nr>`j'-5
            qui sum residual if hulpvar==1
            qui replace sdresidual=r(sd) if firmid==`i' & nr==`j'
            qui replace countobs=r(N) if firmid==`i' & nr==`j'
            qui drop hulpvar
            di `i' " / " `j' " / " `m'
        }
    }
}
replace sdresidual=. if countobs<3
replace countobs=. if countobs<3
```

Now we have create the Dechow and Dichev [2002] accrual estimation error variable for a large sample of firms. Finish by inspecting the data and saving the generated data to a new file.

```
sum fyear sdresidual countobs
tabstat countobs, by(countobs) stats(N)
keep gvkey fyear sdresidual
save "$path\OutFiles\ccm_annual_dd_sdresidual.dta", replace
```

For my sample, I obtain the following statistics. Variable `sdresidual` is the variable of interest. The larger the value of this variable, the lower the quality of accruals according to the reasoning in Dechow and Dichev [2002]. The measure is available only for 38,356 out of the 50,534 observations due to the requirement of at least 3 observations in the calculations above.

```
    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
       fyear |     50,354    2009.569    5.836116      2001       2020
  sdresidual |     38,356    .0613732    .0593479  .0001077    .519264
    countobs |     38,356    4.619043    .7047051         3          5
```

As in many situations when using Stata, there is more than one way to achieve an objective. Although the code used above did the trick for us of calculating the rolling-window standard deviations of accrual model residuals, the procedure was quite inefficient because we had to use a loop within a loop. An alternative approach, which produces the same results but which is much faster, is the following.

First, we can replace the regression estimation by industry-year group with a one-step estimation using `reghdfe`, as explained in Section A.2:

```
reghdfe wca, absorb(sic2id##c.(cfom1 cfo cfop1 dsal ppeg)) resid noconst
predict residual_reghdfe, res
```

This much faster procedure provides us with the same estimates of the residuals:

```
. sum residual_reghdfe residual

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
residual_r~e |     50,354   -5.45e-19    .0969238  -.7108377   .5537614
    residual |     50,354   -1.35e-11    .0969238  -.7108377   .5537614
```

Second, we can use features of the `egen` function, which can help us compute the standard deviation of a variable for specific groups of observations. To achieve this, we should first transform the data in such a way that the (maximum) five residuals for *each* firm-year are contained in separate rows. Specifically, for year 2020 of firm A we need five separate rows for the residuals of 2016–2020, while for 2019 of firm A we need we need five separate rows for the residuals of 2015–2019. We can achieve this objective using the `reshape` command:

```
sort gvkey fyear
gen n=_n
gen res_1=residual_reghdfe
gen res_2=l.residual_reghdfe
gen res_3=l2.residual_reghdfe
gen res_4=l3.residual_reghdfe
gen res_5=l4.residual_reghdfe
reshape long res_, i(n) j(j)
egen sdresidual_alt=sd(res_), by(n)
egen countobs_alt=count(res_), by(n)
replace sdresidual_alt=. if countobs_alt<3
keep if j==1
```

Inspection of the variable computed this way with the alternative procedure used earlier confirms that indeed these approaches lead to the exact same results:

```
. sum sdresidual sdresidual_alt

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
  sdresidual |     38,356    .0613732    .0593479   .0001077    .519264
sdresidual~t |     38,356    .0613732    .0593479   .0001077    .519264

. pwcorr sdresidual sdresidual_alt

             | sdresi~l sdresi~t
-------------+------------------
  sdresidual |   1.0000
sdresidual~t |   1.0000   1.0000
```

A.4 Obtaining regression output without a loop

In chapters A.2 and A.3 we used a `forvalues` loop to perform the estimation of regressions for many different groups of observations. The good news is that this automated process saves us a lot of work. The bad news is that sometimes we have to wait for a long time for the loops to finish when the sample is large. For example, in Gassen et al. [2020] we computed the regression R^2 for a total of more than half a million regressions. Luckily, with regressions that include only 1 or 2 independent variables we can also compute the relevant regression statistics using simple formulas without having to use the `reg` command in a loop.

Take for instance a simple CAPM regression of a firm's daily stock returns on the daily market index, and assume this regression is performed for every firm-year ($i\tau$) in a sample:

$$R_{it} = \alpha_{i\tau} + \beta_{i\tau} R_{mt} + \varepsilon_{it} \tag{A.4}$$

For this regression, we know that $\beta_{i\tau} = \frac{Cov[R_{it}, R_{mt}]}{Var[R_{mt}]}$ and $R^2_{i\tau} = \frac{\beta^2_{i\tau} Var[R_{mt}]}{Var[R_{it}]}$, and we can use these equations to tell Stata how to compute the regression statistics for each specific subset of a sample using the `egen` function. We can do the same for a regression that includes two independent variables, as in Gassen et al. [2020]:

$$R_{it} = \alpha_{i\tau} + \beta_{1,i\tau} R_{mt} + \beta_{2,i\tau} R_{mt-1} + \varepsilon_{it} \tag{A.5}$$

The following code does the trick of calculating the regression coefficients and explanatory powers for a large set of firm-years. The input data I used is a file with US daily stock returns from CRSP for the period 2000–2021 (153,612 unique firm-years; 36,025,138 daily stock return observations). A lagged daily market return variable (R_{mt}) has been added to this dataset. To ensure a meaningful estimation, we first clean the data and eliminate firm-year observations with fewer than 100 daily return observations. I also eliminate those firm-years for which all daily returns are equal to zero:[6]

```
use "$path\InFiles\crspdaily.dta", clear
drop if ret==.
drop if vwretd==.
```

[6]Note that even though the program described below is relatively fast for having to compute the coefficients and R-squareds for 153,612 unique firm-years, its speed can be increased significantly when the `egen` functions are replaced by `gegen` functions, which are available through the `gtools` package. On my computer, running the code below took 107 seconds. After replacing all instances of `egen` with `gegen`, running the code took 88 seconds. For more information on `gtools`, see https://gtools.readthedocs.io/en/latest/.

```
gen year=year(date)
egen id=group(permno year)
egen count=count(ret), by(id)
sum count
drop if count<100
gen nonzero=1 if ret!=0
egen countnz=count(nonzero), by(id)
drop if countnz==0
drop nonzero countnz
```

Next, we start by estimating the relevant firm-year-specific parameters for the one-variable model. Because the `egen` function can easily compute means and standard deviations by groups, but not covariances, we need to use the formulas for covariance calculation in combination with the `egen` function.

```
egen meanx=mean(vwretd), by(id)
egen meany=mean(ret), by(id)
gen diffx=vwretd-meanx
gen diffxx=diffx*diffx
gen diffy=ret-meany
gen diffyy=diffy*diffy
gen diffxy=diffx*diffy
egen sumdiffxx=sum(diffxx), by(id)
qui replace sumdiffxx=sumdiffxx*(1/(count-1))
qui egen sumdiffxy=sum(diffxy), by(id)
qui replace sumdiffxy=sumdiffxy*(1/(count-1))
gen beta=sumdiffxy/sumdiffxx
gen alpha=meany-beta*meanx
gen res=ret-alpha-beta*vwretd
gen resres=res*res
egen tss=sum(diffyy), by(id)
egen rss=sum(resres), by(id)
gen r2=1-(rss/tss)
sum beta r2
replace r2=.00001 if r2<.00001
keep permno date year id beta r2
```

We can similarly compute the parameters for the two-variable model as follows:

```
drop if vwretdm1==.
egen count=count(ret), by(id)
egen meanx1=mean(vwretd), by(id)
egen meanx2=mean(vwretdm1), by(id)
egen meany=mean(ret), by(id)
gen diffx1=vwretd-meanx1
gen diffx2=vwretdm1-meanx2
gen diffx1x1=diffx1*diffx1
gen diffx1x2=diffx1*diffx2
gen diffx2x2=diffx2*diffx2
gen diffy=ret-meany
```

```
gen diffyy=diffy*diffy
gen diffx1y=diffx1*diffy
gen diffx2y=diffx2*diffy
egen sumdiffx2x2=sum(diffx2x2), by(id)
egen sumdiffx1y=sum(diffx1y), by(id)
egen sumdiffx1x2=sum(diffx1x2), by(id)
egen sumdiffx2y=sum(diffx2y), by(id)
egen sumdiffx1x1=sum(diffx1x1), by(id)
gen beta1=((sumdiffx2x2*sumdiffx1y)-(sumdiffx1x2*sumdiffx2y))/
↪  ((sumdiffx1x1*sumdiffx2x2)-(sumdiffx1x2*sumdiffx1x2))
gen beta2=((sumdiffx1x1*sumdiffx2y)-(sumdiffx1x2*sumdiffx1y))/
↪  ((sumdiffx1x1*sumdiffx2x2)-(sumdiffx1x2*sumdiffx1x2))
gen alpha=meany-beta1*meanx1-beta2*meanx2
gen res=ret-alpha-beta1*vwretd-beta2*vwretdm1
egen idvol=sd(res), by(id)
gen resres=res*res
egen rss=sum(resres),by(id)
egen tss=sum(diffyy),by(id)
gen r22=1-(rss/tss)
replace r22=.00001 if r22<.00001
```

Finally, we can transform the daily return dataset into a firm-year dataset and retain the firm-year specific variables we just created:

```
duplicates drop permno year, force
keep permno year beta* r2*
save "$path\OutFiles\firmyears_r2.dta", replace
```

Using this code, and exploiting the ability to compute regression statistics for models with only one or two independent variables, we have been able to compute statistics for many regressions *without* having to use the regression command in a loop. Because Stata can be very slow when having large datasets in its memory (as is the case here), this saves us a lot of time.

Important: Note that this code works properly *only* if we ensure there are no missing variables for any of the input variables. This is why we initially dropped missing values for `ret` and `vwretd`, and for the two-variable model also dropped missing values for `vwretdm1`. In addition, the `count` variable had to be recalculated.

A.5 Transforming data from annual to monthly

Section 3.2 discussed the `reshape` command. Another useful command for the purpose of changing the unit of our data, is `expand`. Let's say we have a firm-year dataset with annual accounting data that we would like to attach to a firm-month dataset for an asset pricing study. For example, it is common in asset pricing studies to assume that accounting data are available to the market within four months after the fiscal year end. Hence, for a firm with a fiscal year end in December, we want to match the year t accounting data to the firm-month data for the months May in year *t+1* through April in year *t+2*. To achieve this, we should transform the firm-*year* dataset into a firm-*month* dataset. Because this means the dataset will have to become 12 times as large, we can simply use the `expand` command and create 11 duplicates for each observation. Let's say the accounting variable we want to attach is the book value of common equity (`ceq`). To make it easier to sort the data back to its original order, we also create the `n` variable:

```
use "$path\InFiles\funda_ceq_nld.dta", clear
destring gvkey, replace
drop if ceq==.
duplicates report gvkey datadate

. sum ceq

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
         ceq |      4,447    4976.274     95854.2  -7395.969    4553200

. gen n=_n

. expand(12)
(48,917 observations created)

. sum ceq

    Variable |        Obs        Mean    Std. dev.       Min        Max
-------------+---------------------------------------------------------
         ceq |     53,364    4976.274    95844.32  -7395.969    4553200
```

Using the variable `datadate` as the variable that contains the fiscal year end date for the firm, and the following additional set of commands, we can now create a new dataset based on unique firm-month combinations that reflect the `ceq` data for the most recent fiscal year:

```
bysort n: gen m_=_n
gen year=year(datadate)
```

```
gen month=month(datadate)
replace month=month+m_+4
replace year=year+2 if month>24
replace month=month-24 if month>24
replace year=year+1 if month>12
replace month=month-12 if month>12
```

This procedure may induce some duplicate observations when firms change their fiscal year-end date (e.g., in my data one company had its year-end in both June and December of 2020 because it changed its fiscal-year end month from June to December in that year). Assuming we want to keep the most recent fiscal year data that is available in a month, the following lines of code help achieve this:

```
duplicates report gvkey year month
gsort gvkey year month -datadate
duplicates drop gvkey year month, force
keep gvkey year month ceq
save "$path\OutFiles\ceq_monthly_nld.dta", replace
```

Note that by creating this dataset with forward-looking firm-months, we might actually create firm-months that do not exist because a firm may go bankrupt or is acquired. This is not a problem, because these non-existing firm-months will be eliminated from our sample when merging the data to our main firm-month dataset that contains, for example, monthly stock returns. Let's assume this main dataset has the same firm/security identifier `gvkey`, unique variables for calendar years and months, and a monthly return variable `ret` (note that we will construct this file in chapter A.6, which is why it is located in my "OutFiles" folder). We can now merge our accounting data to this file as follows:

```
use "$path\OutFiles\monthlyret_nld.dta", clear
sum year month ret
joinby gvkey year month using "$path\OutFiles\ceq_monthly_nld.dta",
↪  unmatched(master)
drop _merge
sum gvkey retm lagmv ceq
save "$path\OutFiles\monthlyret_nld_ceq.dta", replace
```

A.6 Computing stock returns from Compustat Global data

When using US samples, the use of stock return data is easy given the existence of the CRSP database. For international samples, however, this is less straightforward. One option is to calculate daily or monthly returns from Datastream's return index (`RI`). Another option is to use Compustat Global data and derive return data from there. Given the latter data are easy to obtain from WRDS, but computing returns is relatively complex, I will illustrate how to calculate daily and monthly stock returns from Compustat Global data. I will do this for a sample of Dutch listed companies (country code `NLD`) and price data over the period 2003–2017 (to be able to compute returns in January 2003 as well, I also downloaded price data for December 2002).

The first step is to compute daily stock returns from the data in the Compustat Global Security Daily database. To achieve this, we should take into account both dividend distributions and stock splits. Because one firm may have multiple securities traded on an exchange, we should first identify and eliminate duplicate firm-day observations by retaining only the primary listing of each firm (identified by variable `prirow`). I also apply some additional filters, such as requiring the stock to be traded on the country's primary exchange (in this case, Euronext Amsterdam, which is indicated by exchange code 104) and requiring the stock to be quoted in euro currency:

```
use "$path\InFiles\compdaily_nld.dta", clear
keep if fic=="NLD"
keep if exchg==104
keep if curcdd=="EUR"
destring gvkey, replace
duplicates report gvkey datadate
drop if iid==""
drop if prirow==""
keep if iid==prirow
drop iid prirow
duplicates report gvkey datadate
```

We should also eliminate observations that are non-trading days. For some stocks, Compustat lists their prices on days that are not actual trading days, while for most others it appears that it does not. To ensure we compute returns between consecutive trading days for every firm, we therefore need to eliminate all non-trading days on the exchange. The following block of code helps in this respect, and eliminates all days on which the majority of firms are inactive (note that the choice for `ratio<.1` is somewhat arbitrary, so carefully check the distribution of

ratio).

```
gen year=year(datadate)
gen month=month(datadate)
sort gvkey year month datadate
// Count number of active firms per day
gen nonzero=1 if prccd!=prccd[_n-1] & gvkey==gvkey[_n-1]
egen count=count(nonzero), by(datadate)
// Count number of firms available per month
sort year month gvkey
egen firmmonth=group(year month gvkey)
egen min=min(firmmonth), by(year month)
replace firmmonth=firmmonth+1-min
egen totalfirms=max(firmmonth), by(year month)
// Compare counts:
gen ratio=count/totalfirms
sum ratio,d
sort ratio datadate
drop if ratio<.1
drop firmmonth min totalfirms count nonzero ratio
sort gvkey datadate
```

Because the data contain information about stock splits only for the days on which a stock split has taken place, and not the cumulative split factor, we first need to construct this factor from the data. To achieve this, note that the objective is to create a variable that equals 1 for the most recent observation in the data. For example, if a firm has two years of data and split its stock 2:1 at the end of the first year, we want the split factor to have a value of 2 for the first year, and a value of 1 for the second year. We can then divide the stock price by this split factor and create split-adjusted daily prices. We can do the same for the variable that captures the dividends per share.

```
gen splitday=1 if split!=.
replace split=1 if split==.
replace split=split[_n-1]*split[_n] if gvkey[_n]==gvkey[_n-1]
sort gvkey datadate
replace split=split[_n+1] if splitday==1
drop splitday
gen prccd_adj=prccd/split
gen div_adj=div/split
save "$path\OutFiles\compdaily_nld_splitadj.dta", replace
```

Now that the data are split-adjusted, we can use the daily prices and dividends to compute daily returns. Before doing this, we should inform Stata about the panel structure of the data using the `tsset` function:

```
use "$path\OutFiles\compdaily_nld_splitadj.dta", clear
egen firmid=group(gvkey)
```

```
egen timeid=group(datadate)
tsset firmid timeid
replace div_adj=0 if div_adj==.
gen retdaily=(d.prccd_adj+div_adj)/l.prccd_adj
sum retdaily, d
save "$path\OutFiles\dailyret_nld.dta", replace
```

At this point, it is probably wise to inspect the data and apply some filters to correct for potential data errors. For example, Griffin et al. [2010] provide a rule that helps correct for potential data errors. They filter their daily return data (see their Appendix B.3) using the following rule: if the current day (r_t) or lagged day return (r_{t-1}) exceeds 100% *and* $(1 + r_{t-1})(1 + r_t) - 1 < 0.2$, both r_t and r_{t-1} are set to missing. In addition, any daily return is set to missing when it exceeds 200%. On top of this rule, I filter the return data by dropping those observations for which the minimum closing stock price in the previous month was below one euro (based on the stock price *before* the split-adjustment).

```
use "$path\OutFiles\dailyret_nld.dta", clear
egen minp=min(prccd), by(gvkey year month)
sort gvkey year month datadate
by gvkey year month: gen d=_n
gen lagminp=minp[_n-1] if d==1 & gvkey==gvkey[_n-1]
egen lagminp2=max(lagminp), by(gvkey year month)
drop if lagminp2<1
sum retdaily, d
tsset
gen filter=1 if retdaily>1 & ((1+l.retdaily)*(1+retdaily)-1)<.2
replace filter=1 if l.retdaily>1 & ((1+l.retdaily)*(1+retdaily)-1)<.2
sum retdaily l.retdaily if filter==1
replace retdaily=. if filter==1
replace retdaily=. if f.filter==1
replace retdaily=. if retdaily>2
keep gvkey year month datadate retdaily monthend prccd* div* cshoc
sum retdaily, d
save "$path\OutFiles\dailyret_nld_clean.dta", replace
```

Next, we can compute monthly returns by retaining the final day of the month. This final day of the month is identified by Compustat's `monthend` variable, which indicates the last day for which an observation is available for the specific firm-month. Because we already eliminated some days (probably including some observations for which `monthend==1`), however, we need to adjust this variable first. In addition, I add additional variables for the current and previous month market value of equity. These variables are useful in situations such as those explained in chapter A.7.

```
use "$path\OutFiles\dailyret_nld_clean.dta", clear
sort gvkey year month datadate
```

```
egen maxd=max(datadate), by(gvkey year month)
replace monthend=1 if datadate==maxd
egen divsum=sum(div_adj), by(gvkey year month)
keep if monthend==1
egen ym=group(year month)
tsset gvkey ym
gen retm=(d.prccd_adj+divsum)/l.prccd_adj
sum retm, d
gen mv=prccd*cshoc
gen lagmv=l.mv
drop if retm==.
drop if mv==.
drop if lagmv==.
keep if year(datadate)>2000 & year(datadate)<2022
gvkey year month retm mv lagmv
sort gvkey year month
save "$path\OutFiles\monthlyret_nld.dta", replace
```

Of course, it is also important that we inspect the outcome of these procedures. For example, for the monthly return data I observe the following distribution of the monthly return variable:

```
.         sum retm, d

                            retm
-------------------------------------------------------------
      Percentiles      Smallest
 1%    -.2951205      -.9792603
 5%    -.1469194      -.9601626
10%    -.0964174      -.9325301       Obs               30,768
25%     -.037351      -.9176471       Sum of wgt.       30,768

50%     .0031142                      Mean           .0059527
                        Largest       Std. dev.      .1085518
75%     .0466843       1.664757
90%     .1075459       1.834146       Variance       .0117835
95%     .1597111       1.890909       Skewness       1.373714
99%     .3216495       2.599727       Kurtosis       32.55586
```

Finally, note that this example of using Compustat Global data to compute returns was still relatively straightforward. The reason is that I used only one country and a period for which there was only one currency. Additional adjustments would have to be made to account for the switch to euro if an earlier time period was taken. In addition, some stocks I excluded are actually quoted in US dollars on Euronext Amsterdam. What should we do with those? Compute returns based on prices in US dollars, or convert the daily prices to euro prices first? What if a stock switches in currency over the sample period, while others do not? What if a country had more than one primary stock exchange? These are important questions that you

might encounter when working with international stock price data. What matters is that the specific institutional setting is well understood. As with all examples provided here, do not just copy and paste the code, but first try to understand your data and the research setting.

A.7 Computing Fama-French factor-model abnormal returns

This chapter illustrates how to calculate Fama and French [1993] three-factor model abnormal monthly returns in an international setting. We use rolling window regressions, where for every firm-month the three-factor model is estimated for the previous 60 firm-months (5 years). A minimum of 18 prior months is required for the estimation. The monthly stock return data in the example are for Dutch listed companies as calculated from the Compustat Global Security Daily database (see chapter A.6). In addition, we need data on the annual book value of equity from the Compustat Global Fundamentals file for the calculation of the monthly book-to-market ratio. Accounting data are assumed to be available four months after a company's fiscal year end, and are attached to the monthly stock return data following the steps explained in chapter A.5. Here we simply assume that a firm-month file with the following variables is available:

Variable	Description
gvkey	Compustat company identifier
year	calendar year of observation
month	calendar month of observation
retm	monthly stock return
mv	market value of equity at month-end
lagmv	market value of equity at the end of the previous month
ceq	book value of equity from the most recent fiscal year

The three factors in the three-factor model are the market return (R^{MKT}), size portfolio return (SMB), and book-to-market portfolio return (HML). For simplicity, we ignore the adjustments for the risk-free rate here:

$$R_{it} = \beta_0 + \beta_1 R_t^{MKT} + \beta_2 SMB_t + \beta_3 HML_t + \varepsilon_{it} \tag{A.6}$$

We can create these factors based on the input data available in the file that was created in the previous step. First ensure the book equity and market value variables are denoted in the same units before generating the book-to-market ratio. Next, for every calendar month, determine the ranks of firm size and book-to-market as a first step to generating the Fama-French factors. Here, we choose to organize the data into quintile (5) portfolios every month.[7]

```
use "$path\OutFiles\monthlyret_nld_ceq.dta", clear
drop if retm==.
drop if lagmv==.
drop if ceq==.
replace ceq=ceq*1000000
```

[7]Note that if the `gtools` package is installed in Stata, you do not have to run the loop to compute the quintile portfolios by year-month. Instead you can use the `gquantiles` function in combination with the `, by()` and `xtile` options. For more information on `gtools`, see https://gtools.readthedocs.io/en/latest/.

```
gen btm=ceq/lagmv
sum btm,d
gen mvq=.
gen btmq=.
sort gvkey year month
egen yearmonth=group(year month)
sum yearmonth
scalar maxy=r(max)
local k=maxy
forvalues i=1(1)`k'{
    qui xtile xmv=mv if yearmonth==`i', nq(5)
    qui replace mvq=xmv if yearmonth==`i'
    qui drop xmv
    qui xtile xbtm=btm if yearmonth==`i', nq(5)
    qui replace btmq=xbtm if yearmonth==`i'
    qui drop xbtm
    di `i' " / " `k'
}
```

Now create the SMB factor as the difference in average monthly returns of small firms (quintiles 1 and 2) minus the average monthly returns of large firms (quintiles 4 and 5). Similarly, create the HML factor as the difference in average returns between firms with high and low book-to-market.

```
gen ret12=ret if mvq==1 | mvq==2
gen ret45=ret if mvq==4 | mvq==5
egen ewret12=mean(ret12), by(yearmonth)
egen ewret45=mean(ret45), by(yearmonth)
gen smb=ewret12-ewret45
drop ret12 ret45 ewret12 ewret45
gen ret12=ret if btmq==1 | btmq==2
gen ret45=ret if btmq==4 | btmq==5
egen ewret12=mean(ret12), by(yearmonth)
egen ewret45=mean(ret45), by(yearmonth)
gen hml=ewret45-ewret12
drop ret12 ret45 ewret12 ewret45
sum smb hml
save "$path\OutFiles\monthlyret_factors1.dta", replace
```

Assume we need the market-return (R^{MKT}) to be a value-weighted instead of an equal-weighted average return. To this end, we can weight the individual firm returns by the lagged market values of equity each month. Careful, in some countries only a few companies dominate the market in terms of market capitalization. This implies that the weights can be very high for some firms. Carefully think about whether or not this is desirable.

```
use "$path\OutFiles\monthlyret_factors1.dta", clear
egen summv=sum(lagmv),by(year month)
```

```
gen weight=lagmv/summv
sum weight, d
gen wret=weight*retm
egen mktret=sum(wret), by(year month)
duplicates drop year month, force
sum mktret smb hml
save "$path\OutFiles\monthlyret_factors2.dta", replace
```

Now go back to the monthly stock return file and merge this with the Fama-French factors:

```
use "$path\OutFiles\monthlyret_nld_ceq.dta", clear
joinby year month using "$path\OutFiles\monthlyret_factors2.dta",
↪  unmatched(master)
drop _merge
sum retm mktret smb hml
```

As a quick check, we can estimate a pooled regression of monthly firm returns on the self-created factors to see if the correlations make sense. My results reveal the following:

```
.         reg retm mktret smb hml

      Source |       SS           df       MS      Number of obs   =    30,768
-------------+----------------------------------   F(3, 30764)     =   2241.55
       Model |  65.0322367         3  21.6774122   Prob > F        =    0.0000
    Residual |  297.510489    30,764  .009670735   R-squared       =    0.1794
-------------+----------------------------------   Adj R-squared   =    0.1793
       Total |  362.542725    30,767  .011783493   Root MSE        =    .09834

------------------------------------------------------------------------------
        retm | Coefficient  Std. err.      t    P>|t|     [95% conf. interval]
-------------+----------------------------------------------------------------
      mktret |   .8709698   .0110473    78.84   0.000     .8493165     .892623
         smb |   .4404377   .0204808    21.50   0.000     .4002945    .4805808
         hml |   .1515699   .0214182     7.08   0.000     .1095893    .1935505
       _cons |   .0048284     .00058     8.32   0.000     .0036915    .0059652
------------------------------------------------------------------------------
```

At this point, we can finally start with the real objective of this exercise, which is to a) compute these coefficients for each firm and months $[-60, -1]$ and b) use them to compute abnormal returns in event month 0. Remember that we estimate the model over a window of 60 months prior to the month of interest with a minimum of 18 observations. Therefore, we should eliminate all observations without a minimum of 18 prior months available:

```
keep gvkey year month retm mktret smb hml
sort gvkey year month
egen fid=group(gvkey)
by gvkey: gen nr=_n
gen nr2=nr-18
```

```
replace nr2=. if nr2<1
egen maxnr2=max(nr2),by(fid)
drop if maxnr2==.
drop fid maxnr2
egen fid=group(gvkey)
save "$path\OutFiles\monthlyret_nld_ceq2.dta", replace
```

Now we can start estimating the regressions and store the coefficients. Because for each firm the regression is estimated multiple times, and the number of times varies by firm, we will run a loop within a loop. This means we should use different indices for the loops (here: `i` and `j`). After having estimated the parameters of the three-factor model for every firm-month based on the previous 18–60 months, we can then use these parameters for the current month to compute abnormal returns as follows.

```
use "$path\OutFiles\monthlyret_nld_ceq2.dta", clear
gen beta0=.
gen beta1=.
gen beta2=.
gen beta3=.
gen countobs=.
sum fid
scalar maxp=r(max)
local p=maxp
forvalues i=1(1)`p'{
    qui sum nr2 if fid==`i'
    scalar maxn=r(max)
    local n=maxn
    forvalues j=1(1)`n'{
        qui reg retm mktret smb hml if fid==`i' & nr<=`j'+17 & nr>`j'-43
        qui replace beta0=_b[_cons] if fid==`i' & nr2==`j'
        qui replace beta1=_b[mktret] if fid==`i' & nr2==`j'
        qui replace beta2=_b[smb] if fid==`i' & nr2==`j'
        qui replace beta3=_b[hml] if fid==`i' & nr2==`j'
        qui replace countobs=e(N) if fid==`i' & nr2==`j'
        di `i' " /// " `p' " : " `j' " \\\ " `n'
    }
}
sum beta*
gen abret=ret-beta0-beta1*mktret-beta2*smb-beta3*hml
sum abret,d
save "$path\OutFiles\monthlyret_ff3.dta", replace
```

This example illustrated how to compute abnormal returns using the Fama-French three-factor model. Several simplifications were introduced. Therefore, when using this code make sure to look back at the actual steps to be taken when computing, for example, the portfolio returns. For US data, these factors are readily available. To a lesser extent, these are available for

other countries. Nonetheless, this example should be helpful for analyses of a similar kind. In addition, the code can be easily modified to other multi-factor models.

A.8 Estimating implied cost of equity capital

This chapter illustrates how we can estimate measures of the expected rate of return (implied cost of capital) based on a Residual Income Valuation model, following the work of, for example Claus and Thomas [2001] and Gebhardt et al. [2001]. In this example, the valuation is based on a three-year forecast horizon similar to equation (5) in Veenman [2013]:

$$V_0 = bv_0 + \frac{ri_1}{(1+r_e)} + \frac{ri_2}{(1+r_e)^2} + \frac{ri_3}{(1+r_e)^3} + \frac{ri_3(1+g)}{(r_e-g)(1+r_e)^3} \tag{A.7}$$

where bv_0 is the current book value of equity per share, r_e is the cost of equity capital, ri_t is residual income in period t—which is the difference between expected earnings per share (EPS) in period t and r_e times the beginning book value of equity per share bv_{t-1}, and g is the assumed rate of growth in residual income beyond the three-year forecast horizon. The implied cost of equity capital is the rate of r_e that results in the valuation (V_0) being equal to the current market price (P_0). Thus:

$$0 = -P_0 + bv_0 + \frac{ri_1}{(1+r_e)} + \frac{ri_2}{(1+r_e)^2} + \frac{ri_3}{(1+r_e)^3} + \frac{ri_3(1+g)}{(r_e-g)(1+r_e)^3} \tag{A.8}$$

where P_0, bv_0, ri_1, ri_2, ri_3, and g are known parameters. To simplify things in this illustration, I set the expected annual growth in residual income beyond the final forecasting year equal to one percent ($g = 0.01$).[8] The data used for this example are from a global sample of firms with one-year, two-year, and three-year ahead EPS forecasts in the intersection of Compustat Global and IBES (2011–2022). Firm-years are included only if the book value of equity and future earnings variables are positive. The input variables in this example are:

Variable	Description
gvkey	Company identifier
year	Year
p	Current stock price
eps1	One-year ahead EPS forecast
eps2	Two-year ahead EPS forecast
eps3	Three-year ahead EPS forecast
bv0	Current book value of equity per share
bv1	Expected book value of equity per share in period t=1
bv2	Expected book value of equity per share in period t=2

To obtain the estimates of implied cost of capital, we rely on Stata's separate programming language Mata (`[help mata]`). Specifically, we rely on the `mm_root` function—Brent's

[8] Of course it is better to choose the values of this parameter based on actual expectations that may vary across observations. Keep in mind that, as a practical matter, the choice of growth rate affects the extent to which the procedure described below is able to find valid estimates, since the valuation model requires $g < r_e$.

univariate zero (root) finder—which searches for the value of x that causes the function $f(x)$ to approximate zero (`[help mf_mm_root]`). If not installed already, type `ssc inst moremata` to install this function. Replacing r_e with x, we can write the problem as:

$$f(x) = -P_0 + bv_0 + \frac{ri_1}{(1+x)} + \frac{ri_2}{(1+x)^2} + \frac{ri_3}{(1+x)^3} + \frac{ri_3(1+g)}{(x-g)(1+x)^3} = 0 \qquad \text{(A.9)}$$

That is, we want to know which value of x (i.e., r_e) results in $f(x) = 0$.

Although my dataset consists of 59,292 firm-year observations, the following code is extremely fast in estimating the implied cost of equity capital for each of these observations. On my computer, it took only a few seconds to solve the 59,292 equations, which illustrates the power and efficiency of Mata:

```
clear all
use "$path\InFiles\icc.dta", clear
gen g=0.01
gen icc=.
gen returncode=.

mata:
    st_view(p=., ., "p")
    st_view(bv0=., ., "bv0")
    st_view(bv1=., ., "bv1")
    st_view(bv2=., ., "bv2")
    st_view(eps1=., ., "eps1")
    st_view(eps2=., ., "eps2")
    st_view(eps3=., ., "eps3")
    st_view(g=., ., "g")
    n=rows(p)

    solutions=J(n,1,.)
    rcode=J(n,1,.)
    function myfunc(x, p_i, bv0_i, bv1_i, bv2_i, eps1_i, eps2_i, eps3_i,
    ↪  g_i){ ///
        return(-p_i+bv0_i+ ///
        (eps1_i-x*bv0_i)/(1+x)+ ///
        (eps2_i-x*bv1_i)/(1+x)^2+ ///
        (eps3_i-x*bv2_i)/(1+x)^3+ ///
        ((eps3_i-x*bv2_i)*(1+g_i))/((x-g_i)*(1+x)^3))
        }

    for(i=1; i<=n; i++) {
        p_i=p[i]
        bv0_i=bv0[i]
        bv1_i=bv1[i]
        bv2_i=bv2[i]
        eps1_i=eps1[i]
```

```
        eps2_i=eps2[i]
        eps3_i=eps3[i]
        g_i=g[i]
        rc=mm_root(x=., &myfunc(), .01001, .5, 1e-9, 1000, p_i, bv0_i, bv1_i,
        ↪  bv2_i, eps1_i, eps2_i, eps3_i, g_i)
        solutions[i,1]=x
        rcode[i,1]=rc
    }
    st_store(., "icc", solutions[.,1])
    st_store(., "returncode", rcode[.,1])
end

sum icc, d
```

We obtain the following distribution of the implied cost of equity capital for this sample:

```
                          icc
-------------------------------------------------------------
      Percentiles      Smallest
 1%     .0226857       .0100862
 5%     .0357135       .0101096
10%     .0439906       .0101111       Obs               59,292
25%     .0587209       .0101379       Sum of wgt.       59,292

50%     .0772947                      Mean           .0860031
                       Largest       Std. dev.       .047668
75%     .1012116             .5
90%     .1324107             .5       Variance       .0022722
95%     .1595183             .5       Skewness       3.528458
99%       .26236             .5       Kurtosis       25.83708
```

Now let's look at what the code does exactly. We first create a new variable for the growth rate (as said, here we set $g = 0.01$ but this can also vary across observations) and create an empty numeric variable `icc` that will the store the estimated values of r_e. We also create a new empty variable `returncode`. This is not required, but still useful as it will contain information on the success of the root finder.

Next, by specifying `mata:` we enter Mata mode, which means we enter a completely different environment with a different programming language (although Mata can communicate with the Stata environment to load or write data). Such communication happens for example with the `st_view()` Mata function, which takes a "view" on the data currently loaded in Stata's memory (the alternative is `st_data()`, which actually makes a copy of the data; both approaches have their advantages and disadvantages, see [`help mata st_view()`]). For instance, the line `st_view(p=., ., "p")` tells the program to create a new vector called "p" (the first p) that contains all of the observations (as indicated by the dot) of the variable `p` that is currently in

Stata's memory (the second p). The same is done for the other variables that we need as inputs for the estimations.

The line `n=rows(p)` simply creates a new scalar called "n" that captures the total number of observations in the data based on the number of rows in the `p` vector we just created. Next, with `solutions=J(n,1,.)` we create a matrix, specifically an $N \times 1$ row vector, with empty values (as indicated by the dot). In this vector we will be storing the individual estimates of implied costs of capital. We do the same with `rcode=J(n,1,.)` to store information on the success of the root finder.

The command starting with `function` is an important one, because the function we create here is the key ingredient for the solver that we will use to obtain estimates. As you can see, this command stretches over multiple lines of the do-file (note that the `///` preceded by a space allows us to ensure the command continues on the subsequent line; this is often useful with long commands when we want to make the code more easily readable). To simplify things, note that the command simplifies to:

```
function myfunc(A){
    return(B)
}
```

Here, `myfunc` simply refers to the name of the function we create. The part in between parentheses, `A`, captures all of the input scalars that the function will use. Note that this function will use scalar values as inputs instead of the complete vectors of data, which is why we use `p_i` instead of `p` as input. The actual value of `p_i` (which is the stock price of observation i) is determined later before we actually start to use this function. The other input scalars are defined similarly, except for the scalar `x`, which will be the placeholder for the value of x that solves the problem defined in equation (A.9). The `return(B)` part refers to the value that is returned by the function, where `B` is the function $f(x)$ depicted in equation (A.9) that we want to set equal to zero.

The next part displays the structure of the `for` loop in Mata. For values of the index starting at value 1, incremented by a value of 1 each iteration, and ending at the value `n`, the loop structure is as follows:

```
for(i=1; i<=n; i++) {
        ...
}
```

In the first eight lines within the loop, starting with `p_i=`, we store the values of the variables for the ith observation into the scalars we will use as input into the function `myfunc`. Now the real action takes place in the next line, which simplifies to:

```
rc=mm_root(x=., &myfunc(), start, .5, 1e-9, 1000, varlist)
```

As described in more detail in `[help mata mm_root()]`, the value stored in scalar `rc` is the return code. If the value equals 0, all went well. If not, no valid solution was found.[9] The first argument of `mm_root` is `x=.`, which will contain the solution value of $x = r_e$. The second argument `&myfunc()` is a "pointer scalar" which contains the address of the function for which we are searching the zero value solution. In our case the function name is `myfunc` as defined before. The next two arguments with values .01001 and .5 determine the lower and upper endpoints of the search interval to find the implied cost of capital solutions. The fifth argument specifies the tolerance level of the estimate, which refers to how far away from zero the calculation should be before the search stops. The sixth element with value 1000 specifies the number of iterations used by the root finder. Finally, `varlist` is set to `p_i, bv0_i, bv1_i, bv2_i, eps1_i, eps2_i, eps3_i, g_i`, which are the additional parameters that need to be passed on to the function `myfunc()`.[10]

Given the solution `x` that sets the function equal to zero, we store the values of the solutions into the `solutions` vector. We do the same for the return codes. For example, with the line `solutions[i,1]=x` we tell Mata to store the solution `x` of observation i in the ith row (and first column) of the `solutions` vector. Finally, after having completed the loop, we can push the vectors back to Stata using `st_store()`. For example, `st_store(., "icc", solutions[.,1])` pushes the information in the complete `solutions` vector (`[.,1]` refers to all rows (as indicated by the first element, the dot) and the first column (as indicated by the second element, the 1)) back to Stata. Now we can save the data and return to our usual practice of using Stata for the additional data management and hypothesis testing.

Of course this example was very specific to the valuation model and three-year forecast horizon described at the start of this chapter. But the procedure can be adapted to compute

[9] In my data, I find that 186 observations have a return code equal to 3 and their values of implied cost of capital are set to the upper bound of 0.5 even though no solution was found in the optimization problem. We should therefore probably discard these observations.

[10] Note that one limitation of the `mm_root()` function is that the number of additional parameters is restricted to a maximum of ten. Since we already use eight in `varlist`, this means we are close to this limit. That also means that if we would use a more complicated valuation model than in equation (A.7) with a longer forecast horizon, the function would not work for us. In this case, we should be a bit creative by adjusting the optimization equation (A.9) in such a way that only a maximum of ten input parameters are needed. The illustration at the end of this chapter provides an example of how to do this.

the internal rates of return for other models as well. Below is an example of how to estimate the implied cost of equity capital using a residual income valuation model with a five-year finite forecast horizon. Given the maximum number of parameters that can be passed to the `mm_root()` function, in this code the future book values of equity are computed within the equations of the `myfunc` function based on the clean surplus relation and a dividend payout ratio captured by variable `k`:

```
clear all
use "$path\InFiles\icc.dta", clear
gen g=0.01
gen icc=.
gen returncode=.

mata:
    st_view(p=., ., "p")
    st_view(bv0=., ., "bv0")
    st_view(bv1=., ., "bv1")
    st_view(eps1=., ., "eps1")
    st_view(eps2=., ., "eps2")
    st_view(eps3=., ., "eps3")
    st_view(eps4=., ., "eps4")
    st_view(eps5=., ., "eps5")
    st_view(g=., ., "g")
    st_view(k=., ., "k")
    n=rows(p)

    solutions=J(n,1,.)
    rcode=J(n,1,.)
    function myfunc(x, p_i, bv0_i, bv1_i, eps1_i, eps2_i, eps3_i, eps4_i,
    ↪  eps5_i, g_i, k_i) ///
        return(-p_i+bv0_i+ ///
        ((eps1_i-x*bv0_i)/(1+x))+ ///
        ((eps2_i-x*bv1_i)/(1+x)^2)+ ///
        ((eps3_i-x*(bv1_i+eps2_i*(1-k_i)))/(1+x)^3)+ ///
        ((eps4_i-x*(bv1_i+eps2_i*(1-k_i)+eps3_i*(1-k_i)))/(1+x)^4)+ ///
        ((eps5_i-x*(bv1_i+eps2_i*(1-k_i)+eps3_i*(1-k_i)+eps4_i*(1-k_i)))
        ↪  /(1+x)^5)+ ///
        (((eps5_i-x*(bv1_i+eps2_i*(1-k_i)+eps3_i*(1-k_i)+eps4_i*(1-k_i)))
        ↪  *(1+g_i))/((x-g_i)*(1+x)^5)))

    for(i=1; i<=n; i++) {
        p_i=p[i]
        bv0_i=bv0[i]
        bv1_i=bv1[i]
        eps1_i=eps1[i]
        eps2_i=eps2[i]
        eps3_i=eps3[i]
        eps4_i=eps4[i]
        eps5_i=eps5[i]
```

```
        k_i=k[i]
        g_i=g[i]
        rc=mm_root(x=., &myfunc(), .01001, 0.5, 1e-9, 1000, p_i, bv0_i,
        ↪  bv1_i, eps1_i, eps2_i, eps3_i, eps4_i, eps5_i, g_i, k_i)
        solutions[i,1]=x
        rcode[i,1]=rc
    }
    st_store(., "icc", solutions[.,1])
    st_store(., "returncode", rcode[.,1])
end
```

A.9 Computing marginal effects

Because the coefficients from a logit or probit estimation cannot be interpreted as easily as those obtained using OLS, we often want to compute "marginal effects" after the estimations. These marginal effects capture the change in the probability that $y = 1$ when we change the independent variable x by a specific amount. For example, in Table 4 of Veenman and Verwijmeren [2018] we calculated the changes in the probability that a firm reports a nonnegative earnings surprise ($y = Nonneg$) when an independent variable would increase by one unit. Specifically, we computed the marginal effect of changing the key test variable $Pess\_Combined$ from 0 to 1 on the probability that $Nonneg = 1$, while holding all other factors constant at their means. To achieve this, we also had to perform this estimation on a monthly basis and average all of the marginal effects across the 360 months in our sample. The command we used for the computation of the marginal effects was `mfx` ([`help mfx`]), although the more recent command `margins` can do the same ([`help margins`]).

Below is the code used for the calculation of the marginal effects in Table 4 of Veenman and Verwijmeren [2018]. There are 15 independent variables for which we have to store the monthly marginal effects. The test variable $Pess\_Combined$ is labeled `qsumpess` in the data. To ensure the code is easier to read, I store the string of independent variable names in a local macro called `varlist`.

```
use "$path\InFiles\vvdata.dta", clear
forvalues i=1(1)15{
    qui gen m_`i'=.
}

local varlist "qsumpess qlnmv qlnbtm qlagret1 qlagret qidvol qinst qlnn
↪  qassetgr qopprof qdqearn qes_lag*"

forvalues i=1(1)360{
    di `i'
    qui logit nonneg `varlist' if ym==`i'
    qui local obs = e(N)
    qui mfx if ym==`i', var(`varlist')
    qui mat margeff = e(Xmfx_dydx)
    qui svmat margeff
    forvalues j=1(1)15{
        qui sum margeff`j'
        qui replace m_`j'=r(mean) if ym==`i'
    }
    qui drop margeff*
}
```

```
duplicates drop ym, force
tabstat m_*, columns(statistics)
```

The output gives us the marginal effects for the 15 variables contained in the local macro `varlist`:

```
. tabstat m_*, columns(statistics)

    variable |      mean
-------------+----------
         m_1 |  .1620645
         m_2 | -.0145295
         m_3 | -.0439386
         m_4 |  .1067522
         m_5 |  .0862231
         m_6 | -.0084982
         m_7 |  .0310358
         m_8 |  .0451097
         m_9 |  -.015957
        m_10 |  .0114448
        m_11 |  .0262856
        m_12 |  .1359686
        m_13 |   .029327
        m_14 |  .0089677
        m_15 |  .0205288
------------------------
```

A.10 Computing earnings surprises using analyst forecast data

This chapter illustrates how to compute earnings surprises based on consensus analyst earnings forecasts and how to attach these surprises to a dataset with firm-year accounting data. These earnings surprises are potentially useful for the estimation of earnings response coefficients (ERCs) or for the construction of an indicator variable that identifies firms that meet or (just) beat the analyst earnings forecast (see Bissessur and Veenman [2016] for references and a critical discussion of this indicator variable). Below, I use the latest median consensus EPS forecast released before the annual earnings announcement to create this indicator variable for firms that meet or just beat the analyst forecast. Data used for this example are:

- Dataset 1: Firm-year level accounting data (here: Compustat Fundamentals Annual US) with the IBES firm/security identifier attached (`ticker`).
- Dataset 2: Monthly consensus analyst EPS forecast data for the upcoming annual earnings announcement (here: IBES, US detail file; fiscal period indicator `fpi=1`.
- Dataset 3: IBES actual EPS data (here: IBES, US actuals file).

Input variables for these datasets are as follows:

Variable	Description dataset 1: `comp_firmyears.dta`
gvkey	Compustat company identifier
fyear	Fiscal year
datadate	Fiscal-year-end date
ticker	IBES identifier
...	*Any accounting data*

Variable	Description dataset 2: `ibes_statsumu_epsus.dta`
ticker	IBES identifier
statpers	Date/month of consensus forecast
medest	Median consensus forecast of EPS
fpedats	Fiscal year end date to which the forecast refers

Variable	Description dataset 3: `ibes_actu_epsus.dta`
ticker	IBES identifier
pends	Fiscal year end date
value	Value of actual EPS
anndats	Earnings announcement date

Open the do-file editor, start a clean do-file, and save the empty do-file to a location and name of preference. Now use this do-file to store all the programming code below. Open dataset 2 and rename the fiscal year end-date variable to be consistent with dataset 3, from `fpedats` to `pends`. Next, merge the data using `joinby` based on the IBES ticker and fiscal year-end with dataset 3, such that EPS forecasts and actual EPS are combined into one file. Also make sure you drop observations with missing fields and observations where the consensus forecast is not measured before the earnings announcement:

```
use "$path\InFiles\ibes_statsumu_epsus.dta", clear
ren fpedats pends
keep ticker statpers pends medest
sort ticker pends
joinby ticker pends using "$path\InFiles\ibes_actu_epsus.dta", unmatched(master)
drop _merge
drop if medest==.
drop if value==.
drop if statpers>=anndats
```

Next, we take the following step to retain only the last consensus forecast that is released before each earnings announcement:

```
gsort ticker pends -statpers
duplicates drop ticker pends, force
```

Now we can compute the earnings surprise (or "forecast error") variable. In this step, it is important to make sure that we round the data to the nearest cent. This is because Stata sometimes changes the contents of a cel by a marginal amount, e.g., 0.01 can become 0.010001 because of floating point value precision and the way Stata stores decimal information in its memory. While an earnings surprise of 0.010001 should be counted as meet/beat by 0 or 1 cent, Stata will not do so because technically speaking $0.010001 > 0.01$. Therefore, I use the `round` function, combined with translation of the forecast and actual values in cents, to eliminate the use of decimals.

```
gen valueround=round(100*value)
gen forecastround=round(100*medest)
gen ferror=valueround-forecastround
```

It is useful to inspect the resulting data to see whether the data conform to our expectations. For example, we may check to see whether we can observe the well-known discontinuity in the distribution of the earnings surprise variable around zero:

```
tabstat ferror if abs(ferror)<=10,by(ferror) stats(N)
```

Next, we create an indicator variable `meet` that is set equal to 1 for observations where the (rounded) earnings surprise is equal to 0 or 1 cent, and 0 otherwise.

```
gen meet=0
replace meet=1 if ferror==0 | ferror==1
```

We should also make sure that we have common variables with dataset 1, to be able attach the newly created variable to our dataset of firm-year observations. To do so, generate a fiscal year

`fyear` variable and sort on company identifier and fiscal year. Here, we rely on Compustat's convention to set `fyear` equal to the calendar year for fiscal years that end in June or later, and the previous calendar year for fiscal years that end before June.

```
gen fyear=year(pends)
replace fyear=fyear-1 if month(pends)<6
duplicates drop gvkey fyear, force
sort gvkey fyear
save "$path\OutFiles\meet.dta", replace
```

Finally, we can open dataset 1 and again use `joinby` to merge the dataset with the newly created dataset that contains the `meet` variable.

```
use "$path\InFiles\comp_firmyears.dta", clear
joinby ticker fyear using "$path\OutFiles\meet.dta", unmatched(master)
drop _merge
drop if at==.
sum at meet
drop if meet==.
sort gvkey fyear
save "$path\OutFiles\firmyears_meet.dta", replace
```

With these steps, we have created a dataset with firm-year level accounting data and an additional variable that captures whether or not a firm's annual earnings per share exactly meet or marginally beat the analyst consensus forecast that was available just before the earnings announcement.

A.11 Computing earnings surprises using individual forecasts

Chapter A.10 relied on consensus forecasts of EPS obtained from the IBES summary files to construct the final earnings expectations before an earnings announcement. This consensus, however, is only constructed on a monthly basis (more precisely: on the Thursday before the third Friday of the month). This means that for one earnings announcement the consensus may be one day old, while for another earnings announcement the consensus can be almost one month old. In addition, this pre-calculated consensus may contain old forecasts that we might rather exclude from our proxy for the market's earnings expectations. Therefore, this chapter illustrates how you can construct a consensus forecast yourself using data from the IBES detail files.[11] The code below is based on Schafhäutle and Veenman [2023]. It relies on quarterly earnings announcements and can be used in combination with chapter A.12. The input data are the raw IBES data obtained from the WRDS server (see chapter A.1):

- `actu_epsus.dta`: a file with actual EPS data obtained from the IBES unadjusted actuals file for US firms.
- `detu_epsus.dta`: a file with individual analyst EPS forecasts from the IBES unadjusted files for US firms.

The first step is to clean up and prepare a file with actual (reported) quarterly earnings data.

```
use "$path\InFiles\actu_epsus.dta", clear
rename *, lower
// Keep only quarterly earnings:
keep if pdicity=="QTR"
keep if measure=="EPS"
drop if value==.
gduplicates report ticker pends
gduplicates tag ticker pends, gen(dupid)
gsort dupid ticker pends
keep if curr_act=="USD"
gduplicates report ticker pends
keep ticker pends value anndats
ren pends fpedats
ren value actual
sort ticker fpedats
save "$path\OutFiles\actualq.dta", replace
```

Next, prepare and clean up the individual-analyst quarterly forecast data.

```
use "$path\InFiles\detu_epsus.dta", clear
keep if report_curr=="USD"
```

[11] Note that the data are from the IBES unadjusted files, which means additional adjustments for stock splits need to be made (to the extent a stock split occurs between the individual analyst forecast and the earnings announcement).

```
// Keep only those quarterly forecasts for the current or next quarter:
keep if fpi=="6" | fpi=="7"
// Drop ambiguous analyst codes:
drop if analys==0
drop if analys==1
// Manage duplicate forecasts by same analyst for same firm on one day:
gen n=_n
gduplicates report analys ticker fpedats anndats
gegen gr=group(analys ticker fpedats anndats)
gegen count=count(gr),by(gr)
gegen anntims2=group(anntims)
gegen revtims2=group(revtims)
gegen acttims2=group(acttims)
gsort count gr fpi -anntims2 -revdats -revtims2 -actdats -acttims2 n
gduplicates drop analys ticker fpedats anndats, force
ren anndats anndats_d
sum value
// Keep each analyst's latest forecast only:
gsort ticker fpedats analys -anndats_d
gduplicates drop ticker fpedats analys, force
sum value
// Attach actual earnings file:
gsort ticker fpedats
joinby ticker fpedats using "$path\OutFiles\actualq.dta", unmatched(master)
drop _merge
drop if actual==.
drop if anndats==.
drop if anndats_d>=anndats
drop if anndats<=fpedats
```

Following Schafhäutle and Veenman [2023], we can apply filters that restrict the earnings announcement to occur a maximum of 120 days after the fiscal quarter end and the forecast to be a maximum of 120 days old at the time of the earnings announcement. These parameters are easily changed and can depend on the institutional context (e.g., earnings announcement delays vary by country).

```
// Drop late EAs and stale forecasts at EA date
drop if anndats>fpedats+120
drop if anndats_d<anndats-120
sum value
keep ticker fpedats anndats* actual value analys estimator
sort ticker fpedats analys
save "$path\OutFiles\ibes_individual.dta", replace
```

This code block creates a file that contains each individual analyst's forecast of quarterly EPS for each firm-quarter for which the earnings announcement date and actual EPS are identified in IBES. We will use this file below to construct our consensus forecast, but we can also use it in a setting such as that described in chapter A.12.

This newly created dataset of each individual analyst's latest forecast before the earnings announcement can be used to construct the most recent consensus forecast that is available before each earnings announcement. All we have to do is compute the mean (or alternatively: median) of the outstanding individual forecasts. Using this mean as the consensus forecast, we can compute an earnings surprise variable (`surp`) that is based on all the forecasts that have been issued before earnings are announced. Similar to chapter A.10, we can verify the accuracy of the data by inspecting descriptive statistics and a histogram for this new variable.

```
use "$path\OutFiles\ibes_individual.dta", clear
// Compute forecast consensus:
egen mean=mean(value), by(ticker fpedats)
drop if mean==.
duplicates drop ticker fpedats, force
// Compute earnings surprises:
replace actual=100*actual
replace actual=round(actual)
replace mean=100*mean
replace mean=round(mean)
gen surp=actual-mean
sum surp,d
tabstat surp if abs(surp)<=10, by(surp) stats(N)
hist surp if abs(surp)<=10
keep ticker surp fpedats anndats
save "$path\OutFiles\consensus_surprise.dta", replace
```

A.12 Special case: measures of past analyst forecast pessimism

The code presented below helps to compute the consensus and individual forecast pessimism measures from Veenman and Verwijmeren [2018]. The primary goal of the code is to construct firm-month measures of past analyst forecast pessimism, based either on past consensus forecast errors (12 quarters) or on past individual analyst forecast errors (12 months). See the paper, and in particular Section III and Figure 1 of the paper, for more details. The code assumes the following input files:

- `crspm_ibes.dta`: a data file with stock returns from the CRSP monthly stock files with the IBES ticker attached. Here the file includes the following variables (but this will depend on the specific research project): `permno`; `date`; `ret`; `ticker` (from IBES); `year` (based on `date`); and `month` (based on `date`). This can be considered the master file, to which the two measures for past forecast pessimism will be attached.
- `consensus_surprise.dta`: a file with self-constructed consensus earnings surprises, created as in chapter A.11.
- `ibes_individual.dta`: a file with individual analysts' latest EPS forecasts before earnings announcements, created as in chapter A.11.

A.12.1 Past consensus forecast pessimism measure

The first measure is a score of the most recently announced 12 quarterly earnings surprises that turned out to be positive (i.e., based on pessimistic forecasts) versus negative (i.e., based on optimistic forecasts). For this purpose, we can use the `consensus_surprise.dta` file. Because all of the data are processed and constructed at the firm-month level, we first need to clean up this file a little. Specifically, because some firm-months may contain multiple earnings announcements (e.g., because one of them is severely delayed), there may be duplicate firm-months in the earnings announcement data. Because we will be merging these data with another firm-month dataset, we'll have to ensure these duplicates are eliminated. In this regard, it is probably wise to retain only the most recent earnings announcement in the data.

```
use "$path\OutFiles\consensus_surprise.dta", clear
gen y=year(anndats)
gen m=month(anndats)
gsort ticker y m -anndats -fpedats
duplicates drop ticker y m, force
keep ticker y m surp fpedats anndats
save "$path\OutFiles\consensus_surprise_clean.dta", replace
```

The block of code presented below creates a new file `pess_consensus.dta`. This file contains a new firm-month-specific variable `pess_consensus` that captures the proportion of the most

recent 12 quarterly earnings surprises that were positive (ignoring surprises equal to zero cents per share), up to month *t-1*. In addition, the code creates up to variables that each contain these 12 quarterly earnings surprises (if available), as well as a date variable that can be used to identify the most recent earnings announcement date used in the construction of `pess_consensus`. Again see the paper, and in particular Section III and Figure 1, for more details.

```
use "$path\InFiles\crspm_ibes.dta", clear
keep ticker year month
// Create new year and month variables for previous month:
gen y=year
gen m=month-1
replace y=y-1 if m<1
replace m=m+12 if m<1
// Expand the dataset to have the [-35,0] month period for each firm-month:
gen n=_n
expand(36)
sort n
by n: gen m_=_n
gen eventm=1 if m_==1
replace m=m+1-m_
replace y=y-1 if m<1
replace m=m+12 if m<1
replace y=y-1 if m<1
replace m=m+12 if m<1
replace y=y-1 if m<1
replace m=m+12 if m<1
drop m_ n
// Attach earnings surprise data:
sort ticker y m eventm
duplicates drop ticker y m, force
joinby ticker y m using "$path\OutFiles\consensus_surprise_clean.dta",
↪  unmatched(master)
drop _merge
sum year surp
// Create variable to store cumulative info on previous pessimism:
sort ticker y m eventm
duplicates drop ticker y m, force
egen tid=group(ticker)
egen ym=group(y m)
tsset tid ym
gen count=1 if surp!=. & surp!=0
gen pess=0 if surp<0
replace pess=1 if surp!=. & surp>0
sum pess
gen p1=0
gen c1=0
// Create variables to store previous 12 quarterly surprises:
forvalues i=1(1)12{
    gen surpm`i'=.
```

```
}
// Create variable for checking the fiscal quarter end and announcement date:
gen surpdate=.
gen surpdateann=.
forvalues i=1(1)36{
    qui replace p1=p1+l`i'.pess if l`i'.pess!=. & c1<12
    qui replace surpm1=l`i'.surp if surpm1==. & l`i'.surp!=. & c1<1
    forvalues u=1(1)11{
        qui local q=`u'+1
        qui replace surpm`q'=l`i'.surp if surpm`q'==. & l`i'.surp!=. &
        ↪  c1==`u'
    }
    qui replace surpdate=l`i'.fpedats if surpdate==. & l`i'.fpedats!=. & c1<1
    qui replace surpdateann=l`i'.anndats if surpdateann==. & l`i'.anndats!=.
    ↪  & c1<1
    qui replace c1=c1+l`i'.count if l`i'.count!=. & c1<12
    di `i'
}
// Require at least 4 quarterly earnings surprises:
tabstat c1, by(c1) stats(N)
drop if c1<4
// Create consensus pessimism measure:
gen pess_consensus=p1/c1
sum pess_consensus
keep if eventm==1
keep ticker y m pess_consensus surpm* surpdate*
format surpdate* %d
forvalues i=1(1)12{
    replace surpm`i'=surpm`i'/100
}
gen year=y
gen month=m
replace month=month+1
replace year=year+1 if month>12
replace month=month-12 if month>12
drop y m
save "$path\OutFiles\pess_consensus.dta", replace
```

A.12.2 Past individual forecast pessimism measure

The second measure is similarly a score based on information from previous quarterly earnings announcements and analyst forecasts. The key difference is that it is based on *individual* instead of *consensus* forecast errors in the 12-month period up to month *t-1*. Below the creation of the dataset with variable `pess_individual` is performed in two steps. In the first step (**Step A**) we create a file with unique analyst-ticker-year-month combinations. In the second step (**Step B**), we calculate the individual analyst pessimism measure over the past 12 months. For every combination of analyst and year-month, we look back at all individual forecast errors

based on earnings releases in the previous 12 months. In the process, we also split the files up in smaller chunks to speed up the process.

Step A

```
*******************************************************************************
* Step A: Create a file with unique analyst-ticker-year-month combinations
*******************************************************************************
use "$path\OutFiles\ibes_individual.dta", clear
// Remove duplicates because of multiple announcements in one month:
gen y=year(anndats)
gen m=month(anndats)
gen invdats=1000000/anndats
gen invdats2=1000000/fpedats
sort analys ticker y m invdats invdats2
duplicates drop analys ticker y m, force
// Create surprise and individual pessimism variables:
replace actual=100*actual
replace actual=round(actual)
replace value=100*value
replace value=round(value)
gen pessimist=0 if value!=. & value>actual
replace pessimist=1 if actual!=. & actual>value
sum pessimist
// Assume analyst has been active 6 months before a specific EA to create a
↪  panel of analyst-year-months:
gen year=year(anndats)
gen month=month(anndats)
keep analys ticker year month pessimist anndats anndats_d fpedats
forvalues i=1(1)6{
    gen m_`i'=`i'
}
gen id=_n
reshape long m_, i(id) j(n)
replace anndats=. if m_>1
replace pessimist=. if m_>1
replace month=month+1-n
replace year=year-1 if month<1
replace month=month+12 if month<1
sort analys ticker year month anndats
duplicates drop analys ticker year month, force
gen m=month
gen y=year
egen ym=group(y m)
sort analys ym
keep analys ticker y m ym pessimist anndats
// Make sure that each analyst covers at least 2 firms
sort analys ticker
by analys: gen nr=_n
tsset analys nr
egen tickerid=group(ticker)
```

```
gen nr2=nr if tickerid!=l.tickerid
drop tickerid
egen count=count(nr2), by(analys)
sum count,d
drop if count<2
keep analys ticker y m pessimist anndats
// Create id for analysts and unique analyst-year-month combinations:
egen ym=group(y m)
sort analys ym ticker
egen aid=group(analys)
egen aidym=group(analys ym)
sort aidym
// Create date variable for verification of no overlap with return month:
gen hulpdate=mdy(m,1,y)
replace hulpdate=hulpdate-1
format hulpdate %d
gen cp=.
gen tot=.
gen maxd=.
compress
save "$path\OutFiles\ibes_individual_pess.dta", replace
```

Step B

```
*************************************************************************
* Step B: Calculate analyst pessimism measure over past 12 months
*************************************************************************
use "$path\OutFiles\ibes_individual_pess.dta", clear
local y=5
sum aid
local h=ceil(r(max)/`y')
local g=r(max)
forvalues a=1(1)`h'{
    qui use "$path\OutFiles\ibes_individual_pess.dta" if aid>(`a'-1)*`y' &
    ↪  aid<=`a'*`y', clear
    qui sum aid
    local f=r(min)
    local k=r(max)
    forvalues i=`f'(1)`k'{
        qui sum aidym if aid==`i'
        local m=r(min)
        local n=r(max)
        forvalues j=`m'(1)`n'{
            qui sum hulpdate if aid==`i' & aidym==`j'
            qui local d=r(min)
            qui sum pessimist if aid==`i' & aidym<`j' & anndats>=`d'-365 &
            ↪  anndats<`d'
            qui replace tot=r(N) if aid==`i' & aidym==`j'
            qui replace cp=r(sum) if aid==`i' & aidym==`j'
            qui sum anndats if aid==`i' & aidym<`j' & anndats>=`d'-365 &
            ↪  anndats<`d'
```

```
            qui replace maxd=r(max) if aid==`i' & aidym==`j'
        }
        di `a' ":::" `h' "   ++++++++" "   " `i' "//\\" `g'
    }
    qui save "$path\OutFiles\Temp\ibes_individual_pess_`a'.dta", replace
}
// Combine individual temporary sub-files created in previous step:
use "$path\OutFiles\ibes_individual_pess.dta", clear
local y=5
sum aid
local h=ceil(r(max)/`y')
local g=r(max)
use "$path\OutFiles\Temp\ibes_individual_pess_1.dta", clear
forvalues a=2(1)`h'{
    qui append using "$path\OutFiles\Temp\ibes_individual_pess_`a'.dta"
    di `a' " / " `h'
}
keep analys ticker y m tot cp maxd
drop if tot==.
drop if tot==0
gen pessclean=cp/tot
sort analys ticker y m
// Aggregate by firm-year-month
egen pess_individual=mean(pessclean), by(ticker y m)
egen md=max(maxd), by(ticker y m)
duplicates drop ticker y m, force
sum pess_individual
keep ticker y m pess_individual md
compress
sort ticker y m
gen year=y
gen month=m
replace month=month+1
replace year=year+1 if month>12
replace month=month-12 if month>12
drop y m
save "$path\OutFiles\pess_individual.dta", replace
```

Finally, we can combine all data and attach the consensus and individual pessimism measure files to our master dataset with monthly return data. The variables can now be transformed into monthly ranks and used in monthly cross-sectional return prediction tests together with other variables.

```
use "$path\InFiles\crspm_ibes.dta", clear
sum ret
// Attach consensus pessimism measure
joinby ticker year month using "$path\OutFiles\pess_consensus.dta",
↪  unmatched(master)
drop _merge
sum ret pess_consensus
```

```
// Attach individual pessimism measure
joinby ticker year month using "$path\OutFiles\pess_individual.dta",
 ↪  unmatched(master)
drop _merge
sum ret pess_consensus pess_individual
drop if pess_consensus==.
drop if pess_individual==.
// Verify that prior pessimism data really measured before start of the
 ↪  month:
gen datehulp=mdy(month(date),1,year(date))
gen diff=datehulp-surpdateann
sum diff,d
drop diff
gen diff=datehulp-md
sum diff,d
drop diff
save "$path\OutFiles\crspm_ibes_pess_measures.dta", replace
```

Bibliography

Abadie, A. and J. Spiess. 2022. Robust Post-Matching Inference. *Journal of the American Statistical Association* 117 (538): 983–995.

Angrist, J. D. and J.-S. Pischke. 2009. *Mostly Harmless Econometrics: An Empiricist's Companion*. Princeton University Press, Princeton.

Armstrong, C. S., S. A. Glaeser, and S. Huang. 2022. Contracting with Controllable Risk. *The Accounting Review* 97 (4): 27–50.

Armstrong, C. S., C. D. Ittner, and D. F. Larcker. 2012. Corporate Governance, Compensation Consultants, and CEO Pay Levels. *Review of Accounting Studies* 17 (2): 322–351.

Ashbaugh-Skaife, H., D. W. Collins, W. R. Kinney Jr., and R. LaFond. 2008. The Effect of SOX Internal Control Deficiencies and Their Remediation on Accrual Quality. *The Accounting Review* 83 (1): 217–250.

Baker, A. C., D. F. Larcker, and C. C. Y. Wang. 2022. How Much Should We Trust Staggered Difference-in-Differences Estimates? *Journal of Financial Economics* 144 (2): 370–395.

Barrios, J. M. 2021. Staggeringly Problematic: A Primer on Staggered DiD for Accounting Researchers. *Working paper, https://papers.ssrn.com/abstract=3794859* .

Baum, C. F. 2015. *An Introduction to Stata Programming, Second Edition*. Stata Press, College Station, Texas, 2nd edition edition.

Bissessur, S. W. and D. Veenman. 2016. Analyst Information Precision and Small Earnings Surprises. *Review of Accounting Studies* 21 (4): 1327–1360.

Breuer, M. and E. deHaan. 2023. Using and Interpreting Fixed Effects Models. *Available at SSRN: https://ssrn.com/abstract=4539828* .

Callaway, B. and P. H. C. Sant'Anna. 2021. Difference-in-Differences with Multiple Time Periods. *Journal of Econometrics* 225 (2): 200–230.

Cameron, A. and P. Trivedi. 2010. Microeconometrics Using Stata, Revised Edition. Stata Press books, StataCorp LP.

Cameron, A. C., J. B. Gelbach, and D. L. Miller. 2008. Bootstrap-Based Improvements for Inference with Clustered Errors. *The Review of Economics and Statistics* 90 (3): 414–427.

Cameron, A. C., J. B. Gelbach, and D. L. Miller. 2011. Robust Inference With Multiway Clustering. *Journal of Business & Economic Statistics* 29 (2): 238–249.

Cameron, A. C. and D. L. Miller. 2015. A Practitioner's Guide to Cluster-Robust Inference. *Journal of Human Resources* 50 (2): 317–372.

Cascino, S., M. Széles, and D. Veenman. 2023. Does CEO Inside Debt Really Improve Financial Reporting Quality? *Working paper, https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4310933* .

Chen, W., P. Hribar, and S. Melessa. 2018. Incorrect Inferences When Using Residuals as Dependent Variables. *Journal of Accounting Research* 56 (3): 751–796.

Chen, W., P. Hribar, and S. Melessa. 2023. Standard Error Biases When Using Generated Regressors in Accounting Research. *Journal of Accounting Research* 61 (2): 531–569.

Christensen, H. B., L. Hail, and C. Leuz. 2013. Mandatory IFRS Reporting and Changes in Enforcement. *Journal of Accounting and Economics* 56 (2–3, Supplement 1): 147–177.

Claus, J. and J. Thomas. 2001. Equity Premia as Low as Three Percent? Evidence from Analysts' Earnings Forecasts for Domestic and International Stock Markets. *The Journal of Finance* 56 (5): 1629–1666.

Conley, T., S. Gonçalves, and C. Hansen. 2018. Inference with Dependent Data in Accounting and Finance Applications. *Journal of Accounting Research* 56 (4): 1139–1203.

Core, J. E., W. Guay, and D. F. Larcker. 2008. The Power of the Pen and Executive Compensation. *Journal of Financial Economics* 88 (1): 1–25.

Correia, S. 2015. Singletons, Cluster-Robust Standard Errors and Fixed Effects: A Bad Mix. *Working paper, http://scorreia.com/research/singletons.pdf* .

Cunningham, S. 2021. *Causal Inference: The Mixtape*. Yale University Press, New Haven ; London.

Dechow, P. M. and I. D. Dichev. 2002. The Quality of Accruals and Earnings: The Role of Accrual Estimation Errors. *The Accounting Review* 77 (4): 35–59.

DeFond, M., D. H. Erkens, and J. Zhang. 2016. Do Client Characteristics Really Drive the Big N Audit Quality Effect? New Evidence from Propensity Score Matching. *Management Science* 63 (11): 3628–3649.

Easton, P. D. 1998. Discussion of Revalued Financial, Tangible, and Intangible Assets: Association with Share Prices and Non-Market-Based Value Estimates. *Journal of Accounting Research* 36: 235–247.

Easton, P. D. 1999. Security Returns and the Value Relevance of Accounting Data. *Accounting Horizons* 13 (4): 399–412.

Easton, P. D. and G. A. Sommers. 2003. Scale and the Scale Effect in Market-based Accounting Research. *Journal of Business Finance & Accounting* 30 (1-2): 25–56.

Ecker, F., J. Francis, P. Olsson, and K. Schipper. 2013. Estimation Sample Selection for Discretionary Accruals Models. *Journal of Accounting and Economics* 56 (2–3): 190–211.

Fama, E. F. and K. R. French. 1993. Common Risk Factors in the Returns on Stocks and Bonds. *Journal of Financial Economics* 33 (1): 3–56.

Francis, J., R. LaFond, P. Olsson, and K. Schipper. 2005. The Market Pricing of Accruals Quality. *Journal of Accounting and Economics* 39 (2): 295–327.

Gassen, J., H. A. Skaife, and D. Veenman. 2020. Illiquidity and the Measurement of Stock Price Synchronicity. *Contemporary Accounting Research* 37 (1): 419–456.

Gassen, J. and D. Veenman. 2023. Outliers and Robust Inference in Archival Accounting Research. *Working paper, https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3880942* .

Gebhardt, W. R., C. M. C. Lee, and B. Swaminathan. 2001. Toward an Implied Cost of Capital. *Journal of Accounting Research* 39 (1): 135–176.

Gerakos, J. 2012. Discussion of Detecting Earnings Management: A New Approach. *Journal of Accounting Research* 50 (2): 335–347.

Goncharov, I. and D. Veenman. 2014. Stale and Scale Effects in Markets-Based Accounting Research: Evidence from the Valuation of Dividends. *European Accounting Review* 23 (1): 25–55.

Gormley, T. A. and D. A. Matsa. 2014. Common Errors: How to (and Not to) Control for Unobserved Heterogeneity. *Review of Financial Studies* 27 (2): 617–661.

Gould, W. 2018. *The Mata Book: A Book for Serious Programmers and Those Who Want to Be*. Stata Press, 1st edition edition.

Gow, I. D., G. Ormazabal, and D. J. Taylor. 2010. Correcting for Cross-Sectional and Time-Series Dependence in Accounting Research. *The Accounting Review* 85 (2): 483.

Griffin, J. M., P. J. Kelly, and F. Nardari. 2010. Do Market Efficiency Measures Yield Correct Inferences? A Comparison of Developed and Emerging Markets. *Review of Financial Studies* 23 (8): 3225–3277.

Hainmueller, J. 2012. Entropy Balancing for Causal Effects: A Multivariate Reweighting Method to Produce Balanced Samples in Observational Studies. *Political Analysis* 20 (1): 25–46.

Hansen, B. 2022. *Econometrics*. Princeton University Press, Princeton.

Hribar, P., T. Kravet, and R. Wilson. 2014. A New Measure of Accounting Quality. *Review of Accounting Studies* 19 (1): 506–538.

Hribar, P. and D. C. Nichols. 2007. The Use of Unsigned Earnings Quality Measures in Tests of Earnings Management. *Journal of Accounting Research* 45 (5): 1017–1053.

Jann, B. 2021. ROBREG: Stata Module Providing Robust Regression Estimators.

King, G. and R. Nielsen. 2019. Why Propensity Scores Should Not Be Used for Matching. *Political Analysis* 27 (4): 435–454.

Landsman, W. R., E. L. Maydew, and J. R. Thornock. 2012. The Information Content of Annual Earnings Announcements and Mandatory Adoption of IFRS. *Journal of Accounting and Economics* 53 (1–2): 34–54.

Lawrence, A., M. Minutti-Meza, and P. Zhang. 2011. Can Big 4 versus Non-Big 4 Differences in Audit-Quality Proxies Be Attributed to Client Characteristics? *The Accounting Review* 86 (1): 259–286.

Leone, A. J., M. Minutti-Meza, and C. E. Wasley. 2019. Influential Observations and Inference in Accounting Research. *The Accounting Review* 94 (6): 337–364.

Leung, E. and D. Veenman. 2018. Non-GAAP Earnings Disclosure in Loss Firms. *Journal of Accounting Research* 56 (4): 1083–1137.

MacKinnon, J. G. 2023. Fast Cluster Bootstrap Methods for Linear Regression Models. *Econometrics and Statistics* 26: 52–71.

MacKinnon, J. G., M. O. Nielsen, and M. D. Webb. 2023. Cluster-Robust Inference: A Guide to Empirical Practice. *Journal of Econometrics* 232 (2): 272–299.

McMullin, J. and B. Schonberger. 2022. When Good Balance Goes Bad: A Discussion of Common Pitfalls When Using Entropy Balancing. *Journal of Financial Reporting* 7 (1): 167–196.

McNichols, M. F. 2002. Discussion of the Quality of Accruals and Earnings: The Role of Accrual Estimation Errors. *The Accounting Review* 77 (4): 61.

Morgan, S. L. and C. Winship. 2015. *Counterfactuals and Causal Inference*. Cambridge University Press, New York, 2nd edition edition.

Owens, E. L., J. S. Wu, and J. Zimmerman. 2017. Idiosyncratic Shocks to Firm Underlying Economics and Abnormal Accruals. *The Accounting Review* 92 (2): 183–219.

Petersen, M. A. 2009. Estimating Standard Errors in Finance Panel Data Sets: Comparing Approaches. *Review of Financial Studies* 22 (1): 435–480.

Roberts, M. R. and T. M. Whited. 2012. Endogeneity in Empirical Corporate Finance. *Working paper, http://papers.ssrn.com/sol3/papers.cfm?abstract_id=1748604* .

Roodman, D., M. O. Nielsen, J. G. MacKinnon, and M. D. Webb. 2019. Fast and wild: Bootstrap inference in Stata using boottest. *The Stata Journal* 19 (1): 4–60.

Roychowdhury, S. 2006. Earnings Management Through Real Activities Manipulation. *Journal of Accounting and Economics* 42 (3): 335–370.

Salibian-Barrera, M. and R. H. Zamar. 2002. Bootstrapping Robust Estimates of Regression. *The Annals of Statistics* 30 (2): 556–582.

Schafhäutle, S. and D. Veenman. 2023. Crowdsourced Forecasts and the Market Reaction to Earnings Announcement News. *The Accounting Review, forthcoming* .

Shipman, J. E., Q. T. Swanquist, and R. L. Whited. 2017. Propensity Score Matching in Accounting Research. *The Accounting Review* 92 (1): 213–244.

Stock, J. and M. Watson. 2018. *Introduction to Econometrics*. Pearson, New York, NY, 4th edition edition.

Sun, L. and S. Abraham. 2021. Estimating Dynamic Treatment Effects in Event Studies with Heterogeneous Treatment Effects. *Journal of Econometrics* 225 (2): 175–199.

Thompson, S. B. 2011. Simple Formulas for Standard Errors That Cluster by Both Firm and Time. *Journal of Financial Economics* 99 (1): 1–10.

Veenman, D. 2012. Disclosures of Insider Purchases and the Valuation Implications of Past Earnings Signals. *The Accounting Review* 87 (1): 313–342.

Veenman, D. 2013. Do Managers Trade on Public or Private Information? Evidence from Fundamental Valuations. *European Accounting Review* 22 (3): 427–465.

Veenman, D. and P. Verwijmeren. 2018. Do Investors Fully Unravel Persistent Pessimism in Analysts' Earnings Forecasts? *The Accounting Review* 93 (3): 349–377.

Veenman, D. and P. Verwijmeren. 2022. The Earnings Expectations Game and the Dispersion Anomaly. *Management Science* 68 (4): 3129–3149.

Verbeek, M. 2021. *Panel Methods for Finance: A Guide to Panel Data Econometrics for Financial Applications: 1.* Berlin ; Boston.

Wooldridge, J. M. 2019. *Introductory Econometrics: A Modern Approach.* Cengage Learning, Boston, MA, 7th edition edition.