We are ready to demo our new **new experimental package for Algorithmic Trading**, flyingfox, which uses reticulate to to bring Quantopian’s open source algorithmic trading Python library, Zipline, to R. The flyingfox library is part of our *NEW Business Science Labs* innovation lab, which is dedicated to bringing experimental packages to our followers *early on* so they can test them out and let us know what they think before they make their way to CRAN. This article includes a long-form code tutorial on how to **perform backtest optimizations of trading algorithms via grid search and parallel processing**. In this article, we’ll show you how to use the combination of tibbletime (time-based extension of tibble) + furrr (a parallel-processing compliment to purrr) + flyingfox (Zipline in R) to **develop a backtested trading algorithm that can be optimized via grid search and parallel processing**.

**New: Business Science Labs**

I (Davis) am excited to introduce a new open source initiative called **Business Science Labs**. A lot of the experimental work we do is done behind the scenes, and much of it you don’t see early on. What you do see is a “refined” version of what we *think* you need based on our perception, which is not always reality. We aim to change this. Starting today, we have created *Business Science Labs*, which is aimed at bringing our experimental software to you earlier so you can test it out and let us know your thoughts!

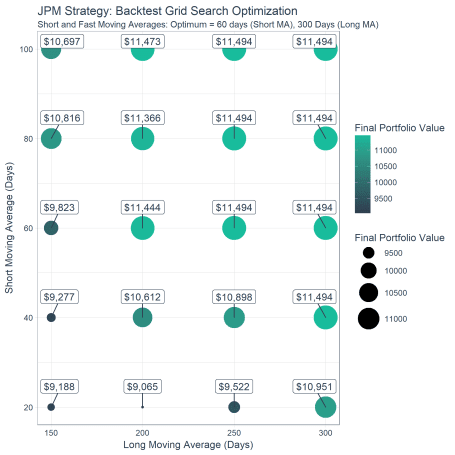
Our first initiative is to bring Quantopian’s Open Source algorithmic trading Python library, Zipline, to R via an experimental package called flyingfox (built using the awesome reticulate package).

**What We’re Going To Learn**

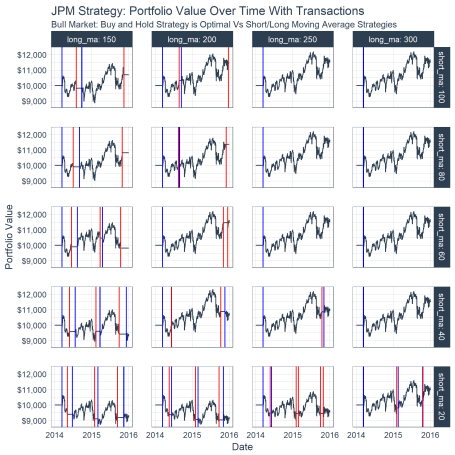
Introducing *Business Science Labs* is exciting, but we really want to educate you on some new packages! In this tutorial, we are going to go over how to **backtest algorithmic trading strategies using parallel processing and Quantopian’s Zipline infrastructure in R**. You’ll gain exposure to a tibbletime, furrr, and our experimental flyingfox package. The general progression is:

* tibbletime: What it is and why it’s essential to performing scalable time-based calculations in the tidyverse
* furrr: Why you need to know this package for speeding up code by processing purrr in parallel
* flyingfox: The story behind the package, and how you can use it to test algorithmic trading strategies
* tibbletime + furrr + flyingfox: Putting it all together to perform parallelized algorithmic trading strategies and analyze time-based performance

Here’s an example of the grid search we perform to determine which are the best combinations of short and long moving averages for the stock symbol JPM (JP Morgan).



Here’s an example of the time series showing the order (buy/sell) points determined by the moving average crossovers, and the effect on the portfolio value.

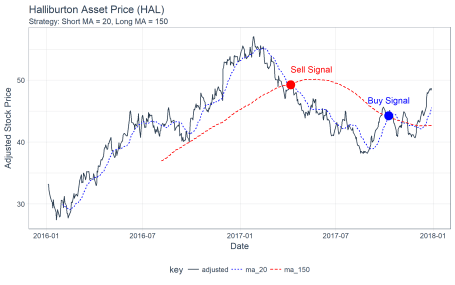


**Algorithmic Trading Strategies And Backtesting**

**Algorithmic trading** is nothing new. Financial companies have been performing algorithmic trading for years as a way of attempting to “beat” the market. It can be very difficult to do, but some traders have successfully applied advanced algorithms to yield significant profits.

Using an algorithm to trade boils down to **buying and selling**. In the simplest case, when an algorithm detects an asset (a stock) is going to go higher, a buy order is placed. Conversely, when the algorithm detects that an asset is going to go lower, a sell order is placed. Positions are managed by buying and selling all or part of the portfolio of assets. To keep things simple, we’ll focus on just the full buy/sell orders.

One very basic method of algorithmic trading is using **short and long moving averages** to detect shifts in trend. The crossover is the point where a buy/sell order would take place. The figure below shows the price of Halliburton (symbol “HAL”), which a trader would have an initial position in of say 10,000 shares. In a hypothetical case, the trader could use a combination of a 20 day short moving average and a 150 day long moving average and look for buy/sell points at the crossovers. If the trader hypothetically sold his/her position in full on the sell and bought the position back in full, then the trader would stand to avoid a delta loss of approximately $5/share during the downswing, or $50,000.



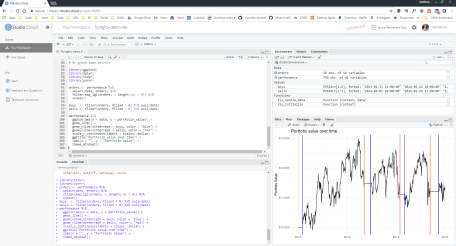
**Backtesting** is a strategy that is used to detect how a trading strategy would have performed in the past. It’s impossible to know what the future will bring, but using trading strategies that work in the past helps to instill confidence in an algorithm.

Quantopian is a platform designed to enable anyone to develop algorithmic trading strategies. To help its community, Quantopian provides several open source tools. The one we’ll focus on is Zipline for backtesting. There’s one downside: it’s only available in Python.

With the advent of the reticulate package, which enables porting any Python library to R, we took it upon ourselves to test out the viability of porting Zipline to R. Our experiment is called flyingfox.

**RStudio Cloud Experiment Sandbox**

In this code-based tutorial, we’ll use an experimental package called flyingfox. It has several dependencies including Python that require setup time and effort. For those that want to test out flyingfox quickly, we’ve created a **FREE RStudio Cloud Sandbox** for running experiments. You can access the Cloud Sandbox here for FREE: https://rstudio.cloud/project/38291



**Packages Needed For Backtest Optimization**

The meat of this code-tutorial is Backtest Optimization Using tibbletime + furrr + flyingfox . However, before we get to it, we’ll go over the *three main packages* used to do high-performance backtesting optimizations:

1. tibbletime: What it is, and why it’s essential to performing scalable time-based calculations in the tidyverse
2. furrr: Why you need to know this package for speeding up code by processing purrr in parallel
3. flyingfox: How to use it to test algorithmic trading strategies

Putting It All Together: tibbletime + furrr + flyingfox for **backtesting optimizations performed using parallel processing and grid search!**

**Install & Load Libraries**

**Install Packages**

For this post, you’ll need to install development version of flyingfox.

Library("flyingfox")

If you are on windows, you should also install the development version of furrr.

Library("furrr")

Other packages you’ll need include tibbletime, furrr, and tidyverse. We’ll also load tidyquant mainly for the ggplot2 themes. We’ll install ggrepel to repel overlapping plot labels. You can install these from CRAN using install.packages().

**Load Packages**

library(tidyverse)

library(tibbletime)

library(furrr)

library(flyingfox)

library(tidyquant)

library(ggrepel)

We’ll cover how a few packages work before jumping into backtesting and optimizations.

**1. tibbletime**

The tibbletime package is a cornerstone for future time series software development efforts at *Business Science*. We have major plans for this package. Here are some key benefits:

* Time periods down to milliseconds are supported
* Because this is a tibble under the hood, we are able to leverage existing packages for analysis without reinventing the wheel
* Scalable grouped analysis is at your fingertips because of collapse\_by() and integration with group\_by()

It’s best to learn now, and we’ll go over the basics along with a few commonly used functions: collapse\_by(), rollify(), filter\_time(), and as\_period().

First, let’s get some data. We’ll use the FANG data set that comes with tibbletime, which includes stock prices for FB, AMZN, NFLX, and GOOG. We recommend using the tidyquant package to get this or other stock data.

data("FANG")

FANG

## # A tibble: 4,032 x 8

## symbol date open high low close volume adjusted

##

## 1 FB 2013-01-02 27.4 28.2 27.4 28 69846400 28

## 2 FB 2013-01-03 27.9 28.5 27.6 27.8 63140600 27.8

## 3 FB 2013-01-04 28.0 28.9 27.8 28.8 72715400 28.8

## 4 FB 2013-01-07 28.7 29.8 28.6 29.4 83781800 29.4

## 5 FB 2013-01-08 29.5 29.6 28.9 29.1 45871300 29.1

## 6 FB 2013-01-09 29.7 30.6 29.5 30.6 104787700 30.6

## 7 FB 2013-01-10 30.6 31.5 30.3 31.3 95316400 31.3

## 8 FB 2013-01-11 31.3 32.0 31.1 31.7 89598000 31.7

## 9 FB 2013-01-14 32.1 32.2 30.6 31.0 98892800 31.0

## 10 FB 2013-01-15 30.6 31.7 29.9 30.1 173242600 30.1

## # ... with 4,022 more rows

Next, you’ll need to convert thistbl\_df object to a tbl\_time object using the tbl\_time() function.

FANG\_time <- FANG %>%

tbl\_time(date) %>%

group\_by(symbol)

FANG\_time

## # A time tibble: 4,032 x 8

## # Index: date

## # Groups: symbol [4]

## symbol date open high low close volume adjusted

##

## 1 FB 2013-01-02 27.4 28.2 27.4 28 69846400 28

## 2 FB 2013-01-03 27.9 28.5 27.6 27.8 63140600 27.8

## 3 FB 2013-01-04 28.0 28.9 27.8 28.8 72715400 28.8

## 4 FB 2013-01-07 28.7 29.8 28.6 29.4 83781800 29.4

## 5 FB 2013-01-08 29.5 29.6 28.9 29.1 45871300 29.1

## 6 FB 2013-01-09 29.7 30.6 29.5 30.6 104787700 30.6

## 7 FB 2013-01-10 30.6 31.5 30.3 31.3 95316400 31.3

## 8 FB 2013-01-11 31.3 32.0 31.1 31.7 89598000 31.7

## 9 FB 2013-01-14 32.1 32.2 30.6 31.0 98892800 31.0

## 10 FB 2013-01-15 30.6 31.7 29.9 30.1 173242600 30.1

## # ... with 4,022 more rows

**collapse\_by()**

Beautiful. Now we have a time-aware tibble. Let’s test out some functions. First, let’s take a look at collapse\_by(), which is used for grouped operations. We’ll collapse by “year”, and calculate the average price for each of the stocks.

FANG\_time %>%

collapse\_by(period = "year") %>%

group\_by(symbol, date) %>%

summarise(mean\_adj = mean(adjusted))

## # A time tibble: 16 x 3

## # Index: date

## # Groups: symbol [?]

## symbol date mean\_adj

##

## 1 AMZN 2013-12-31 298.

## 2 AMZN 2014-12-31 333.

## 3 AMZN 2015-12-31 478.

## 4 AMZN 2016-12-30 700.

## 5 FB 2013-12-31 35.5

## 6 FB 2014-12-31 68.8

## 7 FB 2015-12-31 88.8

## 8 FB 2016-12-30 117.

## 9 GOOG 2013-12-31 442.

## 10 GOOG 2014-12-31 561.

## 11 GOOG 2015-12-31 602.

## 12 GOOG 2016-12-30 743.

## 13 NFLX 2013-12-31 35.3

## 14 NFLX 2014-12-31 57.5

## 15 NFLX 2015-12-31 91.9

## 16 NFLX 2016-12-30 102.

**rollify()**

Next, let’s take a look at rollify(). Remember the chart of Halliburton prices at the beginning. It was created using rollify(), which turns any function into a rolling function. Here’s the code for the chart. Notice how we create two rolling functions using mean() and supplying the appropriate window argument.

hal\_tbl <- tq\_get("HAL", from = "2016-01-01", to = "2017-12-31")

roll\_20 <- rollify(mean, window = 20)

roll\_150 <- rollify(mean, window = 150)

hal\_ma\_tbl <- hal\_tbl %>%

select(date, adjusted) %>%

mutate(

ma\_20 = roll\_20(adjusted),

ma\_150 = roll\_150(adjusted)

) %>%

gather(key = key, value = value, -date, factor\_key = T)

hal\_ma\_tbl %>%

ggplot(aes(date, value, color = key, linetype = key)) +

geom\_line() +

theme\_tq() +

scale\_color\_manual(values = c(palette\_light()[[1]], "blue", "red")) +

labs(title = "Halliburton Asset Price (HAL)",

subtitle = "Strategy: Short MA = 20, Long MA = 150",

x = "Date", y = "Adjusted Stock Price") +

annotate("point", x = as\_date("2017-04-07"), y = 49.25, colour = "red", size = 5) +

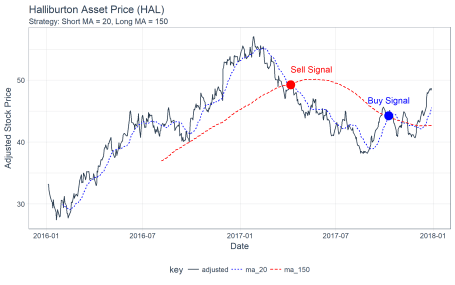
annotate("text", label = "Sell Signal",

x = as\_date("2017-04-07"), y = 49.25 + 2.5, color = "red", hjust = 0) +

annotate("point", x = as\_date("2017-10-09"), y = 44.25, colour = "blue", size = 5) +

annotate("text", label = "Buy Signal",

x = as\_date("2017-10-09"), y = 44.25 + 2.5, color = "blue")



**filter\_time()**

Let’s check out filter\_time(), which enables easier subsetting of time-based indexes. Let’s redo the chart above, instead focusing in on sell and buy signals, which occur after February 2017. We can convert the previously stored hal\_ma\_tbl to a tbl\_time object, group by the “key” column, and then filter using the time function format filter\_time("2017-03-01" ~ "end"). We then reuse the plotting code above.

hal\_ma\_time <- hal\_ma\_tbl %>%

as\_tbl\_time(date) %>%

group\_by(key)

hal\_ma\_time %>%

filter\_time("2017-03-01" ~ "end") %>%

ggplot(aes(date, value, color = key, linetype = key)) +

geom\_line() +

theme\_tq() +

scale\_color\_manual(values = c(palette\_light()[[1]], "blue", "red")) +

labs(title = "Halliburton Asset Price (HAL)", subtitle = "2016 through 2017",

x = "Date", y = "Adjusted Stock Price") +

annotate("point", x = as\_date("2017-04-07"), y = 49.25, colour = "red", size = 5) +

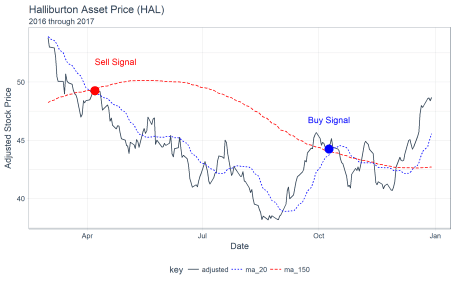
annotate("text", label = "Sell Signal",

x = as\_date("2017-04-07"), y = 49.25 + 2.5, color = "red", hjust = 0) +

annotate("point", x = as\_date("2017-10-09"), y = 44.25, colour = "blue", size = 5) +

annotate("text", label = "Buy Signal",

x = as\_date("2017-10-09"), y = 44.25 + 2.5, color = "blue")



**as\_period()**

We can use the as\_period() function to change the periodicity to a less granular level (e.g. going from daily to monthly). Here we convert the HAL share prices from daily periodicity to monthly periodicity.

hal\_ma\_time %>%

as\_period(period = "monthly", side = "end") %>%

ggplot(aes(date, value, color = key, linetype = key)) +

geom\_line() +

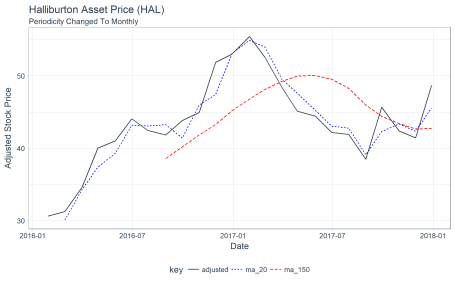
theme\_tq() +

scale\_color\_manual(values = c(palette\_light()[[1]], "blue", "red")) +

labs(title = "Halliburton Asset Price (HAL)",

subtitle = "Periodicity Changed To Monthly",

x = "Date", y = "Adjusted Stock Price")



Next, let’s check out a new package for parallel processing using purrr: furrr!

**2. furrr**

The furrr package combines the awesome powers of future for parallel processing with purrr for iteration. Let’s break these up into pieces by purrr, future, and furrr.

**purrr**

The purrr package is used for *iteration* over a number of different types of generic objects in R, including vectors, lists, and tibbles. The main function used is map(), which comes in several varieties (e.g. map(), map2(), pmap(), map\_chr(), map\_lgl(), map\_df(), etc). Here’s a basic example to get the class() of the columns of the FANG\_time variable. Using map() iterates over the columns of the data frame returning a list containing the contents of the function applied to each column.

FANG\_time %>%

map(class)

## $symbol

## [1] "character"

##

## $date

## [1] "Date"

##

## $open

## [1] "numeric"

##

## $high

## [1] "numeric"

##

## $low

## [1] "numeric"

##

## $close

## [1] "numeric"

##

## $volume

## [1] "numeric"

##

## $adjusted

## [1] "numeric"

**future**

The future package enables parallel processing. Here are a few important points:

* future is a unified interface for parallel programming in R.
* You set a “plan” for how code should be executed, call future() with an expression to evaluate, and call value() to retrieve the value. The first future() call sends off the code to another R process and is *non-blocking* so you can keep running R code in this session. It only blocks once you call value().
* Global variables and packages are automatically identified and  
  exported for you!

**Now, the major point**: If you’re familiar with purrr, you can take advantage of future parallel processing with furrr!

**furrr = future + purrr**

furrr takes the best parts of purrr and combines it with the parallel processing capabilities of future to create a powerful package with an interface instantly familiar to anyone who has used purrr before. All you need to do is two things:

1. Setup a plan() such as plan("multiprocess")
2. Change map() (or other purrr function) to future\_map() (or other compatible furrr function)

Every purrr function has a compatible furrr function. For example,  
map\_df() has future\_map\_df(). Set a plan, and run future\_map\_df() and that is all there  
is to it!

**furrr Example: Multiple Moving Averages**

Say you would like to process not a single moving average but multiple moving averages for a given data set. We can create a custom function, multi\_roller(), that uses map\_dfc() to iteratively map a rollify()-ed mean() based on a sequence of windows. Here’s the function and how it works when a user supplies a data frame, a column in the data frame to perform moving averages, and a sequence of moving averages.

multi\_roller <- function(data, col, ..f = mean,

window = c(10, 20, 30, 100, 200), sep = "V\_") {

col\_expr <- enquo(col)

ret\_tbl <- window %>%

set\_names(~ paste0(sep, window)) %>%

map\_dfc(~ rollify(..f, window = .x)(pull(data, !! col\_expr))) %>%

bind\_cols(data, .)

return(ret\_tbl)

}

We can test the function out with the FB stock prices from FANG. We’ll ungroup, filter by FB, and select the important columns, then pass the data frame to the multi\_roller() function with window = seq(10, 100, by = 10). Great, it’s working!

FANG\_time %>%

ungroup() %>%

filter(symbol == "FB") %>%

select(symbol, date, adjusted) %>%

multi\_roller(adjusted, mean, window = seq(10, 100, by = 10), sep = "ma\_")

## # A time tibble: 1,008 x 13

## # Index: date

## symbol date adjusted ma\_10 ma\_20 ma\_30 ma\_40 ma\_50 ma\_60

##

## 1 FB 2013-01-02 28 NA NA NA NA NA NA

## 2 FB 2013-01-03 27.8 NA NA NA NA NA NA

## 3 FB 2013-01-04 28.8 NA NA NA NA NA NA

## 4 FB 2013-01-07 29.4 NA NA NA NA NA NA

## 5 FB 2013-01-08 29.1 NA NA NA NA NA NA

## 6 FB 2013-01-09 30.6 NA NA NA NA NA NA

## 7 FB 2013-01-10 31.3 NA NA NA NA NA NA

## 8 FB 2013-01-11 31.7 NA NA NA NA NA NA

## 9 FB 2013-01-14 31.0 NA NA NA NA NA NA

## 10 FB 2013-01-15 30.1 29.8 NA NA NA NA NA

## # ... with 998 more rows, and 4 more variables: ma\_70 ,

## # ma\_80 , ma\_90 , ma\_100

We can apply this multi\_roller() function to a nested data frame. Let’s try it on our FANG\_time data set. We’ll select the columns of interest (symbol, date, and adjusted), group by symbol, and use the nest() function to get the data nested.

FANG\_time\_nested <- FANG\_time %>%

select(symbol, date, adjusted) %>%

group\_by(symbol) %>%

nest()

FANG\_time\_nested

## # A tibble: 4 x 2

## symbol data

##

## 1 FB

## 2 AMZN

## 3 NFLX

## 4 GOOG

Next, we can perform a rowwise map using the combination of mutate() and map(). We apply multi\_roller() as an argument to map() along with data (variable being mapped), and the additional static arguments, col and window.

FANG\_time\_nested %>%

mutate(ma = map(data, multi\_roller, col = adjusted, ..f = mean,

window = seq(10, 100, by = 10), sep = "ma\_")) %>%

select(-data) %>%

unnest()

## # A time tibble: 4,032 x 13

## # Index: date

## symbol date adjusted ma\_10 ma\_20 ma\_30 ma\_40 ma\_50 ma\_60

##

## 1 FB 2013-01-02 28 NA NA NA NA NA NA

## 2 FB 2013-01-03 27.8 NA NA NA NA NA NA

## 3 FB 2013-01-04 28.8 NA NA NA NA NA NA

## 4 FB 2013-01-07 29.4 NA NA NA NA NA NA

## 5 FB 2013-01-08 29.1 NA NA NA NA NA NA

## 6 FB 2013-01-09 30.6 NA NA NA NA NA NA

## 7 FB 2013-01-10 31.3 NA NA NA NA NA NA

## 8 FB 2013-01-11 31.7 NA NA NA NA NA NA

## 9 FB 2013-01-14 31.0 NA NA NA NA NA NA

## 10 FB 2013-01-15 30.1 29.8 NA NA NA NA NA

## # ... with 4,022 more rows, and 4 more variables: ma\_70 ,

## # ma\_80 , ma\_90 , ma\_100

Great, we have our moving averages. But…

**What if instead of 10 moving averages, we had 500? This would take a really long time to run on many stocks. Solution: Parallelize with furrr!**

There are two ways we could do this since there are two maps:

1. Parallelize the map() internal to the multi\_roller() function
2. Parallelize the rowwise map() applied to each symbol

We’ll choose the former (1) to show off furrr.

To make the multi\_roller\_parallel() function, copy the multi\_roller() function and do 2 things:

1. Add plan("multiprocess") at the beginning
2. Change map\_dfc() to future\_map\_dfc()

That’s it!

multi\_roller\_parallel <- function(data, col, ..f = mean,

window = c(10, 20, 30, 100, 200),

sep = "V\_") {

plan("multiprocess")

col\_expr <- enquo(col)

ret\_tbl <- window %>%

set\_names(~ paste0(sep, window)) %>%

future\_map\_dfc(~ rollify(..f, window = .x)(pull(data, !! col\_expr))) %>%

bind\_cols(data, .)

return(ret\_tbl)

}

In the previous rowwise map, switch out multi\_roller() for multi\_roller\_parallel() and change the window = 2:500. Sit back and let the function run in parallel using each of your computer cores.

FANG\_time\_nested %>%

mutate(ma = map(data, multi\_roller\_parallel, col = adjusted,

..f = mean, window = 2:500, sep = "ma\_")) %>%

select(-data) %>%

unnest()

## # A time tibble: 4,032 x 502

## # Index: date

## symbol date adjusted ma\_2 ma\_3 ma\_4 ma\_5 ma\_6 ma\_7

##

## 1 FB 2013-01-02 28 NA NA NA NA NA NA

## 2 FB 2013-01-03 27.8 27.9 NA NA NA NA NA

## 3 FB 2013-01-04 28.8 28.3 28.2 NA NA NA NA

## 4 FB 2013-01-07 29.4 29.1 28.6 28.5 NA NA NA

## 5 FB 2013-01-08 29.1 29.2 29.1 28.8 28.6 NA NA

## 6 FB 2013-01-09 30.6 29.8 29.7 29.5 29.1 28.9 NA

## 7 FB 2013-01-10 31.3 30.9 30.3 30.1 29.8 29.5 29.3

## 8 FB 2013-01-11 31.7 31.5 31.2 30.7 30.4 30.1 29.8

## 9 FB 2013-01-14 31.0 31.3 31.3 31.1 30.7 30.5 30.3

## 10 FB 2013-01-15 30.1 30.5 30.9 31.0 30.9 30.6 30.4

## # ... with 4,022 more rows, and 493 more variables: ma\_8 ,

## # ma\_9 , ma\_10 , ma\_11 , ma\_12 , ma\_13 ,

## # ma\_14 , ma\_15 , ma\_16 , ma\_17 , ma\_18 ,

## # ma\_19 , ma\_20 , ma\_21 , ma\_22 , ma\_23 ,

## # ma\_24 , ma\_25 , ma\_26 , ma\_27 , ma\_28 ,

## # ma\_29 , ma\_30 , ma\_31 , ma\_32 , ma\_33 ,

## # ma\_34 , ma\_35 , ma\_36 , ma\_37 , ma\_38 ,

## # ma\_39 , ma\_40 , ma\_41 , ma\_42 , ma\_43 ,

## # ma\_44 , ma\_45 , ma\_46 , ma\_47 , ma\_48 ,

## # ma\_49 , ma\_50 , ma\_51 , ma\_52 , ma\_53 ,

## # ma\_54 , ma\_55 , ma\_56 , ma\_57 , ma\_58 ,

## # ma\_59 , ma\_60 , ma\_61 , ma\_62 , ma\_63 ,

## # ma\_64 , ma\_65 , ma\_66 , ma\_67 , ma\_68 ,

## # ma\_69 , ma\_70 , ma\_71 , ma\_72 , ma\_73 ,

## # ma\_74 , ma\_75 , ma\_76 , ma\_77 , ma\_78 ,

## # ma\_79 , ma\_80 , ma\_81 , ma\_82 , ma\_83 ,

## # ma\_84 , ma\_85 , ma\_86 , ma\_87 , ma\_88 ,

## # ma\_89 , ma\_90 , ma\_91 , ma\_92 , ma\_93 ,

## # ma\_94 , ma\_95 , ma\_96 , ma\_97 , ma\_98 ,

## # ma\_99 , ma\_100 , ma\_101 , ma\_102 ,

## # ma\_103 , ma\_104 , ma\_105 , ma\_106 ,

## # ma\_107 , ...

*Bam! 500 moving averages run in parallel in fraction of the time it would take running in series.*

**3. flyingfox**

We have one final package we need to demo prior to jumping into our Algorithmic Trading Backtest Optimization: flyingfox.

**What is Quantopian?**

Quantopian is a company that has setup a community-driven platform for everyone (from traders to home-gamers) enabling development of algorithmic trading strategies. The one downside is they only use Python.

**What is Zipline?**

Zipline is a Python module open-sourced by Quantopian to help traders back-test their trading algorithms. Here are some quick facts about Quantopian’s Zipline Python module for backtesting algorithmic trading strategies:

* It is used to develop and backtest financial algorithms using Python.
* It includes an event-driven backtester (really good at preventing look-ahead bias)
* Algorithms consist of two main functions:
  1. initialize(): You write an initialize() function that is called *once* at the beginning of the backtest. This sets up variables for use in the backtest, schedules functions to be called daily, monthly, etc, and let’s you set slippage or commission for the backtest.
  2. handle\_data(): You then write a handle\_data() function that is called once per day (or minute) that implements the trading logic. You can place orders, retrieve historical pricing data, record metrics for performance evalutation and more.
* Extra facts: You can use any Python module inside the handle\_data() function, so you have a lot of flexibility here.

**What is reticulate?**

The reticulate package from RStudio is an interface with Python. It smartly takes care of (most) conversion between R and Python objects.

**Can you combine them?**

Yes, and that’s exactly what we did. We used reticulate to access the Zipline Python module from R!

**What is the benefit to R users?**

What if you could write your initialize() and handle\_data() functions in R *utilizing any financial or time series R package for your analysis* and then have them called from Python and Zipline?

**Introducing flyingfox: An R interface to Zipline**

flyingfox integrates the Zipline backtesting module in R! Further, it takes care of the overhead with creating the main infrastructure functions initialize() and handle\_data() by enabling the user to set up:

* fly\_initialize(): R version of Zipline’s initialize()
* fly\_handle\_data(): R version of Zipline’s handle\_data()

flyingfox takes care of passing these functions to Python and Zipline.

**Why “Flying Fox”?**

Zipline just doesn’t quite make for a good hex sticker. A flying fox is a synonym for zipliners, and it’s hard to argue that this majestic animal wouldn’t create a killer hex sticker.



**Getting Started With flyingfox: Moving Average Crossover**

Let’s do a Moving Average Crossover example using the following strategy:

* Using JP Morgan (JPM) stock prices
* If the 20 day short-term moving average crosses *above* the 150 day long-term moving average, buy 100% into JP Morgan
* If 20 day crosses below the 150 day, sell all of the current JPM position

**Setup**

Setup can take a while and take up some computer space due to ingesting data (which is where Zipline saves every major asset to your computer). We recommend one of two options:

1. **No weight option (for people that just want to try it out)**: Use our flyingfox sandbox on RStudio Cloud. You can connect here: https://rstudio.cloud/project/38291
2. **Heavy weight option (for people that want to expand and really test it)**: Follow the instructions on my GitHub page to fly\_ingest() data. The setup and data ingesting process are discussed here: https://github.com/DavisVaughan/flyingfox. Keep in mind that this is still a work in progress. We recommend doing the no weight option as a first start.

**Initialize**

First, write the R function for initialize(). It must take context as an argument. This is where you store variables used later, which are accessed via context$variable. We’ll store context$i = 0L to initialize the tracking of days, and context$asset = fly\_symbol("JPM") to trade the JPM symbol. You can select any symbol that you’d like (provided Quantopian pulls it from Quandl).

fly\_initialize <- function(context) {

# We want to track what day we are on.

context$i = 0L

# We want to trade JP Morgan stock

context$asset = fly\_symbol("JPM")

}

**Handle Data**

Next, write a handle\_data() function:

* This implements the crossover trading algorithm logic
* In this example we also use fly\_data\_history() to retrieve historical data each day for JP Morgan
* We use fly\_order\_target\_percent() to move to a new percentage amount invested in JP Morgan (if we order 1, we want to move to be 100% invested in JP Morgan, no matter where we were before)
* We use fly\_record() to store arbitrary metrics for review later

fly\_handle\_data <- function(context, data) {

# Increment day

context$i <- context$i + 1L

# While < 20 days of data, return

if(context$i < 20L) {

return()

}

# Calculate a short term (20 day) moving average

# by pulling history for the asset (JPM) and taking an average

short\_hist <- fly\_data\_history(data, context$asset, "price",

bar\_count = 20L, frequency = "1d")

short\_mavg <- mean(short\_hist)

# Calculate a long term (150 day) moving average

long\_hist <- fly\_data\_history(data, context$asset, "price",

bar\_count = 150L, frequency = "1d")

long\_mavg <- mean(long\_hist)

# If short > long, go 100% in JPM

if(short\_mavg > long\_mavg) {

fly\_order\_target\_percent(asset = context$asset, target = 1)

}

# Else if we hit the crossover, dump all of JPM

else if (short\_mavg < long\_mavg) {

fly\_order\_target\_percent(asset = context$asset, target = 0)

}

# Record today's data

# We record the current JPM price, along with the value of the short and long

# term moving average

fly\_record(

JPM = fly\_data\_current(data, context$asset, "price"),

short\_mavg = short\_mavg,

long\_mavg = long\_mavg

)

}

**Run The Algorithm**

Finally, run the algorithm from 2013-2016 using fly\_run\_algorithm().

performance\_time <- fly\_run\_algorithm(

initialize = fly\_initialize,

handle\_data = fly\_handle\_data,

start = as.Date("2013-01-01"),

end = as.Date("2016-01-01")

)

If you got to this point, you’ve just successfully run a single backtest. Let’s review the performance output.

**Reviewing The Performance**

Let’s glimpse performance\_time to see what the results show. It’s a tbl\_time time series data frame organized by the “date” column, and there is a ton of information. We’ll focus on:

* **date**: Time stamp for each point in the performance analysis
* **JPM**: This is the price of the asset
* **short\_mavg and long\_mavg**: These are our moving averages we are using for the buy/sell crossover signals
* **portfolio\_value**: The value of the portfolio at each time point
* **transactions**: Transactions stored as a list column. The tibble contains a bunch of information that is useful in determining what happened. More information below.

glimpse(performance\_time)

## Observations: 756

## Variables: 41

## $ date 2013-01-02 21:00:00, 2013-01-03 ...

## $ JPM NaN, NaN, NaN, NaN, NaN, NaN, NaN...

## $ algo\_volatility NaN, 0.00000000, 0.00000000, 0.00...

## $ algorithm\_period\_return 0.00000000, 0.00000000, 0.0000000...

## $ alpha NaN, NaN, NaN, NaN, NaN, NaN, NaN...

## $ benchmark\_period\_return 0.003691629, 0.007396885, 0.01111...

## $ benchmark\_volatility NaN, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0...

## $ beta NaN, NaN, NaN, NaN, NaN, NaN, NaN...

## $ capital\_used 0.000, 0.000, 0.000, 0.000, 0.000...

## $ ending\_cash 10000.0000, 10000.0000, 10000.000...

## $ ending\_exposure 0.00, 0.00, 0.00, 0.00, 0.00, 0.0...

## $ ending\_value 0.00, 0.00, 0.00, 0.00, 0.00, 0.0...

## $ excess\_return 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...

## $ gross\_leverage 0.0000000, 0.0000000, 0.0000000, ...

## $ long\_exposure 0.00, 0.00, 0.00, 0.00, 0.00, 0.0...

## $ long\_mavg NaN, NaN, NaN, NaN, NaN, NaN, NaN...

## $ long\_value 0.00, 0.00, 0.00, 0.00, 0.00, 0.0...

## $ longs\_count 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...

## $ max\_drawdown 0.000000000, 0.000000000, 0.00000...

## $ max\_leverage 0.0000000, 0.0000000, 0.0000000, ...

## $ net\_leverage 0.0000000, 0.0000000, 0.0000000, ...

## $ orders [[], [], [], [], [], [], [], [],...

## $ period\_close 2013-01-02 21:00:00, 2013-01-03 ...

## $ period\_label "2013-01", "2013-01", "2013-01", ...

## $ period\_open 2013-01-02 14:31:00, 2013-01-03 ...

## $ pnl 0.0000, 0.0000, 0.0000, 0.0000, 0...

## $ portfolio\_value 10000.000, 10000.000, 10000.000, ...

## $ positions [[], [], [], [], [], [], [], [],...

## $ returns 0.000000000, 0.000000000, 0.00000...

## $ sharpe NaN, NaN, NaN, NaN, NaN, NaN, NaN...

## $ short\_exposure 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...

## $ short\_mavg NaN, NaN, NaN, NaN, NaN, NaN, NaN...

## $ short\_value 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...

## $ shorts\_count 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...

## $ sortino NaN, NaN, NaN, NaN, NaN, NaN, NaN...

## $ starting\_cash 10000.0000, 10000.0000, 10000.000...

## $ starting\_exposure 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0...

## $ starting\_value 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0...

## $ trading\_days 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11...

## $ transactions [[], [], [], [], [], [], [], [],...

## $ treasury\_period\_return 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...

First, let’s plot the asset (JPM) along with the short and long moving averages. We can see there are a few crossovers.

performance\_time %>%

select(date, JPM, short\_mavg, long\_mavg) %>%

gather(key = "key", value = "value", -date, factor\_key = T) %>%

ggplot(aes(date, value, color = key, linetype = key)) +

geom\_line() +

theme\_tq() +

scale\_color\_manual(values = c(palette\_light()[[1]], "blue", "red")) +

scale\_y\_continuous(labels = scales::dollar) +

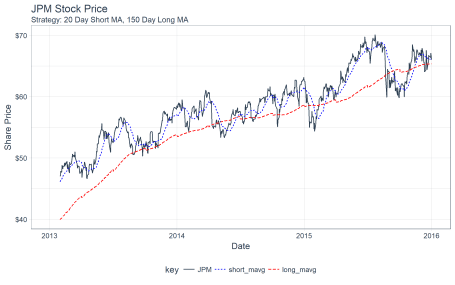
labs(

title = "JPM Stock Price",

subtitle = "Strategy: 20 Day Short MA, 150 Day Long MA",

x = "Date", y = "Share Price"

)



Next, we can investigate the transactions. Stored within the performance\_time output are *transaction information* as nested tibbles. We can get these values by flagging which time points contain tibbles and the filtering and unnesting. A transaction type can be determined if the “amount” of the transaction (number of shares bought or sold) is positive or negative.

transactions\_flagged\_time <- performance\_time %>%

select(date, portfolio\_value, transactions) %>%

mutate(flag\_transactions = map\_lgl(transactions, is.tibble)) %>%

filter(flag\_transactions == TRUE) %>%

unnest() %>%

mutate(transaction\_type = ifelse(amount >= 0, "buy", "sell"))

transactions\_flagged\_time %>%

glimpse()

## Observations: 9

## Variables: 10

## $ date 2013-01-31 21:00:00, 2014-05-07 20:00:...

## $ portfolio\_value 9994.801, 11472.859, 11467.007, 11488.6...

## $ flag\_transactions TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRU...

## $ commission NA, NA, NA, NA, NA, NA, NA, NA, NA

## $ amount 212, -212, 197, 2, -199, 179, -179, 166, 3

## $ sid [, "573c67da9fb6481c8a10cc55cddf57eb", "79...

## $ price 47.07353, 54.02298, 57.44871, 57.55877,...

## $ dt 2013-01-31 21:00:00, 2014-05-07 21:00:...

## $ transaction\_type "buy", "sell", "buy", "buy", "sell", "b...

Finally, we can visualize the results using ggplot2. We can see that the ending portfolio value is just under $11.5K.

performance\_time %>%

ggplot(aes(x = date, y = portfolio\_value)) +

geom\_line(color = palette\_light()[[1]]) +

geom\_point(aes(color = transaction\_type), size = 5,

data = transactions\_flagged\_time) +

geom\_label\_repel(

aes(label = amount), vjust = -1, color = palette\_light()[[1]],

data = transactions\_flagged\_time) +

theme\_tq() +

scale\_color\_tq() +

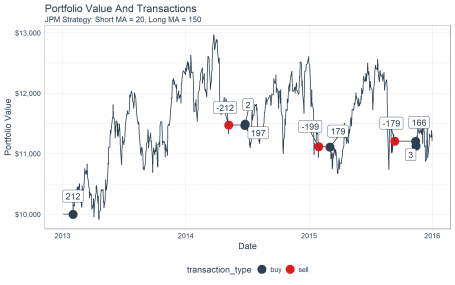
scale\_y\_continuous(labels = scales::dollar) +

labs(title = "Portfolio Value And Transactions",

subtitle = "JPM Strategy: Short MA = 20, Long MA = 150",

x = "Date", y = "Portfolio Value"

)



Last, let’s use tibbletime to see what happened to our portfolio towards the end. We’ll use the portfolio value as a proxy for the stock price, visualizing the crossover of the 20 and 150-day moving averages of the portfolio. Note that the actual algorithm is run with moving averages based on the adjusted stock price, not the portfolio value.

performance\_time %>%

# Data manipulation with tibbletime & multi\_roller() function

select(date, portfolio\_value) %>%

multi\_roller(portfolio\_value, mean, window = c(20, 150), sep = "ma\_") %>%

filter\_time("2015-05-01" ~ "end") %>%

gather(key = key, value = value, -date, factor\_key = T) %>%

# Visualization with ggplot2

ggplot(aes(date, value, color = key, linetype = key)) +

geom\_line() +

theme\_tq() +

scale\_color\_manual(values = c(palette\_light()[[1]], "blue", "red")) +

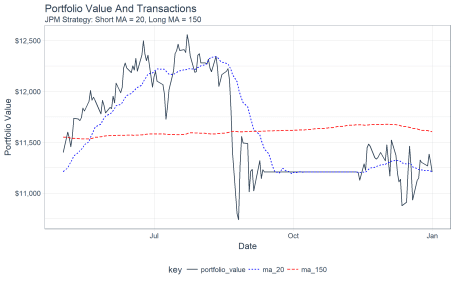
scale\_y\_continuous(labels = scales::dollar) +

labs(title = "Portfolio Value And Transactions",

subtitle = "JPM Strategy: Short MA = 20, Long MA = 150",

x = "Date", y = "Portfolio Value"

)



**Backtest Optimization Via Grid Search**

Now for the main course: **Optimizing our algorithm using the backtested performance**. To do so, we’ll combine what we learned from our three packages: tibbletime + furrr + flyingfox.

Let’s say we want to use backtesting to find the optimal combination or several best combinations of short and long term moving averages for our strategy. We can do this using [Cartesian Grid Search](https://en.wikipedia.org/wiki/Hyperparameter_optimization), which is simply creating a combination of all of the possible “hyperparameters” (parameters we wish to adjust). Recognizing that running multiple backtests can take some time, we’ll parallelize the operation too.

**Preparation**

Before we can do grid search, we need to adjust our fly\_handle\_data() function to enable our parameters to be adjusted. The two parameters we are concerned with are the short and long moving averages. We’ll add these as arguments of a new function fly\_handle\_data\_parameterized().

fly\_handle\_data\_parameterized <- function(context, data,

short\_ma = 20L, long\_ma = 150L) {

# Increment day

context$i <- context$i + 1L

# While < 300 days of data, return

if(context$i < 300) {

return()

}

# Calculate a short term (20 day) moving average

# by pulling history for the asset (JPM) and taking an average

short\_hist <- fly\_data\_history(data, context$asset, "price",

bar\_count = short\_ma, frequency = "1d")

short\_mavg <- mean(short\_hist)

# Calculate a long term (150 day) moving average

long\_hist <- fly\_data\_history(data, context$asset, "price",

bar\_count = long\_ma, frequency = "1d")

long\_mavg <- mean(long\_hist)

# If short > long, go 100% in JPM

if(short\_mavg > long\_mavg) {

fly\_order\_target\_percent(asset = context$asset, target = 1)

}

# Else if we hit the crossover, dump all of JPM

else if (short\_mavg < long\_mavg) {

fly\_order\_target\_percent(asset = context$asset, target = 0)

}

# Record today's data

# We record the current JPM price, along with the value of the short and long

# term moving average

fly\_record(

JPM = fly\_data\_current(data, context$asset, "price"),

short\_mavg = short\_mavg,

long\_mavg = long\_mavg

)

}

**Making The Grid**

Next, we can create a grid of values from a list() containing the hyperparameter values. We can turn this into a cross product as a tibble using the cross\_df() function.

ma\_grid\_tbl <- list(

short\_ma = seq(20, 100, by = 20),

long\_ma = seq(150, 300, by = 50)

) %>%

cross\_df()

ma\_grid\_tbl

## # A tibble: 20 x 2

## short\_ma long\_ma

##

## 1 20 150

## 2 40 150

## 3 60 150

## 4 80 150

## 5 100 150

## 6 20 200

## 7 40 200

## 8 60 200

## 9 80 200

## 10 100 200

## 11 20 250

## 12 40 250

## 13 60 250

## 14 80 250

## 15 100 250

## 16 20 300

## 17 40 300

## 18 60 300

## 19 80 300

## 20 100 300

Now that we have the hyperparameters, let’s create a new column with the function we wish to run. We’ll use the partial() function to partially fill the function with the hyper parameters.

ma\_grid\_tbl <- ma\_grid\_tbl %>%

mutate(f = map2(.x = short\_ma,

.y = long\_ma,

.f = ~ partial(fly\_handle\_data\_parameterized,

short\_ma = .x, long\_ma = .y)))

ma\_grid\_tbl

## # A tibble: 20 x 3

## short\_ma long\_ma f

##

## 1 20 150

## 2 40 150

## 3 60 150

## 4 80 150

## 5 100 150

## 6 20 200

## 7 40 200

## 8 60 200

## 9 80 200

## 10 100 200

## 11 20 250

## 12 40 250

## 13 60 250

## 14 80 250

## 15 100 250

## 16 20 300

## 17 40 300

## 18 60 300

## 19 80 300

## 20 100 300

**Running Grid Search In Parallel Using furrr**

Now for the Grid Search. We use the future\_map() function to process in parallel. Make sure to setup a plan() first. The following function runs the fly\_run\_algorithm() for each fly\_handle\_data() function stored in the “f” column.

plan("multiprocess")

results\_tbl <- ma\_grid\_tbl %>%

mutate(results = future\_map(

.x = f,

.f = ~ fly\_run\_algorithm(

initialize = fly\_initialize,

handle\_data = .x,

start = as.Date("2013-01-01"),

end = as.Date("2016-01-01")

)

))

results\_tbl

## # A tibble: 20 x 4

## short\_ma long\_ma f results

##

## 1 20 150

## 2 40 150

## 3 60 150

## 4 80 150

## 5 100 150

## 6 20 200

## 7 40 200

## 8 60 200

## 9 80 200

## 10 100 200

## 11 20 250

## 12 40 250

## 13 60 250

## 14 80 250

## 15 100 250

## 16 20 300

## 17 40 300

## 18 60 300

## 19 80 300

## 20 100 300

**Inspecting The Backtest Performance Results**

The performance results are stored in the “results” column as tbl\_time objects. We can examine the first result.

results\_tbl$results[[1]]

## # A time tibble: 756 x 41

## # Index: date

## date JPM algo\_volatility algorithm\_period\_~ alpha

##

## 1 2013-01-02 21:00:00 NaN NaN 0 NaN

## 2 2013-01-03 21:00:00 NaN 0 0 NaN

## 3 2013-01-04 21:00:00 NaN 0 0 NaN

## 4 2013-01-07 21:00:00 NaN 0 0 NaN

## 5 2013-01-08 21:00:00 NaN 0 0 NaN

## 6 2013-01-09 21:00:00 NaN 0 0 NaN

## 7 2013-01-10 21:00:00 NaN 0 0 NaN

## 8 2013-01-11 21:00:00 NaN 0 0 NaN

## 9 2013-01-14 21:00:00 NaN 0 0 NaN

## 10 2013-01-15 21:00:00 NaN 0 0 NaN

## # ... with 746 more rows, and 36 more variables:

## # benchmark\_period\_return , benchmark\_volatility ,

## # beta , capital\_used , ending\_cash ,

## # ending\_exposure , ending\_value , excess\_return ,

## # gross\_leverage , long\_exposure , long\_mavg ,

## # long\_value , longs\_count , max\_drawdown ,

## # max\_leverage , net\_leverage , orders ,

## # period\_close , period\_label , period\_open ,

## # pnl , portfolio\_value , positions ,

## # returns , sharpe , short\_exposure ,

## # short\_mavg , short\_value , shorts\_count ,

## # sortino , starting\_cash , starting\_exposure ,

## # starting\_value , trading\_days , transactions ,

## # treasury\_period\_return

We can also get the final portfolio value using a combination of pull() and tail().

results\_tbl$results[[1]] %>%

pull(portfolio\_value) %>%

tail(1)

## [1] 9187.849

We can turn this into a function and map it to all of the columns to obtain the “final\_portfolio\_value” for each of the grid search combinations.

results\_tbl <- results\_tbl %>%

mutate(final\_portfolio\_value = map\_dbl(results,

~ pull(.x, portfolio\_value) %>%

tail(1)))

results\_tbl

## # A tibble: 20 x 5

## short\_ma long\_ma f results final\_portfolio\_value

##

## 1 20 150 9188.

## 2 40 150 9277.

## 3 60 150 9823.

## 4 80 150 10816.

## 5 100 150 10697.

## 6 20 200 9065.

## 7 40 200 10612.

## 8 60 200 11444.

## 9 80 200 11366.

## 10 100 200 11473.

## 11 20 250 9522.

## 12 40 250 10898.

## 13 60 250 11494.

## 14 80 250 11494.

## 15 100 250 11494.

## 16 20 300 10951.

## 17 40 300 11494.

## 18 60 300 11494.

## 19 80 300 11494.

## 20 100 300 11494.

**Visualizing The Backtest Performance Results**

Now let’s visualize the results to see which combinations of short and long moving averages maximize the portfolio value. It’s clear that short >= 60 days and long >= 200 days maximize the return. But, why?

results\_tbl %>%

ggplot(aes(long\_ma, short\_ma, color = final\_portfolio\_value)) +

geom\_point(aes(size = final\_portfolio\_value)) +

geom\_label\_repel(

aes(label = scales::dollar(round(final\_portfolio\_value, 0))),

vjust = -1, color = palette\_light()[[1]]) +

theme\_tq() +

scale\_color\_gradient(low = palette\_light()[[1]], high = palette\_light()[[3]]) +

scale\_size(range = c(1, 12)) +

labs(

title = "JPM Strategy: Backtest Grid Search Optimization",

subtitle = "Short and Fast Moving Averages: Optimum = 60 days (Short MA), 300 Days (Long MA)",

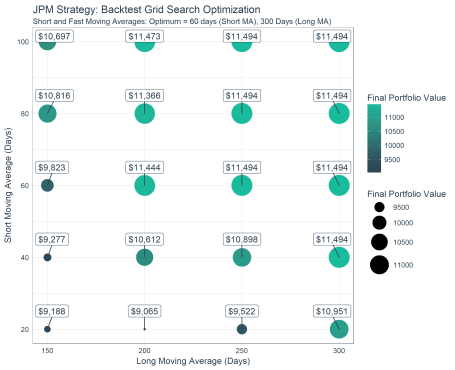
x = "Long Moving Average (Days)", y = "Short Moving Average (Days)",

color = "Final Portfolio Value", size = "Final Portfolio Value"

) +

theme(legend.direction = "vertical",

legend.position = "right")



Let’s get the transaction information (buy/sell) by unnesting the results and determining which transactions are buys and sells.

transactions\_tbl <- results\_tbl %>%

unnest(results) %>%

select(short\_ma:date, portfolio\_value, transactions) %>%

mutate(flag\_transactions = map\_lgl(transactions, is.tibble)) %>%

filter(flag\_transactions == TRUE) %>%

unnest() %>%

mutate(transaction\_type = ifelse(amount >= 0, "buy", "sell")) %>%

mutate(

short\_ma = as.factor(short\_ma) %>% fct\_rev(),

long\_ma = as.factor(long\_ma)

)

transactions\_tbl %>%

glimpse()

## Observations: 105

## Variables: 13

## $ short\_ma 20, 20, 20, 20, 20, 20, 20, 20, 20,...

## $ long\_ma 150, 150, 150, 150, 150, 150, 150, ...

## $ final\_portfolio\_value 9187.849, 9187.849, 9187.849, 9187....

## $ date 2014-03-13 20:00:00, 2014-03-14 20...

## $ portfolio\_value 9994.890, 9888.191, 9404.815, 9400....

## $ flag\_transactions TRUE, TRUE, TRUE, TRUE, TRUE, TRUE,...

## $ commission NA, NA, NA, NA, NA, NA, NA, NA, NA,...

## $ amount 172, 2, -174, 161, 2, -163, 147, -1...

## $ sid [...

## $ order\_id "d8a3551cb36c4ff985c42e4d78e00ef8",...

## $ price 57.44871, 56.82840, 54.02298, 57.44...

## $ dt 2014-03-13 21:00:00, 2014-03-14 21...

## $ transaction\_type "buy", "buy", "sell", "buy", "buy",...

Finally, we can visualize the portfolio value over time for each combination of short and long moving averages. By plotting the buy/sell transactions, we can see the effect on a stock with a bullish trend. The portfolios with the optimal performance are those that were bought and held rather than sold using the moving average crossover. For this particular stock, the benefit of downside protection via the moving average crossover costs the portfolio during the bullish uptrend.

results\_tbl %>%

unnest(results) %>%

mutate(

short\_ma = as.factor(short\_ma) %>% fct\_rev(),

long\_ma = as.factor(long\_ma)

) %>%

group\_by(short\_ma, long\_ma) %>%

filter\_time("2014" ~ "end") %>%

ggplot(aes(date, portfolio\_value)) +

geom\_line(color = palette\_light()[[1]]) +

geom\_vline(aes(xintercept = date), color = "blue",

data = filter(transactions\_tbl, transaction\_type == "buy")) +

geom\_vline(aes(xintercept = date), color = "red",

data = filter(transactions\_tbl, transaction\_type == "sell")) +

facet\_grid(short\_ma ~ long\_ma, labeller = label\_both) +

theme\_tq() +

scale\_y\_continuous(labels = scales::dollar) +

scale\_x\_datetime(date\_labels = "%Y", date\_breaks = "1 year") +

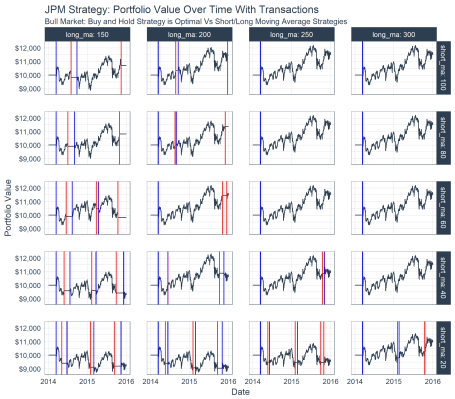
labs(

title = "JPM Strategy: Portfolio Value Over Time With Transactions",

subtitle = "Bull Market: Buy and Hold Strategy is Optimal Vs Short/Long Moving Average Strategies",

x = "Date", y = "Portfolio Value"

)



**Conclusions**

We’ve covered a lot of ground in this article. Congrats if you’ve made it through. You’ve now been exposed to three cool packages:

1. tibbletime: For time-series in the tidyverse
2. furrr: Our parallel processing extension to purrr
3. flyingfox: Our experimental package brought to you as part of our Business Science Labs initiative

Further, you’ve seen how to apply all three of these packages to perform grid search backtest optimization of your trading algorithm.