Applied Business Research

Eric Weisbrod, with collaboration from ACCT 995 Students

2025-01-22

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

```
1 + 1
```

```
[1] 2
```

1 Introduction

This is a work in process website for a potential book on applied business research in R. The goal of the website is to provide tools and examples for reproducible and well-formatted research reports.

This is an example of a Quarto citation Knuth (1984) in a sentence.

2 Project Setup

This chapter will have links and explainers on how to get your quarto and other GitHub projects going…

The top finance/accounting/aio journals are about to be flooded with highly reproducible KU research.

2.1 Git vs. GitHub

Git is the underlying code that helps manage version control of your projects. You can find more information about the details of Git here. Information about how to install git on your machine can be found here.

GitHub is a web-based user interface that makes Git easier to work with by allowing "point-and-click" version control rather than typing git commands. You will want to set up a GitHub account. The rest of this chapter will reference the use of GitHub, although everything discussed can also be accomplished through the Git language.

2.2 Language of GitHub

It is important to remember that GitHub acts as a version control interface for your research projects. So, while we will discuss the verbiage of GitHub, at its core, all that it is doing is keeping track of the changes that you make to your code. It may be helpful to translate the Git language into words you regularly associate with project management.

Repository - A repository is where all the code for a specific project lives. You can think of it like the project folder where your code is stored. The benefit of a repository is that it is stored online, allowing you to easily access it from any machine. You will most likely want to make your repository private, so that only you and people you identify as co-authors can access your code.

Fork - "Forking" a repository is equivalent to making a copy of someone else's repository. If a repository is made public, then anyone can fork the repository to have their own copy. This is unlikely to be something that will commonly occur in your own research, as we will discuss next. You can think of forking a repository as the same as copying someone else's code folder and pasting it onto your computer.

Clone - "Cloning" is where the power of GitHub really begins. Cloning a repository is the same as giving your local machine (e.g., computer) access to the code in the repository. Think of it like installing Dropbox on another computer. Now you have access to all the files stored on Dropbox. Cloning a repository is the exact same thing for code. Once you have cloned the repository, you can now work on the code from that machine. The power comes in by being able to clone the repository on multiple computers, and your coauthors doing the same, allowing you all to work on the same set of code. To clone a repository, you will need the cloning URL. To get this, go to the repository on GitHub, click the green "Code" button, and click on the copy button (two intersecting squares) next to the HTTPS URL.

Staging - Once you save the code you are working on to your local computer, a new row will appear under the "Git" tab in the top right panel of R Studio. In order to officially "save" your changes to your online GitHub repository, you need to first stage the changes. "Staging" is the same thing as "selecting" which changes you want to officially send to GitHub, which will be discussed next.

Commit - Once you have staged your changes, you are ready to begin the process of uploading them to GitHub. The first thing you need to do is to "commit" the changes. As the name implies, a "commit" means you are committed ;) to making the changes to the code. To do this, click the "Commit" button in the top right panel of R Studio. You will need to type a short message which summarizes the changes that you have made, which will be visible on GitHub so you can easily see the evolution of your code over time. Once you have typed your message, click "Commit".

Pull - "Pulling" is effectively the same as downloading the most recent version of your project's code onto your local computer. When you click "Pull" in the top right panel of R Studio, you are "pulling" the code from GitHub onto your local machine. **You should always pull before you push!**

Push - "Pushing" is the final step of saving your code through GitHub. After you have committed your changes and pulled the most recent version of the project to your machine, you can send your changes to GitHub by clicking "Push" in the top right panel of R Studio. Doing this officially sends the changes that you have made up to GitHub, effectively pushing the code from your local device onto GitHub. This can also be thought of as if you made changes to a file on your local computer and then saved that file onto Dropbox. The power of GitHub is that you can see both the old version and new version, and restore your code to older versions if you decide you don't like your changes.

2.3 Power of Projects

When you first begin working with R, one of the first things that you learn are the benefits of storing/saving code in R scripts. In theory, we could always write all of our code in the terminal, and just rewrite it every time we wanted to run a command (gross!). Using R scripts allows us to save our code, which can be run the exact same way every time. In a nutshell, what the R script is really doing is making our code more reproducible, as we can be sure that we are always running the same commands.

The next evolution of your R journey probably went something like this… You realized that trying to code up an entire research project in one R script was burdensome. There are thousands of lines of code, some of which you do not want to run every time. So, you break up your project into multiple R scripts, with each one serving a different function (e.g., data download, data cleaning, analysis, figures, etc.). While this likely initially occurred just by saving the different scripts in the Dropbox folder where you stored your project files (code, data, paper drafts, etc.), we will next discuss how we can integrate GitHub to help make working on a research project easier.

Projects in R can be thought of as self-contained folders, where all your code and associated documents can live. The power of these projects is that they can be integrated with GitHub, making your work more reproducible by keeping all your code for a project in one place along with the version history.[1] To start a new project, you will first create a repository on GitHub with this project's name. You then go into R Studio and click "File -> New Project". From there, you will click on "Version Control -> Git". This brings you to a screen which will allow you to clone (see above if this is unfamiliar) your GitHub repository to your local machine. You will copy and paste the URL from the repository into the "Repository URL" text box. The "Project directory name:" box will populate with a location on your local computer. This is where the project and subsequent R scripts will live. How you manage these comes down to personal preference, but a common way to store GitHub projects locally is to create a "git" folder on your computer's C-drive and then store each project in the git folder. Once you have set the project directory, you can select "Create Project" to create the local version of the repository on your computer.

The key thing to notice (and one of the many powers of working within projects) is located near the top right corner of R Studio. You will now see a blue box with the letter "R" inside of it, which is the image of an R project. Next to this, you will see the name of the project you just created, indicating that you are working "inside" this project. When you look at the "Files" in the bottom right panel, you will see that the directory has been changed to the location of your R project. This allows you to see all the scripts associated with this project, and quickly change between scripts. Importantly, any pushes that you make to GitHub will be pushed to that specific project.

[1] Projects and repositories are linked together. A project lives in R and is an R object, while a repository lives on GitHub. When you read "repository" think "project" or vice versa, however you prefer.

One of the nice features of working in projects is the ability to quickly access your project code and switch between projects. For example, if you were conducting a research project using just R scripts, you would need to save them somewhere (e.g., Dropbox) and navigate to that folder to open/access them. When using an R project, rather than navigating to the folder on your computer that contains your code, you can just open R Studio and click on the R project button in the top right corner (blue cube with "R" inside). This will open up a drop down menu of all the projects you have on your computer, allowing you to open the one you want. When you close the project, whatever R scripts that you have open at the time you close will automatically open again the next time you open the project.

2.4 Reproducible Workflow

All of the things we have discussed up to this point have been about making our code and research projects more reproducible and easier to understand. Beyond these tools, it is important to remember that a truly reproducible research project means that someone can take your project and with limited/no prior knowledge about what you were doing, replicate your findings and understand your code. To make this possible, it is important to try and make your code as clear and concise as possible, including plenty of documentation and comments in the code explaining what is happening. One of the best ways that I have found to keep my workflow easy to track is through developing a consistent naming convention between R scripts and data files.

When creating my R scripts, I always begin the script name with a number, beginning with "00". This is then followed by "-short-description.R" where the "short-description" is a short description of what the code is doing. The scripts are numbered based on when in the sequence they should be run. So, you would run the "00" scripts first, then "01", "02", and so on. Importantly, any data that I save from a script is named beginning with that script's numeric identifier. For example, if I save a data set called "new_data.parquet" from an R script titled "02-create-new-data.R", then I would save the parquet file with the name "02_new_data.parquet" in my Dropbox data folder. Doing this helps along two dimensions. First, it makes your data folder much more organized, with the data ordered in the chronological order that it was generated by your code. Second (and most importantly) it makes it immediately obvious which script generated the data set. That way, if someone has questions about a particular data set, you can immediately know what R script generated the data.

Below is an example project setup with the naming convention. The first level of bullet points are the R scripts (which would be located in the git folder) and the second level are the data sets (which would be saved on Dropbox).

- 00-Global-Parameters.R
- 01-Download-WRDS-Data.R
    - 01_compustat_data_raw.parquet

- 02-Create-Master-Data.R
  - 02_master_data.parquet
- 03-Main-Analyses.R
  - 03.01_table_1.tex
  - 03.02_table_2.tex
- 04-Figures.R
  - 04.01_figure_1.jpeg
  - 04.02_figure_2.jpeg

3 Obtaining and Merging Data

This is my first time working on a Quarto book. So, this first post will be very rough for now. I will try to provide a few different examples of ways to obtain and merge data in R, and a few tips of things to keep in mind.

We already know how to obtain data from WRDS. Let's use this to obtain some returns for the S&P 500. We could use the formal index data, but let's take a shortcut and just use the popular SPY ETF that tracks the S&P 500. To do this, we need to find the CRSP identifier (PERMNO) for the ticker "SPY." We can look in the WRDS stocknames file for this, and then use the SPY PERMNO to pull data from the CRSP monthly stock file.

```r
# Load Libraries [i.e., packages]
library(dbplyr)
library(RPostgres)
library(DBI)
library(glue)
library(arrow)
library(haven)
library(tictoc) #very optional timer, mostly as a teaching example
library(tidyverse) # I like to load tidyverse last to avoid package conflicts

#I have done this in a separate chunk with the options
# results: FALSE
# message: FALSE
#because I don't need to see the messages from loading the packages.
```

```r
# Log in to WRDS --------------------------------------------------------------------

#before running this block, I used these commands to securely store my WRDS username and pass
# keyring::key_set("WRDS_user")
# keyring::key_set("WRDS_pw")

if(exists("wrds")){
  dbDisconnect(wrds)  # because otherwise WRDS might time out
}
```

```r
wrds <- dbConnect(Postgres(),
                  host='wrds-pgdata.wharton.upenn.edu',
                  port=9737,
                  user=keyring::key_get("WRDS_user"),
                  password=keyring::key_get("WRDS_pw"),
                  sslmode='require',
                  dbname='wrds')


# Create WRDS Table References --------------------------------------------------
crsp.msf <- tbl(wrds,in_schema("crsp","msf"))
stocknames <- tbl(wrds,in_schema("crsp","stocknames"))

#I am collecting this data locally to play with duplicates
spy_permnos <- stocknames |> filter(ticker == "SPY") |> collect()
```

Notice that there are six observations in the stocknames table that all share the same ticker "SPY." I am going to use this as a toy example to play with duplicates. My goal is for this data to be unique at the level of ticker-permno links. First, I can check whether this is true.

```r
#check whether there are duplicates
#this simple logic is useful in general
#group by the level I want to make unique,
#count within each group
#sort by descending count so that if there are duplicates
#they will show up at the top.
spy_permnos |>
  group_by(ticker,permno) |>
  count() |>
  arrange(-n)
```

```
# A tibble: 3 x 3
# Groups:   ticker, permno [3]
  ticker permno     n
  <chr>   <int> <int>
1 SPY     84398     3
2 SPY     33910     1
3 SPY     60716     1
```

There are multiple permnos connected to the SPY ticker and some duplicate entries for permno 84398 so I better just look at the data. Also this tells me that there are only a few rows so it doesn't hurt to just print the data.

```
#| #note that we can use the kable commmand to embed a simple table in the quarto document
knitr::kable(spy_permnos)
```

permno	namedt	nameendt	shrcd	exchcd	siccd	ncusip	ticker	comnam	shrcls	permco	hexcd	cusip	st_date	end_date	namedum
33910	1962-07-02	1966-05-24	10	2	2893	NA	SPY	SPEEDRY CHEMICAL PRODS INC	A	2751	3	55914210	1962-07-02	1979-01-22	2
60716	1978-10-03	1987-07-01	10	1	3811	84756710	SPY	SPECTRA PHYSICS INC	NA	4215	1	84756710	1972-12-14	1987-07-01	2
84398	1993-01-29	2009-02-23	73	2	6726	78462F10	SPY	SPDR TRUST	NA	46699	4	78462F10	1993-01-29	2024-12-31	2
84398	2009-02-24	2010-01-26	73	4	6726	78462F10	SPY	SPDR TRUST	NA	46699	4	78462F10	1993-01-29	2024-12-31	2
84398	2010-01-27	2024-12-31	73	4	6726	78462F10	SPY	SPDR S & P 500 E T F TRUST	NA	46699	4	78462F10	1993-01-29	2024-12-31	2

Looking at the data, the company name for permno 84398 matches the SPDR S&P 500 ETF I am looking for. It looks like the duplicate entries might have to do with a change in the listing exchange for the ETF (exchcd) and then a slight name change in 2010 to make the name of the trust more descriptive. Let's keep using this toy example to demonstrate some other functions for dealing with duplicates:

```
#if I want to just collapse the duplicates, I can use "distinct" across the groups that I ca

spy_permnos |>
  select(ticker,permno) |>
  distinct()
```

```
# A tibble: 3 x 2
  ticker permno
  <chr>   <int>
1 SPY     33910
2 SPY     60716
3 SPY     84398
```

Now there are only three observations,which is what I asked for, but sometimes it might matter which of the duplicate observations I keep. For example, perhaps what I should do is keep the most recent observation from the spy_permno dataset, in terms of nameenddt.

```
#select the max data within each group as more advanced way to keep one obs per
#group
spy_permnos |>
  group_by(ticker,permno) |>
  filter(nameenddt==max(nameenddt))
```

```
# A tibble: 3 x 16
# Groups:   ticker, permno [3]
  permno namedt     nameenddt  shrcd exchcd siccd ncusip   ticker comnam  shrcls
   <int> <date>     <date>     <int>  <int> <int> <chr>    <chr>  <chr>   <chr>
1  33910 1962-07-02 1966-05-24    10      2  2893 <NA>     SPY    SPEEDR~ A
2  60716 1978-10-03 1987-07-01    10      1  3811 84756710 SPY    SPECTR~ <NA>
3  84398 2010-01-27 2024-12-31    73      4  6726 78462F10 SPY    SPDR S~ <NA>
# i 6 more variables: permco <int>, hexcd <int>, cusip <chr>, st_date <date>,
#   end_date <date>, namedum <dbl>
```

```
#ultimately we can assign the permno of the current observation, which we already know from 

spy_permno <- spy_permnos |>
  group_by(ticker,permno) |>
  filter(nameenddt==max(nameenddt)) |>
  ungroup() |>
  filter(nameenddt==max(nameenddt)) |>
  select(permno) |>
  as.numeric()

spy_permno
```

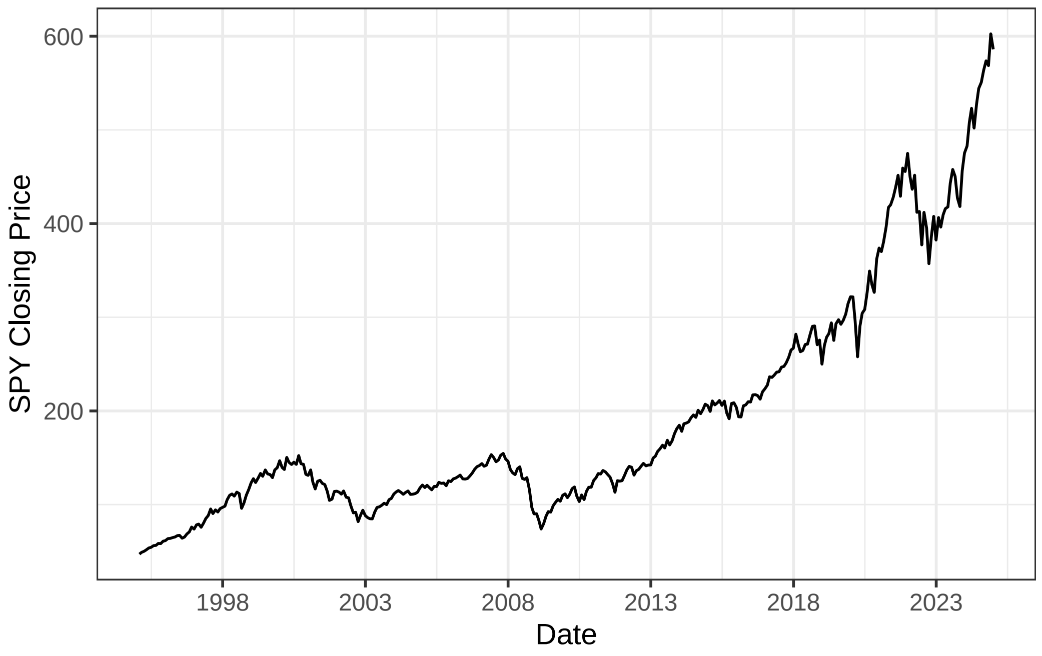
```
[1] 84398
```

Now we can use the SPY permno to pull monthly returns for SPY:

```
# Pull CRSP MSI Data ------------------------------------------------------------

#Data seems to begin in feb 1993, lets start in 1995 as a nice round number
#notice that this implicitly feeds the permno I calculated locally back up to WRDS in my crs
mkt_index <- crsp.msf |>
```

```
  filter(date >= "1995-01-01",
         permno == spy_permno) |>
  select(date,ret,prc) |>
  collect() |>
  mutate(month = month(date),
         year = year(date))
```

Then I can plot them, note that if you look at the source code for this page, I do this in a chunk with echo=false so that I only see the output and not the code. This would be useful for creating an actual paper rather than coding examples:



This plot would look nice with recessions shaded. We can get recession dates from FRED. FRED data can be accessed from an API, there is a custom package to work with FRED data in R called fredr. First you need to obtain a FRED API key by signing up here: https://fred.stlouisfed.org/docs/api/api_key.html

```
#load the fredr package
library(fredr)

#Unblock the below and run to set your password
#keyring::key_set("fred_api_key")

#set my API key which is saved in keyring
```

```r
fredr_set_key(keyring::key_get("fred_api_key"))

#collect the data from the series USRECD
# https://fred.stlouisfed.org/series/USRECD

fred_data<-fredr(series_id = "USRECD",
                 observation_start = as.Date("1995-01-01"),
                 observation_end = as.Date("2024-12-31"),
                 frequency = "m") |>
  #I am going to add month and year variables because I think this is
  #easier for linking
  mutate(month = month(date),
         year = year(date))

# show the first few rows which has a value of 0 or 1 where 1 is recession
fred_data |> head() |> knitr::kable()
```

date	series_id	value	realtime_start	realtime_end	month	year
1995-01-01	USRECD	0	2025-03-19	2025-03-19	1	1995
1995-02-01	USRECD	0	2025-03-19	2025-03-19	2	1995
1995-03-01	USRECD	0	2025-03-19	2025-03-19	3	1995
1995-04-01	USRECD	0	2025-03-19	2025-03-19	4	1995
1995-05-01	USRECD	0	2025-03-19	2025-03-19	5	1995
1995-06-01	USRECD	0	2025-03-19	2025-03-19	6	1995

Now we need to merge the SPY data with the recession data.

```r
merged_data <- mkt_index |>
  #I am going to select only the columns I need from   #the FRED data
  inner_join(fred_data |>
               select(month,year,recession=value),
             by=join_by(month,year))

# check to make sure it is still unique by month
merged_data |>
  group_by(month,year) |>
  count() |>
  arrange(-n)
```

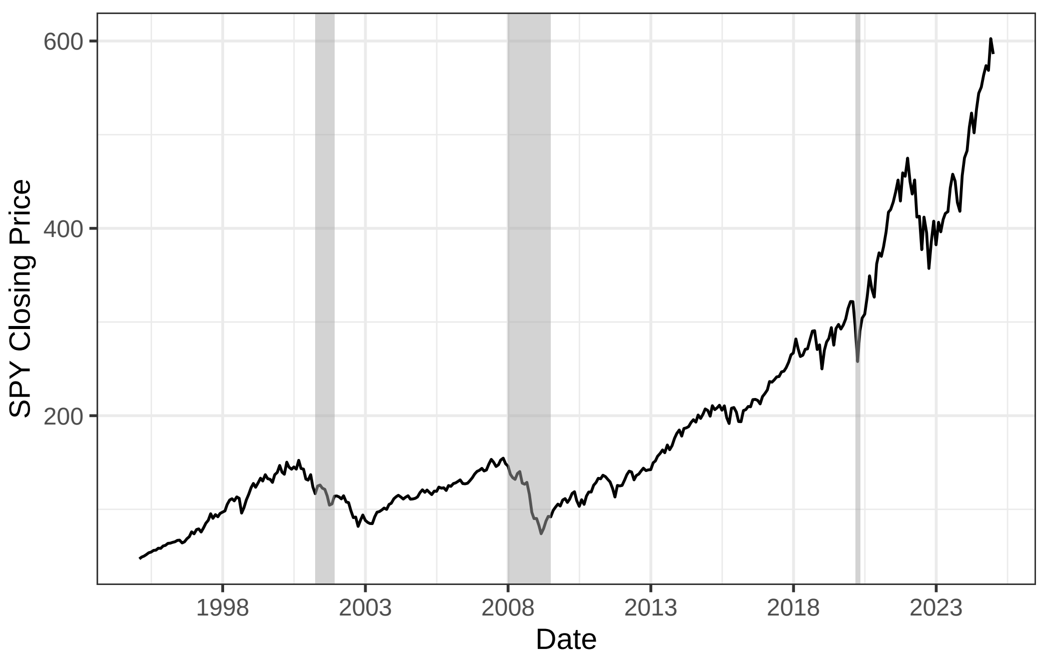
```
# A tibble: 360 x 3
# Groups:   month, year [360]
   month  year     n
   <dbl> <dbl> <int>
 1     1  1995     1
 2     1  1996     1
 3     1  1997     1
 4     1  1998     1
 5     1  1999     1
 6     1  2000     1
 7     1  2001     1
 8     1  2002     1
 9     1  2003     1
10     1  2004     1
# i 350 more rows
```

Now we can make the plot with shades for recession months

```
#turns out the merged data was not the preferred way to do this kind of plot




#here is some code I found online to reshape the recession data and add it to the plot

#rename/assign fred data to recession because
#that was the name in the example I found
recession<-fred_data

#load a package they used
library(ecm)

#reshape the recession data for the way
#geom_rect likes the data shaped
recession$diff<-recession$value-lagpad(recession$value,k=1)
  recession<-recession[!is.na(recession$diff),]
  recession.start<-recession[recession$diff==1,]$date
  recession.end<-recession[recession$diff==(-1),]$date

  if(length(recession.start)>length(recession.end))
  {recession.end<-c(recession.end,Sys.Date())}
  if(length(recession.end)>length(recession.start))
  {recession.start<-c(min(recession$date),recession.start)}
  recs<-as.data.frame(cbind(recession.start,recession.end))
  recs$recession.start<-as.Date(as.numeric(recs$recession.start),origin=as.Date("1970-01-01")
  recs$recession.end<-as.Date(recs$recession.end,origin=as.Date("1970-01-01"))

#look at the reshaped data
recs
```

```
  recession.start recession.end
1      2001-04-01    2001-12-01
2      2008-01-01    2009-07-01
3      2020-03-01    2020-05-01
```

```
#plot the new plot with recession bars
merged_data |>
  ggplot(aes(x=date,y=abs(prc))) +
  geom_line() +
  scale_x_date(name = "Date",
               date_breaks= "5 years",
               date_labels = "%Y") +
  scale_y_continuous(name = "SPY Closing Price") +
  geom_rect(data=recs, inherit.aes=F,
                       aes(xmin=recession.start, xmax=recession.end, ymin=-Inf, ymax=+Inf)
                fill="darkgrey", alpha=0.5)+
  theme_bw()
```



SPY Closing Price
600
400
200
1998
2003
2008
2013
2018
2023
Date

4 Merging Data (In Progress)

This chapter will go over how to merge in data from various sources. This will include examples of how to merge in firm-level data from various sources, and time-series data from FRED/Fama-French Factors/CRSP Index Files.

4.1 Common Sources of Firm-Level Data

Below is a table of common sources and firm identifiers that allow researchers to link data from different data sources:

Data Source	Identifiers	Other Firm Identifiers	Can Be Linked To	Notes
Compustat	GVKEY	Ticker, CIK	CRSP, Audit Analytics	
CRSP	PERMNO	Ticker, CUSIP	Compustat, IBES	To link to Compustat data, researchers should use the CRSP-Compustat link file for PERMNO-GVKEY mapping.
I/B/E/S	TICKER	OFTIC, CUSIP	CRSP	The "TICKER" variable is the I/B/E/S firm identifier **not** the trading symbol. The trading symbol is the "OFTIC" variable.
TAQ	SYMBOL			

Data Source	Identifiers	Other Firm Identifiers	Can Be Linked To	Notes
TRACE	CUSIP		CRSP	
Audit Analytics	CIK		Compustat	
XBRL	CIK		Compustat	
RavenPack	RP_ENTITY_ID	CUSIP	CRSP, IBES	

Blocking out for next attempt

5 Regression Tables

Test of embedding a regression in Quarto.

Table 5.1

	Base	No FE	Year FE	Two-Way FE	With Controls
ROA_t	0.839***	0.756***	0.769***	0.639***	0.624***
	(62.732)	(48.155)	(48.621)	(38.634)	(35.596)
$LOSS$		-0.030***	-0.028***	-0.015***	-0.017***
		(-7.949)	(-7.755)	(-7.556)	(-8.111)
$ROA_t \times LOSS$		0.032	0.012	-0.285***	-0.294***
		(1.470)	(0.535)	(-13.307)	(-12.620)
Year FE			X	X	X
Firm FE				X	X
Controls					X
N	163,298	163,298	163,298	161,635	161,635
R^2	0.594	0.597	0.603	0.707	0.707
R^2 Within			0.580	0.184	0.186

6 Summary

In summary, this book has no content whatsoever.

```
1 + 1
```

```
[1] 2
```

References

Knuth, Donald E. 1984. "Literate Programming." *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.