Working with R (v1)

Taylor G. White

February 24, 2019

Contents

Tutorials

This tutorial includes both an aggregation of already available learning materials for the R language as well as real-world examples of R in use, aimed at business analysts.

Get Started

Download R and RStudio

- R
- RStudio

Syntax, objects, functions

What are the basic rules of the language? How are objects created? How do functions work? What controls the flow of operations that are completed?

- Good overall introduction
- Data types
- Scripting style guide

Exploratory Data Analysis (EDA)

Import data, take a look at it, compute basic summary statitics, filter/reshape your data, plot your data.

- Data import
- Basic summary statistics
- Filter/subset, data reshape, groupwise operations
- More dplyr examples
- Plotting with ggplot2

Advanced manipulations

Combine data from multiple sources for analysis. Clean your data for analysis, reshape it for different operations.

- Combine data with joins
- Tidying and reshaping data

Statistical models

These methods are for fitting a given model to a set of data. These models are good for explaining relationships between quantities of interest and can be used for prediction. Each model is built for a particular purpose and they will not perform well if the assumptions underlying the model are violated.

- Linear regression - Fit a linear model via ordinary least squares.
- Logistic models - Discrete events (e.g. something happened or didn't happen)
- Poisson models - Counting processes
- Time series models - Follow a single entity over time. Observations are correlated over time.
- panel data analysis - for data sets that follow entities (e.g. people, companies, countries) over time.
- quantile regression - OLS regression will find the conditional mean – this method can find the conditional Nth percentile, e.g. 10% percentile, median, or 90th percentile.
- Advanced parametric models

Machine learning

These methods can help explain unseen relationships in data that parametric models might not capture. These methods are better for prediction than explanation, and often outperform parametric models in terms of prediction. The downside is you might not be able to explain your results to stakeholders, as many machine learning methods are black boxes.

- Cluster analysis - create clusters of entities based on a host of variables
- Regression and classification trees - Split data into homogeneous groups and continue doing so until a split no longer produces meaningful differences.
- Neural networks
- Random forests - In parallel, fit many regression trees on bootstrapped input datasets, using a subset of variables each time. Combine the results.
- Boosted trees - In sequence, fit many small regression trees (not "full" trees with many splits) on bootstrapped input datasets. Combine the results.

R Gotchas

Something unexpected is happening!

- You ran a function and it is giving you strange output. Make sure that the `class()` of the object(s) you are operating on are what you expected and that it has the data you expect data.
- Inspect your data. Print it to the console, look at the top and bottom of it. Use plots to visualize your whole dataset in various ways. Make sure you aren't making incorrect assumptions about your data.
- You changed your code in some way, and you are referring to something other than what you expected. Good and consistent naming conventions can help with these problems.

```
# original code
oldthing = 1

some_important_value = oldthing^2

# updated code
oldThing = 2

some_important_value = oldthing^2 # still returns 1 if you didn't clear your environment!
```

This code worked the first time but it isn't working now

You are likely modifying an object in place. This is usually bad and leads to strange errors. The value of the object `thing` is dependent on how many times you ran the `thing = thing + 2` line. This error is unlikely in this case, but it can happen with longer data prep code chunks.

```
# This is modifying the object in place
thing = 1

thing = thing + 2
```

I don't understand that error!

Stackoverflow and Google are your friends. Copy the whole error and paste it into google. See what comes up. If nothing comes up, copy just the first part that isn't particular to your piece of code and see if a search engine finds something. Chances are, you have made a mistake someone else has made many times before.

Nothing is working and I'm freaking out!

Relax. Take a break. Walk away from the code for a bit and come back to it when you have regained composure. Chances are, frantically changing bits of code will lead to even more problems. The way to uncover bugs is to isolate the changes you make in order to systematically test each chunk for issues. Frantic changes will mean you could fix one problem and introduce many others, leading you to believe the whole piece of code is broken in some fundamental way.

I have a lot to do and don't know how to proceed!

Think before you write any code. Go to a piece of paper or whiteboard and map out the problem you are working on. When you have a decent grasp, write code in a piecewise fashion, instead of trying to

tackle everything at once. Small, manageable code pieces that work can be wrapped in functions and pieced together. If there is an error, it's easy to trace that error to a particular function, instead of having to debug a large piece of spaghetti code.

Be careful with `factor` vectors

Factors are essentially vectors that have some sort of label, saved as an integer, treated as a discrete value. Some processes will coerce factors back to integer values, which then would allow for numeric operations. Make sure weirdness isn't happening because of factors.

Example 1 - Download stock prices, estimate beta

This example demonstrates how to get publicly available data to estimate betas for a set of stock tickers.

Steps

- Get prices for multiple stocks: Chevron, ExxonMobile, First Solar Inc., GE, Tesla
- Get values of the S&P 500
- Get values of the risk free interest rate
- Compute monthly returns for all of these series
- Combine data into a single dataset for analysis
- Subtract the risk free rate from asset and market returns
- Regress stock returns in excess of the risk free rate from market returns in excess of the risk free rate
- Extract betas from each model

```
# Set up your script

# load libraries
library(quantmod) # download data and other financial functions
```

```
## Loading required package: xts
```

```
## Loading required package: zoo
```

```
##
## Attaching package: 'zoo'
```

```
## The following objects are masked from 'package:base':
##
##     as.Date, as.Date.numeric
```

```
## Loading required package: TTR
```

```
## Version 0.4-0 included new data defaults. See ?getSymbols.
```

```
library(tidyverse) # host of data handling and visualization functions
```

```
## -- Attaching packages ---------------------------------------------------------------------------
```

```
## v ggplot2 3.1.0       v purrr   0.3.0
## v tibble  2.0.1       v dplyr   0.8.0.1
## v tidyr   0.8.2       v stringr 1.4.0
## v readr   1.3.1       v forcats 0.4.0

## -- Conflicts ------------------------------------------------------------------------------------------
## x dplyr::filter() masks stats::filter()
## x dplyr::first()  masks xts::first()
## x dplyr::lag()    masks stats::lag()
## x dplyr::last()   masks xts::last()
```

```
library(lubridate) # easier handling of dates
```

```
##
## Attaching package: 'lubridate'

## The following object is masked from 'package:base':
##
##     date
```

```
library(scales) # nice scales for ggplot2
```

```
##
## Attaching package: 'scales'

## The following object is masked from 'package:purrr':
##
##     discard

## The following object is masked from 'package:readr':
##
##     col_factor
```

```
# assign vectors of symbols to objects to be passed to quantmod's getSymbols() function
stock_tickers_vec = c('CVX', 'XOM', 'FSLR', 'GE', 'TSLA')
```

My intention is to run the same data preparation code for all of the symbols, so it wouldn't matter if I had to do this process for 1 or 10,000 stock tickers. That being said, I always write out my code for the single case to test each piece before wrapping it into a function.

```
# download daily data for the S&P500, from 2010 to the current date using Yahoo finance data
downloaded_data = getSymbols('^GSPC', auto.assign = F, src = 'yahoo', from = as.Date('2010-01-01'))
```

```
## 'getSymbols' currently uses auto.assign=TRUE by default, but will
## use auto.assign=FALSE in 0.5-0. You will still be able to use
## 'loadSymbols' to automatically load data. getOption("getSymbols.env")
## and getOption("getSymbols.auto.assign") will still be checked for
## alternate defaults.
##
## This message is shown once per session and may be disabled by setting
## options("getSymbols.warning4.0"=FALSE). See ?getSymbols for details.
```

```
##
## WARNING: There have been significant changes to Yahoo Finance data.
## Please see the Warning section of '?getSymbols.yahoo' for details.
##
## This message is shown once per session and may be disabled by setting
## options("getSymbols.yahoo.warning"=FALSE).
```

```
# take a look at the data
class(downloaded_data) # what type of object was downloaded?
```

```
## [1] "xts" "zoo"
```

```
head(downloaded_data)
```

```
##            GSPC.Open GSPC.High GSPC.Low GSPC.Close GSPC.Volume
## 2010-01-04   1116.56   1133.87  1116.56    1132.99  3991400000
## 2010-01-05   1132.66   1136.63  1129.66    1136.52  2491020000
## 2010-01-06   1135.71   1139.19  1133.95    1137.14  4972660000
## 2010-01-07   1136.27   1142.46  1131.32    1141.69  5270680000
## 2010-01-08   1140.52   1145.39  1136.22    1144.98  4389590000
## 2010-01-11   1145.96   1149.74  1142.02    1146.98  4255780000
##            GSPC.Adjusted
## 2010-01-04       1132.99
## 2010-01-05       1136.52
## 2010-01-06       1137.14
## 2010-01-07       1141.69
## 2010-01-08       1144.98
## 2010-01-11       1146.98
```

```
tail(downloaded_data)
```

```
##            GSPC.Open GSPC.High GSPC.Low GSPC.Close GSPC.Volume
## 2019-02-20   2779.05   2789.88  2774.06    2784.70  3835450000
## 2019-02-21   2780.24   2781.58  2764.55    2774.88  3559710000
## 2019-02-22   2780.67   2794.20  2779.11    2792.67  3427810000
## 2019-02-25   2804.35   2813.49  2794.99    2796.11  3804380000
## 2019-02-26   2792.36   2803.12  2789.47    2793.90  3645680000
## 2019-02-27   2787.50   2795.76  2775.13    2792.38  3767130000
##            GSPC.Adjusted
## 2019-02-20       2784.70
## 2019-02-21       2774.88
## 2019-02-22       2792.67
## 2019-02-25       2796.11
## 2019-02-26       2793.90
## 2019-02-27       2792.38
```

```
# get the class of all the columns. This will loop over each column and apply the class function to tha
sapply(downloaded_data, class)
```

```
##      GSPC.Open GSPC.High GSPC.Low GSPC.Close GSPC.Volume GSPC.Adjusted
## [1,] "xts"     "xts"     "xts"    "xts"      "xts"       "xts"
## [2,] "zoo"     "zoo"     "zoo"    "zoo"      "zoo"       "zoo"
```

```r
# Calculate monthly returns. Make sure you know what the underlying values are. In some cases, it's sto
monthly_returns = monthlyReturn(downloaded_data)
head(monthly_returns)
```

```
##            monthly.returns
## 2010-01-29     -0.03823356
## 2010-02-26      0.02851369
## 2010-03-31      0.05879643
## 2010-04-30      0.01475923
## 2010-05-28     -0.08197584
## 2010-06-30     -0.05388244
```

```r
# convert the monthly return to a data.frame and add a date column using the index of the zoo/xts objec
returns_df = data.frame(
  monthly_returns
) %>%
  rename(
    return = monthly.returns # make a prettier name for the monthly.returns column
  ) %>%
  mutate(
    # this will shift each date to the start of the month for simplicity of comparison
    date = floor_date(index(monthly_returns), 'month')
  )

head(returns_df) # everything looks good.
```

```
##        return       date
## 1 -0.03823356 2010-01-01
## 2  0.02851369 2010-02-01
## 3  0.05879643 2010-03-01
## 4  0.01475923 2010-04-01
## 5 -0.08197584 2010-05-01
## 6 -0.05388244 2010-06-01
```

Now I want to "wrap" this code into a function, so I can run the same code efficiently without having to repeat my code many times, which will cause errors and headaches.

```r
# create the name for your function. Use a verb to describe it
# The only thing I need to do to get data for other tickers is to change the ticker that is fed through
# This is captured in the first  parameter defined in the get_monthly_returns_function
# The second parameter is from_date, which will determine how far back the data goes
# This parameter has a default value, set to January 1, 2010
# The third parameter is passed to getSymbols and indicates the source of the data. The default is set

get_monthly_returns = function(ticker, from_date = as.Date('2010-01-01'), source = 'yahoo') {


  # FIRST CHANGE - sp500 is changed to ticker here, so other tickers can be fed through
  # I also changed the hard-coded from date to from_date, which allows for control of the
  # downloaded dates
  downloaded_data = getSymbols(ticker, auto.assign = F, src = source, from = from_date)
```

```
  # SECOND CHANGE -  no need to take a look at the data within the function anymore

  # calculate monthly returns
  monthly_returns = monthlyReturn(downloaded_data)

  # convert the monthly return to a data.frame and add a date column using the index of the zoo/xts obj
  returns_df = data.frame(
    monthly_returns
  ) %>%
    rename(
      return = monthly.returns # make a prettier name for the monthly.returns column
    ) %>%
    mutate(
      # this will shift each date to the start of the month for simplicity of comparison
      date = floor_date(index(monthly_returns), 'month'),

      # !!!!!
      # THIRD CHANGE -- I'm adding the ticker to the returns_df, to help keep track of
      # which data I downloaded. This is super importantant, and helps make sure you know what data
      # you have.
      ticker = ticker
    )

  return(returns_df) # this is what the function gives us or "returns"
}
```

Did you notice the %>% operator in the code chunk above? That is called a pipe, and it helps make super clean code. It takes the results of one operation (function call) and "pipes" those results to the first argument of the next function. This allows code to be chained together linearly, from one step to the next. This stops a never ending series of parentheses.

Now I have this nice function that can download daily values of data using yahoo finance. How can I use it?

Download data for one ticker, Tesla:

```
tesla_returns = get_monthly_returns('TSLA')
```

What about all the tickers? I want to download all of the data and "stack" the returns for efficient analysis. I will feed all of the stock tickers to this function I created and combine the results.

```
# lapply will loop over an input vector, feeding each element to the first argument of a function
# it will return results as a list, which is a handy type of object that can hold anything you like

all_monthly_stock_returns = lapply(stock_tickers_vec, get_monthly_returns)

# all_monthly_stock_returns is a list. How can you access data within a list?
class(all_monthly_stock_returns)
```

```
## [1] "list"
```

```
# look at the data in the first element
all_monthly_stock_returns[[1]] %>% head()
```

```
##        return       date ticker
## 1 -0.07774929 2010-01-01    CVX
## 2  0.00249584 2010-02-01    CVX
## 3  0.04882433 2010-03-01    CVX
## 4  0.07398127 2010-04-01    CVX
## 5 -0.09295185 2010-05-01    CVX
## 6 -0.08135917 2010-06-01    CVX
```

Each element of the `all_monthly_stock_returns` list is the `data.frame` returned by `get_monthly_returns`. How can this data be converted to a `data.frame` for analysis?

```
# bind rows will stack data.frame objects on top of each other. It accepts a series of data.frame objec
all_monthly_stock_returns_stacked = bind_rows(all_monthly_stock_returns)

# take a look at the stacked data
head(all_monthly_stock_returns_stacked)
```

```
##        return       date ticker
## 1 -0.07774929 2010-01-01    CVX
## 2  0.00249584 2010-02-01    CVX
## 3  0.04882433 2010-03-01    CVX
## 4  0.07398127 2010-04-01    CVX
## 5 -0.09295185 2010-05-01    CVX
## 6 -0.08135917 2010-06-01    CVX
```

```
tail(all_monthly_stock_returns_stacked)
```

```
##          return       date ticker
## 540 -0.12229004 2018-09-01   TSLA
## 541  0.27401149 2018-10-01   TSLA
## 542  0.03901341 2018-11-01   TSLA
## 543 -0.05044517 2018-12-01   TSLA
## 544 -0.07746394 2019-01-01   TSLA
## 545  0.02514495 2019-02-01   TSLA
```

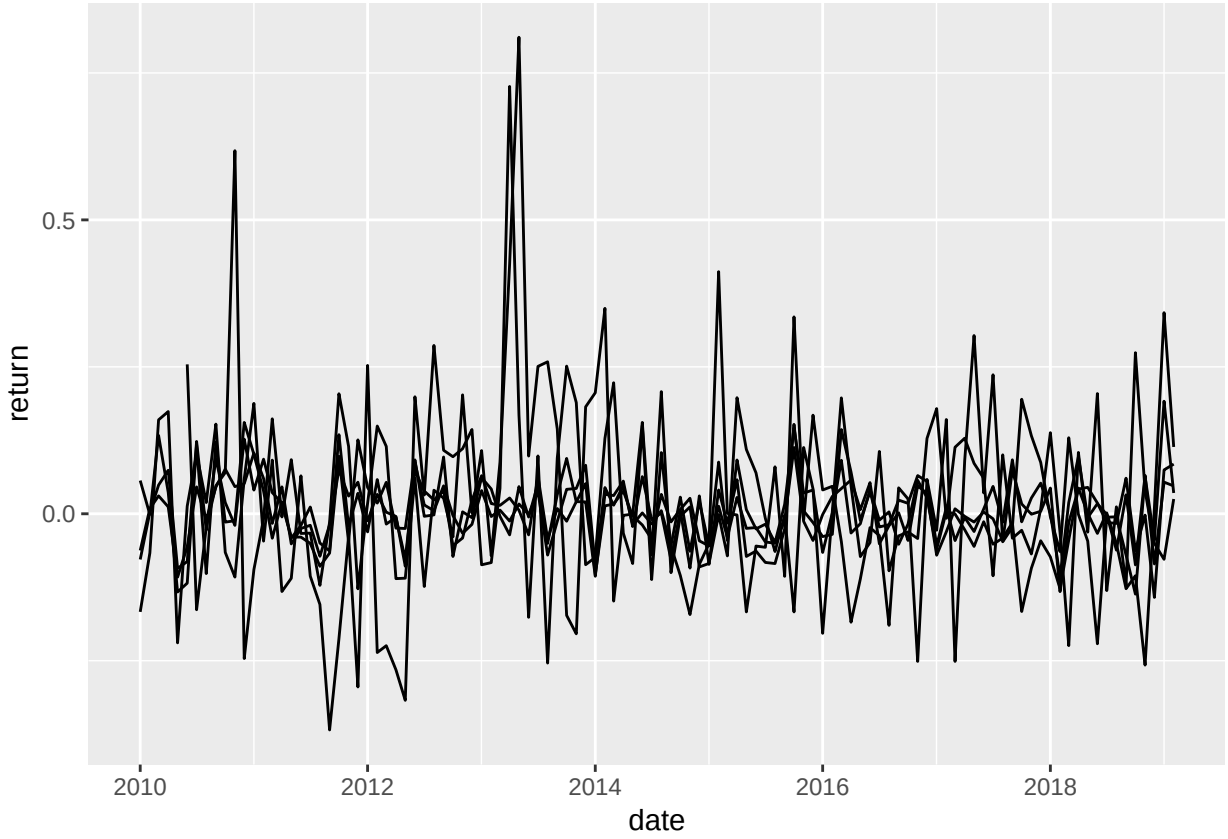
```
# look at the counts for each ticker
table(all_monthly_stock_returns_stacked$ticker)
```

```
##
##  CVX FSLR   GE TSLA  XOM
##  110  110  110  105  110
```

Now that we have all the stock data combined, we can do additional analysis on the data. I like to start by making plots of the data I have, which helps me understand the data I just downloaded, both in terms of how variables relate to each other, and in terms of accuracy (strangeness in data comes out very quickly when you plot all of it).

```
# ggplot is from the ggplot2 package, a (the) premier library for static plots

# the ggplot function sets up the "rules" for the plot.
# The first argument is the data.frame to be used
```

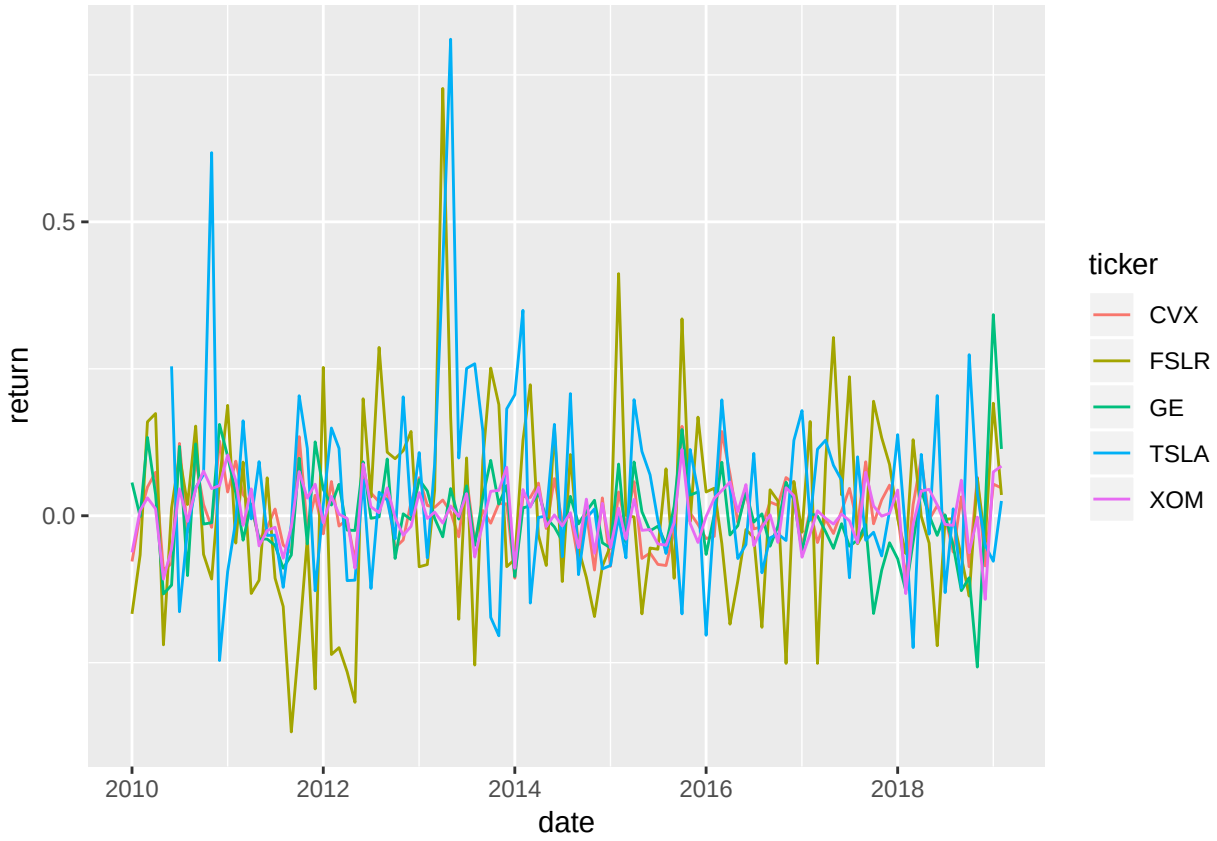
```
# the next argument is aes(), which stands for aesthetics. This says which values should be passed
# on to other "geom_" functions, or layers of the plot for use
# here, date is the x variable, return is the y variable, and the data is plotted by group, using ticke

ggplot(data = all_monthly_stock_returns_stacked, aes(x = date, y = return, group = ticker)) +
  geom_line()
```



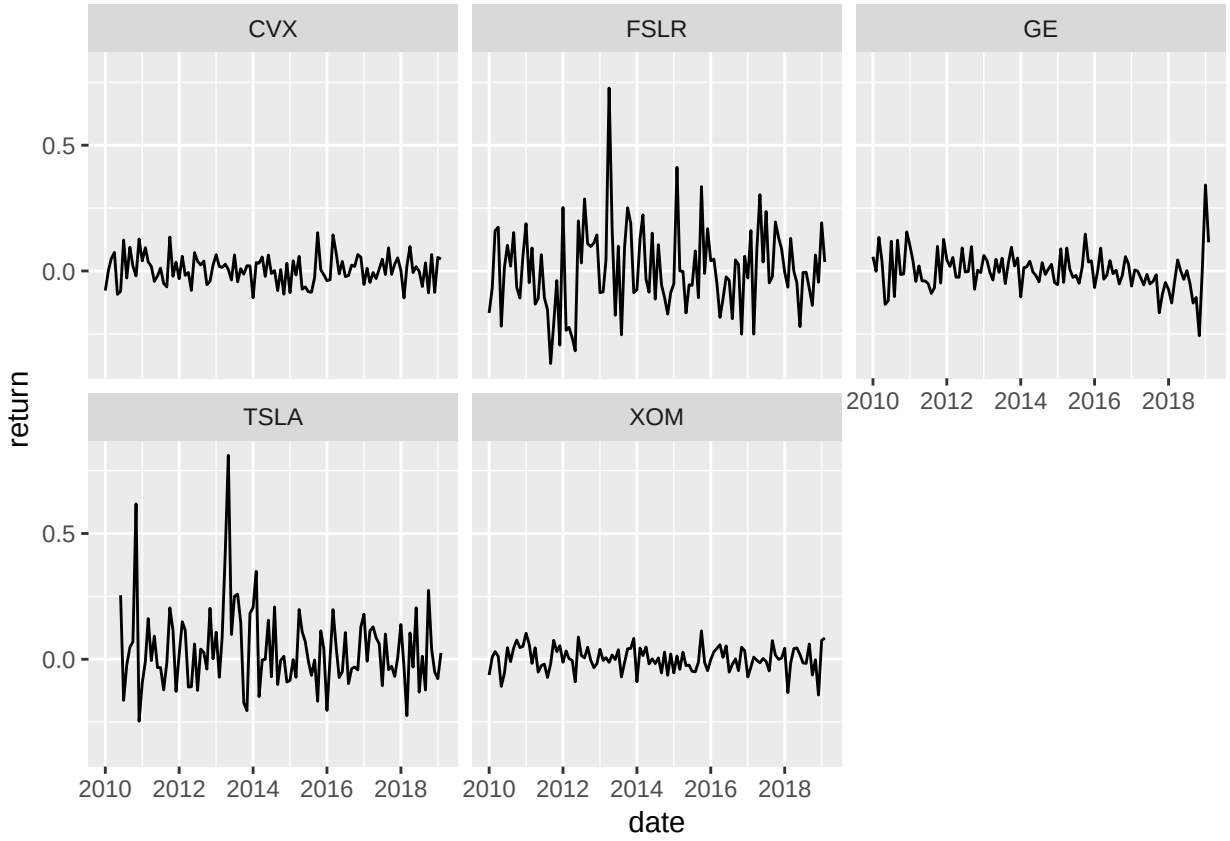
This plot shows one line for each stock ticker that represents monthly returns. I don't see any weird discontinuities and the variability looks like what I would generally expect for stock returns.

One thing about this plot is it's difficult to determine which line belonds to which stock. `ggplot2` makes it easy to plot groups of data, either by assinging different colors to values within a group, or by separating series into their own panels (or facets).

```
# This version of the plot will group and color each line according to ticker
ggplot(all_monthly_stock_returns_stacked, aes(date, return, color = ticker)) +
  geom_line()
```




```
# This version of the plot will split the lines into their own facets by ticker
ggplot(all_monthly_stock_returns_stacked, aes(date, return)) +
  facet_wrap(~ticker) +
  geom_line()
```



This data makes sense to me - larger, more established companies have much less variation in monthly returns than TSLA and FSLR. Data is available at regular intervals.

The next step is to get data for the S&P 500 and the risk free interest rate, which we will call the 13 week treasury bill rate. The same `get_monthly_returns` function can work for the S&P 500, and I'll use getSymbols() directly for the treasury rate with the St. Louis Federal Reserve (FRED) as a source. To download FRED data, I went to FRED and typed in "treasury bill rate" in the seach bar and was presented with a set of options for time series that match that description. The first series is what I want: "3-Month Treasury Bill: Secondary Market Rate." Clicking the "monthly" option leads me to the following view:

Note the url: https://fred.stlouisfed.org/series/TB3MS. The end of the URL (TB3MS) represents the identifier for the time series, which is what we will pass to quantmod to get data from FRED. As an aside, this is essentially how I got data from Yahoo finance for the S&P 500 and the 3 month treasury bill rate as well. I searched for these series and found the ticker representation for those values.

```
# download monthly returns for the S&P 500
sp500 = get_monthly_returns('^GSPC')

# get the monthly risk free rate from FRED
rf_rate = getSymbols('TB3MS', src = 'FRED', auto.assign = F)
rf_rate_adj = data.frame(
  return = rf_rate/100, # the risk free rate is presented as a percent, needs to be in same terms as th
  date = index(rf_rate)
  ) %>%
  rename(
    return = TB3MS
  )
tail(rf_rate_adj)
```

```
##            return       date
## 2018-08-01 0.0203 2018-08-01
## 2018-09-01 0.0213 2018-09-01
## 2018-10-01 0.0225 2018-10-01
## 2018-11-01 0.0233 2018-11-01
## 2018-12-01 0.0237 2018-12-01
## 2019-01-01 0.0237 2019-01-01
```

Now that I have all the data I need, I want to combine them for analysis. Since I want to run regressions of the form `(stock return - risk free rate) = alpha + beta(market_return - risk free rate) + error`, I need to have each stock return associated with the relevant risk free rate and market return for that

month. We need to keep in mind that the stock returns are currently stacked on top of each other, so we can't simply run a regression with all that data – we need to split the data by ticker and run each regression. There are efficient ways to do that in R, so it makes sense to keep those as a "long" `data.frame`.

First, let's combine the S&P 500 data and the risk free rate. In this case, I want these values side by side (joined) by date. Right now, the data isn't quite set up to cleanly join these two `data.frame` objects - I have a date that can be used for a join (see the joins tutorial for an in-depth presentation) but I have columns that have the same name. If I were to simply join these, I'd have funky column names, e.g. `return` and `return.1` or something like that to identify which table the column came from. Further, `dplyr` join functions automatically will join values based on column names, so this would attempt to join by return and ticker, which would result in zero matches.

```
# I can do everything in one step:
# get rid of columns I don't want
# rename columns to something different
# use a left join to pull all the data together
joined_comparators = rf_rate_adj %>%
  rename(
    rf_rate_return = return # rename the return column to rf_rate_return
  ) %>%
  left_join(
    # Pay attention to what is happening here
    # I am doing the column dropping and renaming steps WITHIN the left_join call
    # This means the join will happen on the result of these operations
    select(sp500, -ticker) %>%
      rename(
        sp500_return = return
      )
  )
```

```
## Joining, by = "date"
```

```
# Lets take a look at the result:
head(joined_comparators)
```

```
##   rf_rate_return       date sp500_return
## 1         0.0072 1934-01-01           NA
## 2         0.0062 1934-02-01           NA
## 3         0.0024 1934-03-01           NA
## 4         0.0015 1934-04-01           NA
## 5         0.0016 1934-05-01           NA
## 6         0.0015 1934-06-01           NA
```

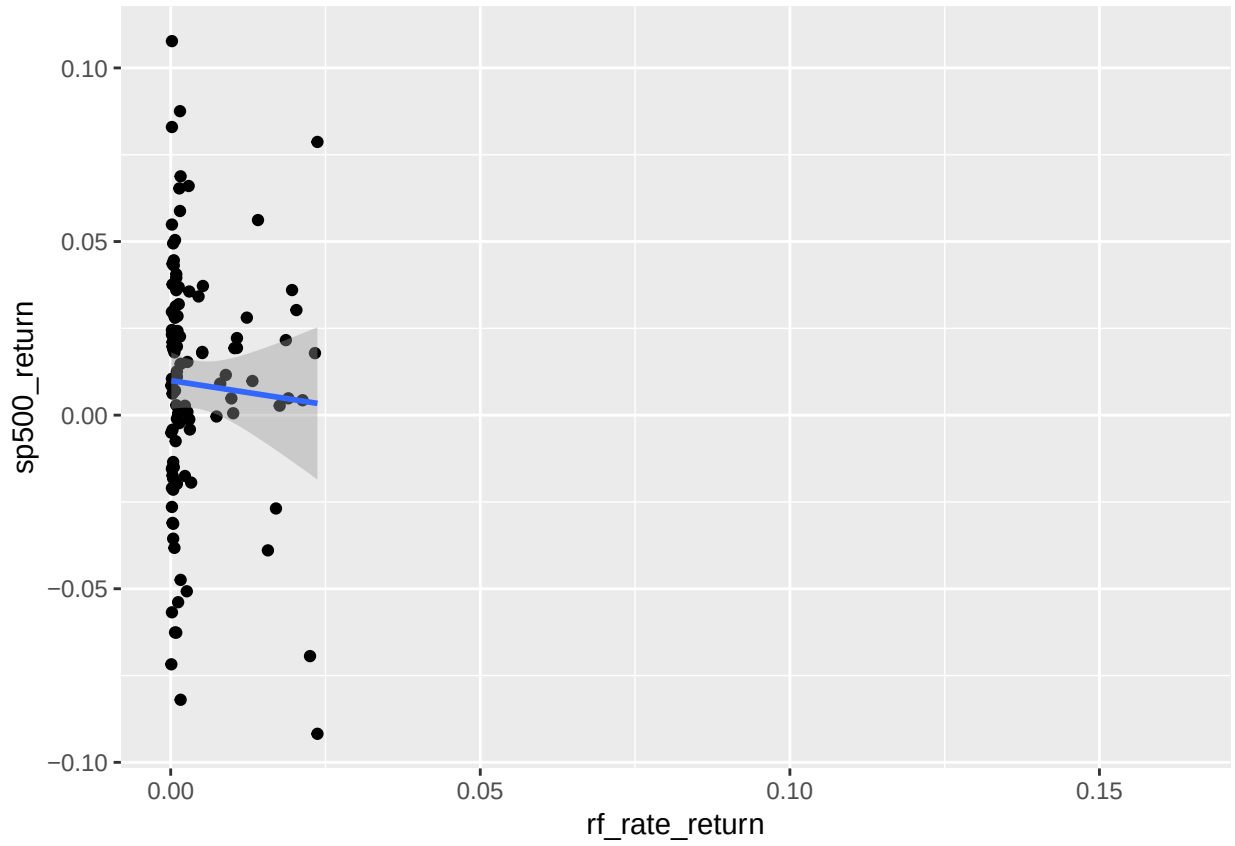
```
tail(joined_comparators)
```

```
##      rf_rate_return       date sp500_return
## 1016         0.0203 2018-08-01  0.030263211
## 1017         0.0213 2018-09-01  0.004294287
## 1018         0.0225 2018-10-01 -0.069403356
## 1019         0.0233 2018-11-01  0.017859357
## 1020         0.0237 2018-12-01 -0.091776895
## 1021         0.0237 2019-01-01  0.078684402
```

```
# How do the series compare?
ggplot(joined_comparators, aes(rf_rate_return, sp500_return)) +
  geom_point() +
  stat_smooth(method = 'lm') # add a linear model to summarize the relationship
```

```
## Warning: Removed 912 rows containing non-finite values (stat_smooth).
```

```
## Warning: Removed 912 rows containing missing values (geom_point).
```



Now we have our comparators ready, we can join this data onto the stock tickers using the date column. We don't have to worry about renaming any columns now – the only common column is `date`. I use a left join again because that is the safest join – it keeps all of the rows in the left hand side and only adds rows that match from the right. This allows one to see what data is potentially missing.

Additional steps should be taken to ensure that your dates look right – it's easy for missing values to go unnoticed in important places. Are there any dates in the comparators that aren't found in the stock returns? We already know that the series go out further for the FRED data (notice the `head` of the comparators) because `getSymbols` for FRED didn't respect the `from` argument.

```
# setdiff will subtract two sets and find the values in the left hand side that
# aren't in the right side
# letters is a default R object that contains the letters of the alphabet.
# letters[1:5] has the first 5 letters, letters[2:6] has 2-6. The difference is the first letter, 'a'
# because a is in the left side and not the right
setdiff(letters[1:5], letters[2:6])
```

```
## [1] "a"
```

```
# lets check our dates:
unique_stock_months = unique(all_monthly_stock_returns_stacked$date)
unique_comparator_months = unique(joined_comparators$date)

# all of the months in the stocks are found in the comparators! It's still possible for sp500 returns t
setdiff(unique_stock_months, unique_comparator_months)
```

```
## [1] 17928
```

```
# lets join all of our data together and compute values net of the risk free rate!
all_joined_analysis_data = left_join(all_monthly_stock_returns_stacked, joined_comparators) %>%
  mutate(
    return_net_rf = return - rf_rate_return,
    sp500_net_rf = sp500_return - rf_rate_return
  )
```

```
## Joining, by = "date"
```

```
head(all_joined_analysis_data)
```

```
##        return       date ticker rf_rate_return sp500_return return_net_rf
## 1 -0.07774929 2010-01-01    CVX         0.0006  -0.03823356   -0.07834929
## 2  0.00249584 2010-02-01    CVX         0.0011   0.02851369    0.00139584
## 3  0.04882433 2010-03-01    CVX         0.0015   0.05879643    0.04732433
## 4  0.07398127 2010-04-01    CVX         0.0016   0.01475923    0.07238127
## 5 -0.09295185 2010-05-01    CVX         0.0016  -0.08197584   -0.09455185
## 6 -0.08135917 2010-06-01    CVX         0.0012  -0.05388244   -0.08255917
##   sp500_net_rf
## 1  -0.03883356
## 2   0.02741369
## 3   0.05729643
## 4   0.01315923
## 5  -0.08357584
## 6  -0.05508244
```

```
tail(all_joined_analysis_data)
```

```
##          return       date ticker rf_rate_return sp500_return
## 540 -0.12229004 2018-09-01   TSLA         0.0213  0.004294287
## 541  0.27401149 2018-10-01   TSLA         0.0225 -0.069403356
## 542  0.03901341 2018-11-01   TSLA         0.0233  0.017859357
## 543 -0.05044517 2018-12-01   TSLA         0.0237 -0.091776895
## 544 -0.07746394 2019-01-01   TSLA         0.0237  0.078684402
## 545  0.02514495 2019-02-01   TSLA             NA           NA
##     return_net_rf sp500_net_rf
## 540   -0.14359004 -0.017005713
## 541    0.25151149 -0.091903356
## 542    0.01571341 -0.005440643
## 543   -0.07414517 -0.115476895
## 544   -0.10116394  0.054984402
## 545            NA           NA
```

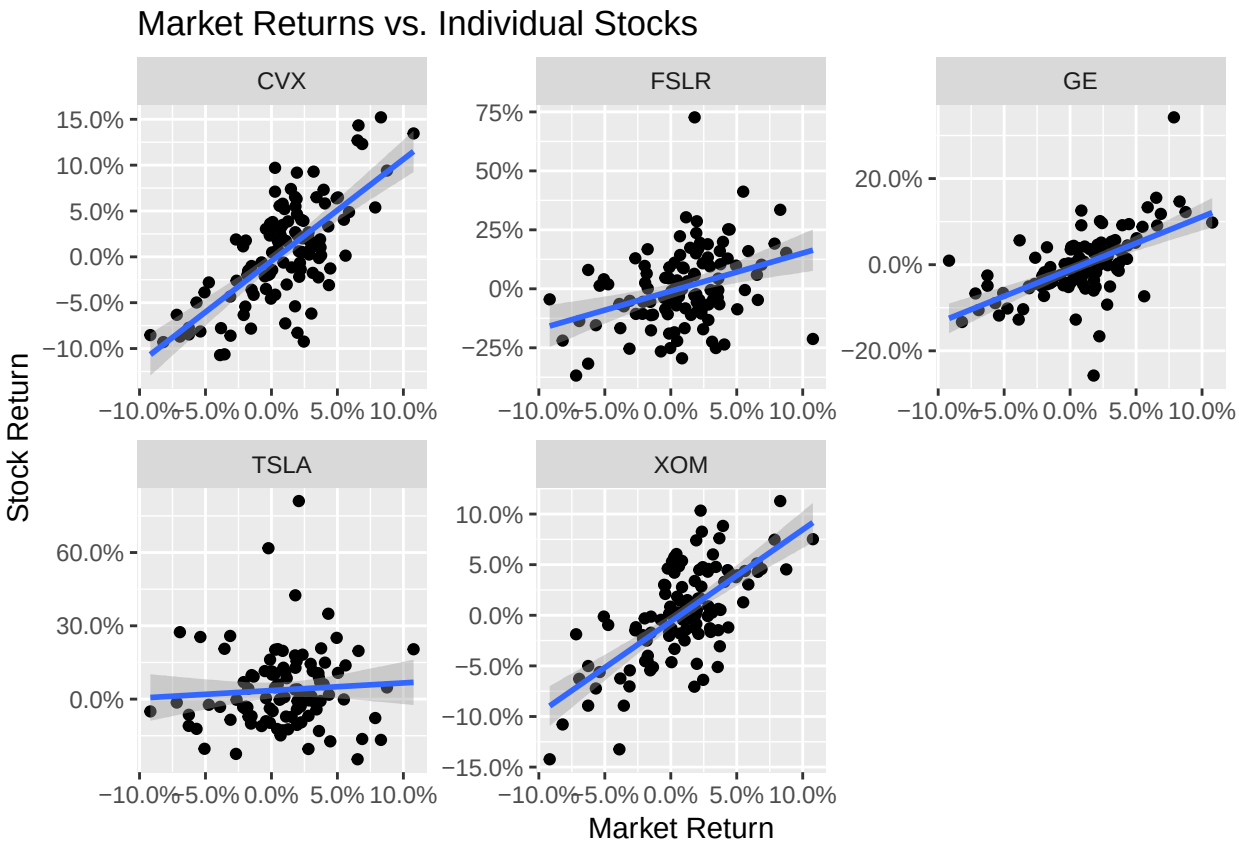
Now we have all of our data joined together for analysis. You may have noticed that a fair amount of work was required simply to get to this point. The typical refrain from professional analysts is that around 90% of your time is spent gathering and cleaning data, with the remainder spent on analyzing/modeling/interpretation. That's what makes R so important - you can automate data preparation steps and you can do complex manipulations super efficiently (once you are over the learning curve) compared to tools like Excel.

Before we run our models, let's take one last look at our data for a sanity check. How do the series relate?

```
# Market versus individual stocks
ggplot(all_joined_analysis_data, aes(sp500_return, return)) +
  facet_wrap(~ticker, scales = 'free') + # allow each panel to have its own scale
  geom_point() +
  stat_smooth(method = 'lm') +
  labs(
    x = 'Market Return', y = 'Stock Return', title = 'Market Returns vs. Individual Stocks'
  ) +
  scale_y_continuous(labels = percent) +
  scale_x_continuous(labels = percent)
```

```
## Warning: Removed 5 rows containing non-finite values (stat_smooth).
```

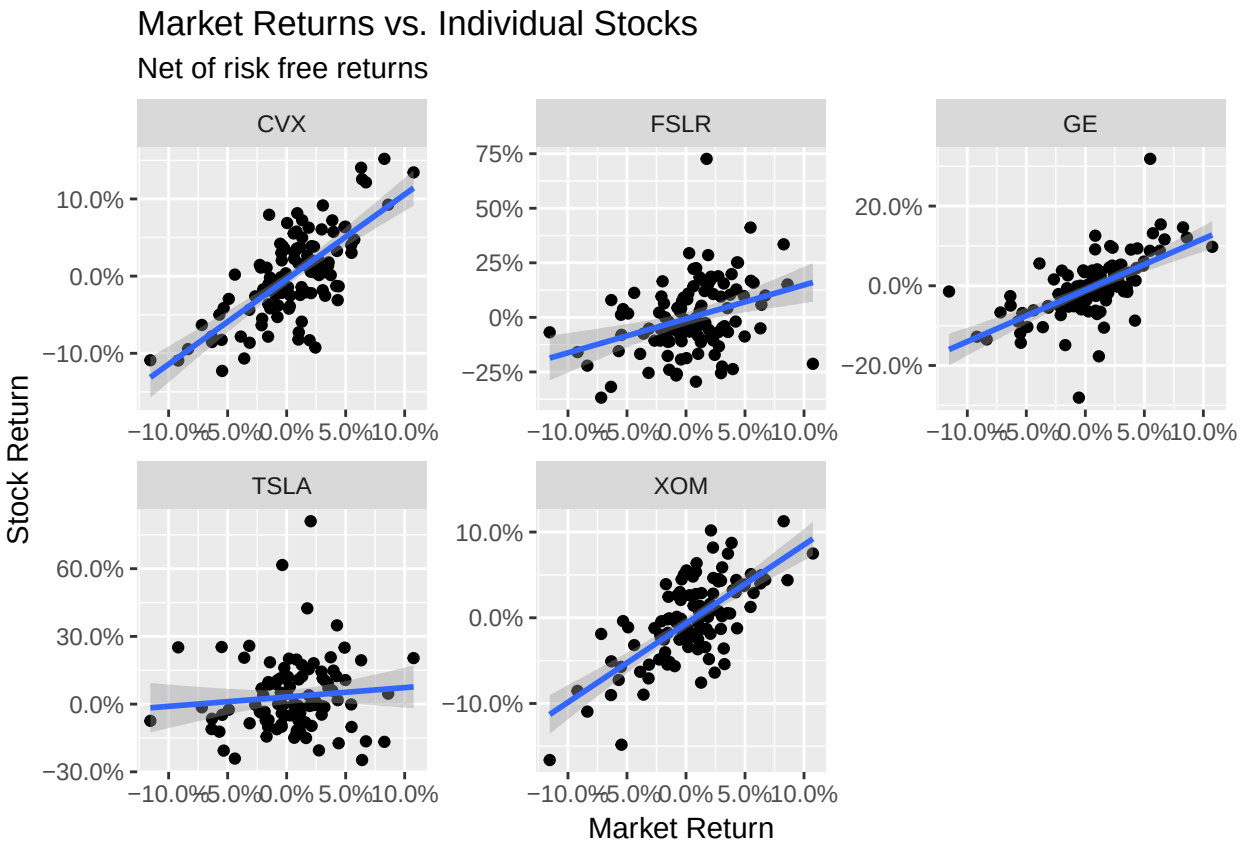
```
## Warning: Removed 5 rows containing missing values (geom_point).
```



```
# net of risk free rate
ggplot(all_joined_analysis_data, aes(sp500_net_rf, return_net_rf)) +
  facet_wrap(~ticker, scales = 'free') + # allow each panel to have its own scale
  geom_point() +
  stat_smooth(method = 'lm') +
  labs(
    x = 'Market Return', y = 'Stock Return', title = 'Market Returns vs. Individual Stocks',
    subtitle = 'Net of risk free returns'
  ) +
  scale_y_continuous(labels = percent) +
  scale_x_continuous(labels = percent)
```

```
## Warning: Removed 5 rows containing non-finite values (stat_smooth).
```

```
## Warning: Removed 5 rows containing missing values (geom_point).
```



As theory predicts, there is a strong relationship between the market returns and individual stock returns net of the risk free rate. Let's estimate those betas. We want to run a model for each subset of data and extract model coefficients.

As with before, I'll run some code for the single ticker case and then demonstrate how to do it for all tickers.

```
# subset to tesla
tesla_subset = filter(all_joined_analysis_data, ticker == 'TSLA')
head(tesla_subset)
```

```
##        return       date ticker rf_rate_return sp500_return return_net_rf
## 1  0.25421053 2010-06-01   TSLA         0.0012  -0.05388244    0.25301053
## 2 -0.16323957 2010-07-01   TSLA         0.0016   0.06877785   -0.16483957
## 3 -0.02306926 2010-08-01   TSLA         0.0016  -0.04744918   -0.02466926
## 4  0.04774127 2010-09-01   TSLA         0.0015   0.08755110    0.04624127
## 5  0.07006369 2010-10-01   TSLA         0.0013   0.03685599    0.06876369
## 6  0.61767408 2010-11-01   TSLA         0.0014  -0.00229025    0.61627408
##   sp500_net_rf
## 1  -0.05508244
## 2   0.06717785
## 3  -0.04904918
## 4   0.08605110
## 5   0.03555599
## 6  -0.00369025
```

```
tail(tesla_subset)
```

```
##          return       date ticker rf_rate_return sp500_return
## 100 -0.12229004 2018-09-01   TSLA         0.0213  0.004294287
## 101  0.27401149 2018-10-01   TSLA         0.0225 -0.069403356
## 102  0.03901341 2018-11-01   TSLA         0.0233  0.017859357
## 103 -0.05044517 2018-12-01   TSLA         0.0237 -0.091776895
## 104 -0.07746394 2019-01-01   TSLA         0.0237  0.078684402
## 105  0.02514495 2019-02-01   TSLA             NA           NA
##     return_net_rf sp500_net_rf
## 100   -0.14359004 -0.017005713
## 101    0.25151149 -0.091903356
## 102    0.01571341 -0.005440643
## 103   -0.07414517 -0.115476895
## 104   -0.10116394  0.054984402
## 105            NA           NA
```

```
# fit linear model via OLS. To the left of ~ is the Y of the analysis, to the right are the Xs or predi
# data specifies the data to be used
tesla_model = lm(return_net_rf ~ sp500_net_rf, data = tesla_subset)

# this will provide a summary of the model
summary(tesla_model)
```

```
##
## Call:
## lm(formula = return_net_rf ~ sp500_net_rf, data = tesla_subset)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.30582 -0.09561 -0.03279  0.07543  0.77020
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   0.03166    0.01593   1.987   0.0496 *
## sp500_net_rf  0.41487    0.43707   0.949   0.3448
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 0.1609 on 102 degrees of freedom
##   (1 observation deleted due to missingness)
## Multiple R-squared:  0.008756,   Adjusted R-squared:  -0.000962
## F-statistic: 0.901 on 1 and 102 DF,  p-value: 0.3448
```

The Estimate column shows the estimated coefficients for the intercept and the slope estimate (beta) for the market. We want to extract that value as our beta.

```
# this extracts just the coefficients
coef(tesla_model)
```

```
##  (Intercept) sp500_net_rf
##   0.03165677   0.41487429
```

```
# you can also extract components from the summary function:
names(summary(tesla_model)) # this is what you can get
```

```
##  [1] "call"          "terms"         "residuals"     "coefficients"
##  [5] "aliased"       "sigma"         "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"  "na.action"
```

```
# nice table of coefficient information with standard errors and t values
summary(tesla_model)$coefficients
```

```
##                Estimate Std. Error   t value   Pr(>|t|)
## (Intercept)  0.03165677 0.01593243 1.9869394 0.04961091
## sp500_net_rf 0.41487429 0.43707044 0.9492161 0.34475433
```

```
# we just want the estimate for the market beta though.

# this is a numeric vector, but as we saw above, it has names for each element.
# USE THE NAMES TO EXTRACT DATA, NOT THE ELEMENT NUMBER
class(coef(tesla_model))
```

```
## [1] "numeric"
```

```
# using the name will make your life way easier in the long run. You can actually see what it is that y
tesla_beta = coef(tesla_model)['sp500_net_rf']
print(tesla_beta)
```

```
## sp500_net_rf
##    0.4148743
```

Now we know how to get the beta for one stock, lets extract it for all of them in a loop!

```
# I'm down at the bottom of my script and I forgot the name of the vector
# where I stored all the stock tickers
# this function tells me what is in memory:
ls()
```

```
##  [1] "all_joined_analysis_data"
##  [2] "all_monthly_stock_returns"
##  [3] "all_monthly_stock_returns_stacked"
##  [4] "downloaded_data"
##  [5] "get_monthly_returns"
##  [6] "joined_comparators"
##  [7] "monthly_returns"
##  [8] "returns_df"
##  [9] "rf_rate"
## [10] "rf_rate_adj"
## [11] "sp500"
## [12] "stock_tickers_vec"
## [13] "tesla_beta"
## [14] "tesla_model"
## [15] "tesla_returns"
## [16] "tesla_subset"
## [17] "unique_comparator_months"
## [18] "unique_stock_months"
```

```r
# loop over the stock tickers, subset the data, run the model, extract the beta
all_betas = lapply(stock_tickers_vec, function(the_ticker){
  ticker_sub = filter(all_joined_analysis_data, ticker == the_ticker)
  the_model = lm(return_net_rf ~ sp500_net_rf, data = ticker_sub)
  beta = coef(the_model)['sp500_net_rf']
  names(beta) = NULL # get rid of the sp500_net_rf column name
  return(beta)
})

# all betas is a list, let's give each element the name of the ticker it represents
names(all_betas) = stock_tickers_vec

print(all_betas)
```

```
## $CVX
## [1] 1.10013
##
## $XOM
## [1] 0.9192335
##
## $FSLR
## [1] 1.553911
##
## $GE
## [1] 1.289475
##
## $TSLA
## [1] 0.4148743
```

Now you have your betas saved in a list and you can do what you want with them! Do these values make sense given what was seen in the scatterplot before? Now you can use these betas to build a portfolio.