Beyond the Numbers: A Trader's Guide to Statistical Arbitrage and Convex Optimization

Contents

Introduction

This ebook consolidates work that was originally published as a series of articles on robotwealth.com on statistical arbitrage.

It introduces practical considerations for pairs trading as I encountered it in professional trading environments. The simplicity contained in these approaches may be surprising.

The book then discusses approaches to general statistical arbitrage - which I define as a framework for trading of a basket of related assets, and may or may not include both cross-sectional and time-series predictors. Also included are some ideas for predictive features for cryptocurrency trading.

We then introduce important aspects of researching features for general statistical arbitrage: quantifying alpha strength and decay characteristics. We also consider how we might combine several alphas, including both cross-sectional and time-series features.

The remainder of the book is concerned with the question of how to trade a set of statistical arbitrage features in the face of trading costs and co-movement of assets. In general terms, this is a problem of

navigating the trade-offs between harnessing our edges and incurring trading costs, as well as considering portfolio volatility (and external constraints such as leverage limits).

For the practical purposes of the trader, the main consideration is the trade-off between our *uncertain* edges and our *certain* costs of trading.

Ideally, we want to find the sweet spot where we do enough trading to harness our edges, but not so much that we get killed by costs. Consider the extreme case of a predictive but hyperactive signal: it may do a good job of predicting short-term price changes, but if we trade it naively, our costs can be greater than our edge. Are there times when it makes sense to not trade into the theoretically optimal position (from an expected return perspective), and instead maintain an existing positive-expected-return-but-not-optimal position?

There are a number of approaches to navigating this problem. In the book, we explore two.

We first explore a simple heuristic approach for minimising trading that we call the "no-trade buffer." Essentially, instead of constantly rebalancing back to ideal positions, we only rebalance if our existing positions get out of whack by some defined amount.

This approach is simple to reason about and easy to implement. On the other hand, it doesn't explicitly consider asset co-movement or external constraints.

Another approach is to frame the problem as a mathematical constrained optimisation and then use convex optimisation techniques to figure out how to trade in the next period.

This approach is a little harder to reason about and implement, but it does explicitly consider asset co-movement and portfolio volatility (or other definitions of risk) and external constraints.

It requires your features to be modelled as expected returns. So the book includes an exploration of this topic.

The book also includes a section on building intuition for convex optimisation and a practical guide to using CVXR for framing general statistical arbitrage trading problems. CVXR is a high-level, user-friendly modelling library for convex optimisation problems.

Finally, we show how to simulate trading with convex optimisation, including how to explore the lambda-tau parameter space for the classic mean-variance with costs framework

The code for the book is all written in R, which is a wonderfully productive language for doing data analysis.

A Real World Take on Pairs Trading

In textbooks, one often sees pairs trading algorithms start by regressing prices of Asset A on Asset B to calculate a hedge ratio.

I've rarely seen anyone actually do this in the real world.

That's because it is a very unstable thing – especially for a pair of volatile assets, and especially over a large amount of data.

The basic pairs trading algorithm which you see out in the real world doesn't attempt to do this. Instead, you do the following (or something like it):

- calculate the ratio of the stock prices
- apply a moving average to that ratio
- apply standard deviation to that ratio
- calculate z-score levels to trade

- when placing a trade, allocate the same margin to both legs at the price when you open the trade. (i.e. assign equal $ weight to each leg if it is a stock)

The thinking is the following…

> If it reverts, it reverts, however I calculate the hedge ratio – so I'm just going to assign equal risk exposure to each leg and save myself some hassle.

This simple approach tends to work well for equities and similar futures contracts. (i.e. different stock indexes or bonds of different durations).

Where you have two totally different things (you're spreading JGBs against copper because you're a maniac, for example… and yes, I did this once upon a time….) then you can still make the simple approaches work by weighting the ratio by something sensible like the ratio of the realised or implied volatility of each leg.

Conclusion

The main lesson from this is that the smartest-seeming thing is often not the best in trading.

In academic land, you often see people calculating hedge ratios using dynamic linear models (Kalman filters, etc), copulas, genetic algorithms, etc. When I started out, I did all of these things as well.

In the real world, where implementation, scalability and profitability are prioritised, the simpler approach tends to win out.

A General Approach for Exploiting Statistical Arbitrage Alphas

Statistical arbitrage is a well-understood concept: find pairs or baskets of assets you expect to move together, wait for them to diverge, and bet on them converging again.

Simple enough.

But making it work, especially at scale, is a little more complicated.

A somewhat old-school approach takes pairs of correlated assets and trades them long-short together. *This is the classic pairs trade.*

But we don't have to constrain ourselves to this approach.

And when we consider a more general approach, we find exploitable opportunities that may not work under the classic pairs trading paradigm.

For instance, trading a long-short portfolio of equity pairs is expensive and margin-intensive. The more general statistical arbitrage approach I'll discuss here will greatly reduce costs and margin requirements.

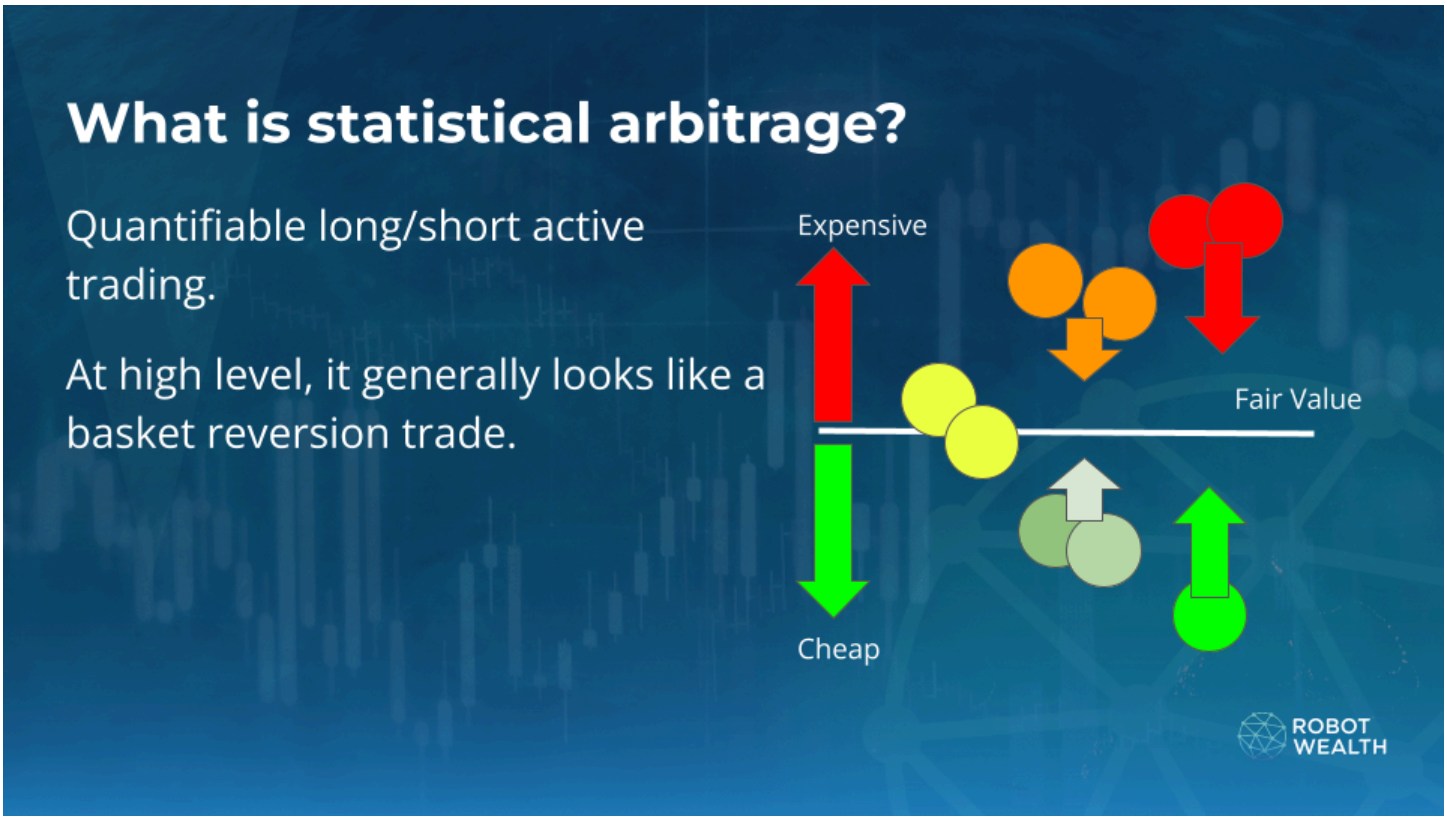
In this article, I'll share a general approach for managing a portfolio of statistical arbitrage trades. It involves navigating the trade-offs around maximising expected returns while minimising portfolio variance and trading costs.

Generalizing Statistical Arbitrage

When I use the term "statistical arbitrage", I'm referring to any quantifiable, systematic, long-short, active trading.

A classic example is the ***basket reversion trade***.

Imagine you have a bunch of assets and some criteria that allow you to predict whether those assets are expensive or cheap.



In practical terms, "expensive" means you expect it to go down, and "cheap" means you expect it to go up.

Then, you can rank all of your assets according to your criteria and aim to be:

- Very short the stuff that's very expensive
- Slightly short the stuff that's a bit expensive
- Slightly long the stuff that's a bit cheap
- Very long with stuff that's very cheap

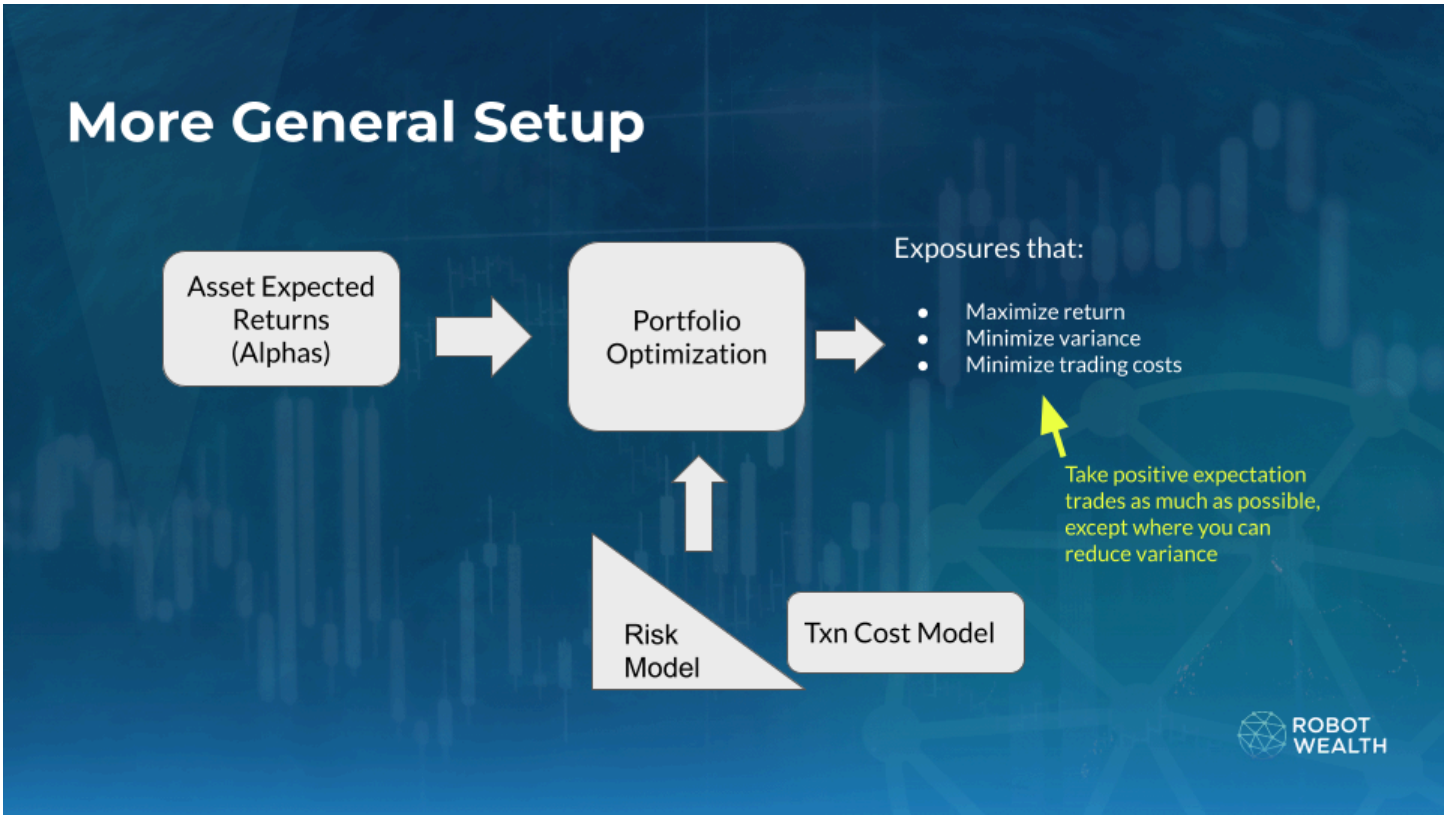
The trade also seeks to minimize portfolio return variance by being about as short as it is long a bunch of similar assets. In other words, you aim to isolate the alpha embedded in the divergence of these similar assets and cancel out everything else.

So, you can consider it a reversion trade of effectively everything in the basket converging with fair value.

Of course, the implementation details can get a little complicated, but conceptually, it’s a very simple trade.

Conceptual Implementation of Statistical Arbitrage

Here's a general setup for implementing a statistical arbitrage portfolio:

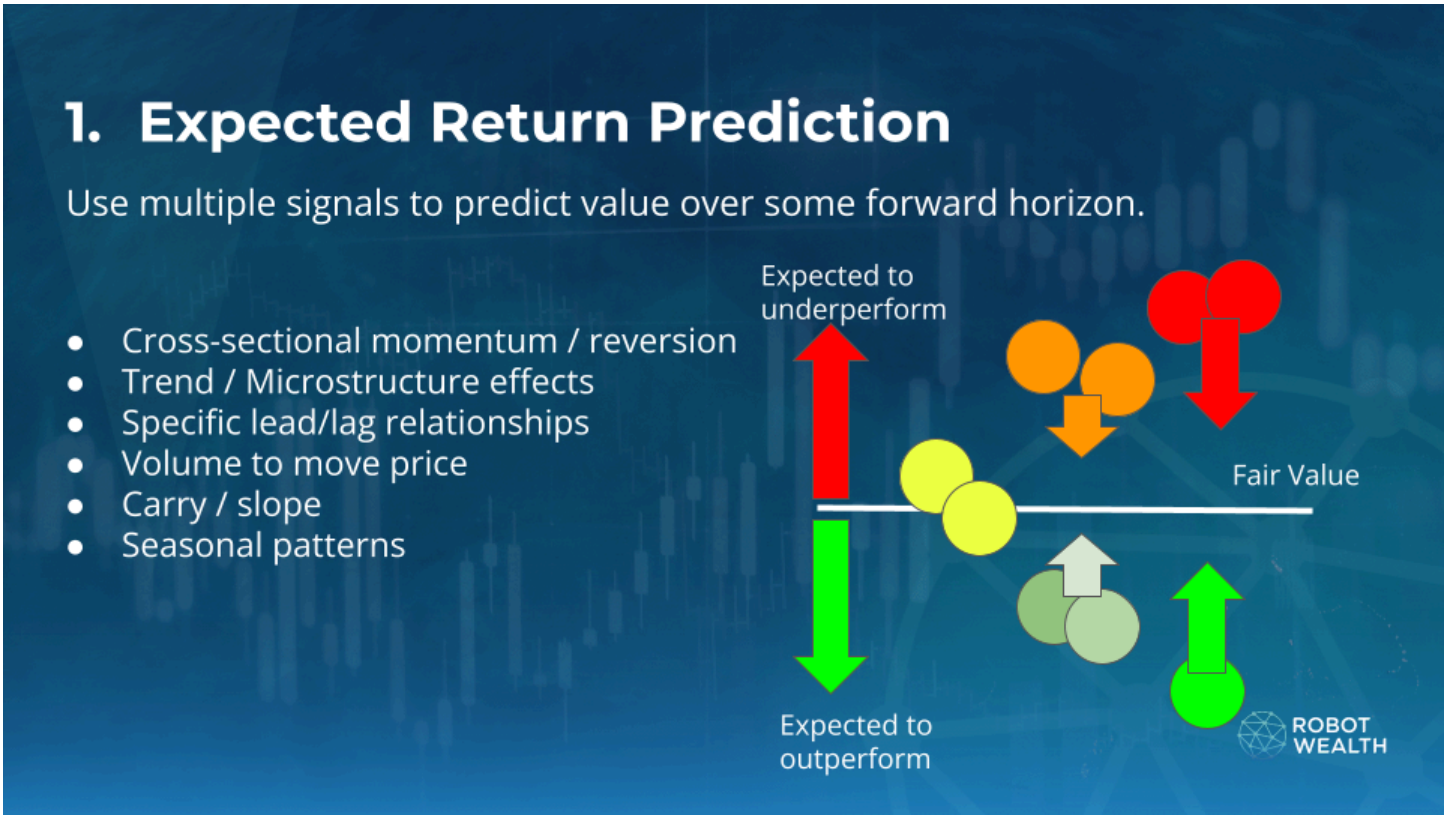


Let's go through all the parts.

1. Estimate Expected Returns

The first step is to calculate expected returns. This is the most difficult and important part.

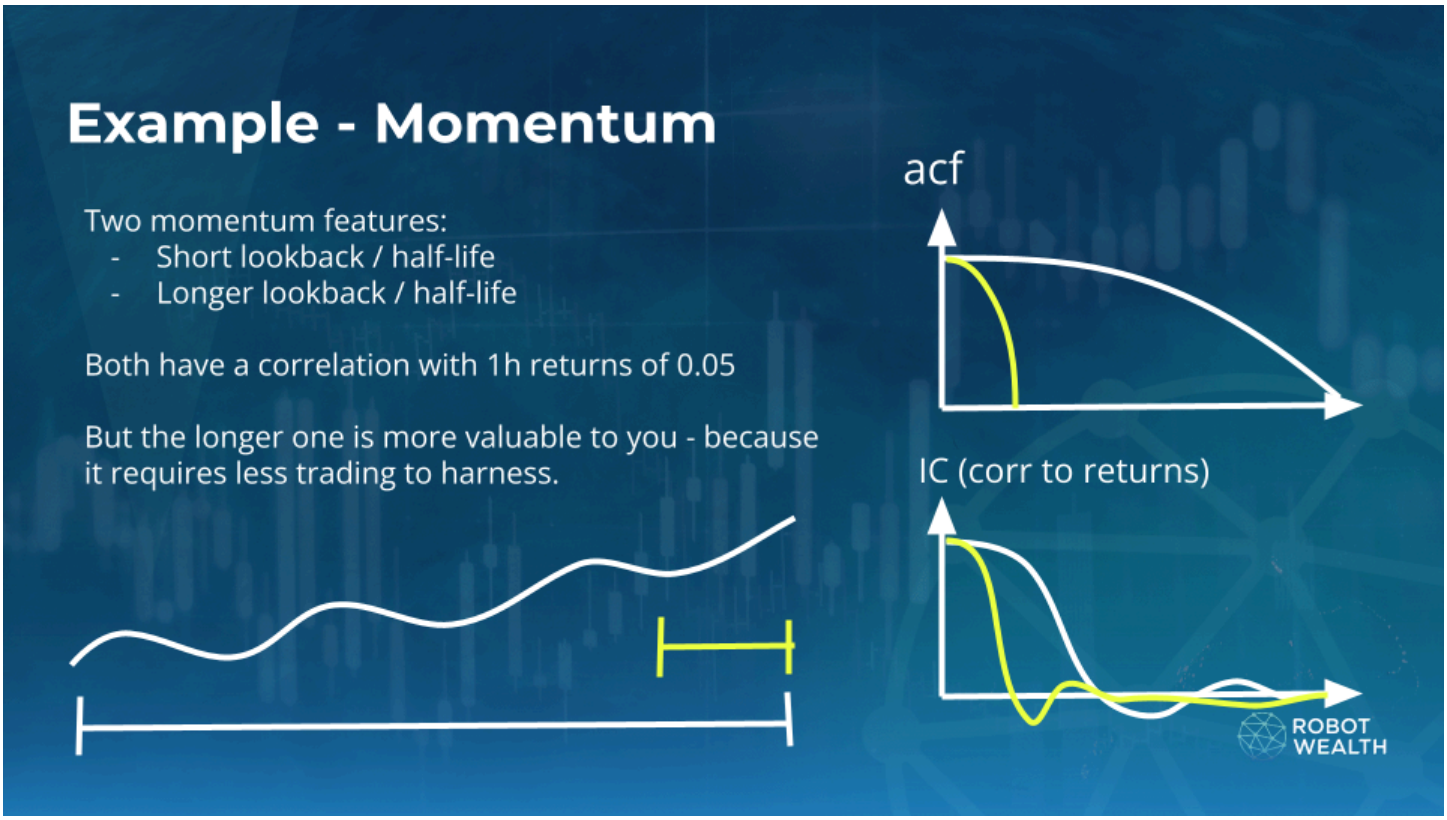
Your expected returns model will generally be the output of an extensive research process where you identify various features that predict future returns. Examples might include cross-sectional momentum or mean-reversion, microstructure effects, lead-lag relationships, carry, etc.



One useful metric for quantifying the effectiveness of a feature is the information coefficient, which measures the correlation between the feature and step-ahead returns.

We also like to consider how quickly the feature's predictiveness decays over time. Alphas with high information coefficients are great, but in practice, we prefer alphas that decay slowly, as they are more forgiving to trade (rely less on execution skill), turnover less frequently, and can be harnessed with less trading (all things being equal).

Consider two momentum features as an example.



- The two features both have a correlation coefficient with hour-ahead returns of 0.05.
- The first feature has a shorter lookback period and decays quickly.

- The second feature has a longer lookback period and decays more slowly.

The second feature is more valuable because it requires less trading to harness.

We also prefer alphas that are uncorrelated with other alphas – these tend to give you more bang for your buck at the portfolio level.

Say you have several predictive features with different information coefficients and decay characteristics. You then need to combine them into an expected return for each asset. This usually involves up-weighting or down-weighting the different alphas depending on their characteristics, and perhaps scaling them to an understood range, such as using a z-score.

Useful tools for understanding your alphas and informing good decisions about how you combine them include:

- Scatterplots
- Ranks and sorts
- Information coefficient
- Information coefficient decay
- Autocorrelation function

This is a fairly big topic in itself. I'll revisit it with some specific examples in a future article.

2. Minimising Variance

In traditional pairs trading, we would trade equally weighted long and short positions in correlated assets. However, in a more general statistical arbitrage approach, we can take advantage of asset relationships to reduce portfolio risk.

For example, instead of maximizing expected returns, we can consider the co-movement of different assets and adjust our positions accordingly. This means we may end up shorting assets that we expect to go up, but not as much as other assets, to reduce volatility.

If you were maximising expected returns, you'd simply get long the stuff you expect to go up the most.

But if we're maximising risk-adjusted returns, shorting assets correlated with the highest expected return assets makes sense, even if we expect those to go up as well. Doing so essentially isolates the divergence and nets off the other risks – at least to the extent that our assets are exposed to the same risks.

As an aside, why would it be smart to reduce volatility? Because this tends to increase your long-run geometric expected returns, which lets you compound your money more effectively.

Including a risk model for the purposes of reducing portfolio variance usually implies doing some sort of constrained mean-variance optimisation (*constrained* because we usually want to consider real-world constraints such as leverage limits, beta-neutrality, or whatever else we are beholden to).

But the risk model doesn't need to be based on an explicit optimisation – you can, for example, use simple heuristics such as equal dollar weight long-short, or slightly more complicated heuristics that reduce target exposures in assets that are on average more correlated to other things in the portfolio.

This is another fairly big topic and I'll show you some examples in a future post.

3. Minimising Trading Costs

Reducing turnover is also crucial in making a statistical arbitrage portfolio work in the real world.

Every time you trade, you incur a cost – fees, spreads, market impact. And our costs are largely known (especially if we're trading small and can ignore market impact). In contrast, our returns are uncertain (that's why we refer to them as *expected*).

This implies that we don't want to be naively whacking an edge – we need to do so in a cost-aware manner.

Here's a really simple example to make it clear:

Consider three assets that have the following expected returns over the next day:

- Asset A: +0.5%
- Asset B: 0%
- Asset C: -0.1%

What's the trade?

Well, assuming no interesting correlation relationships, you'd long asset A and short asset C.

You expect asset A to go up the most, and you expect asset C to go down the most. So that's an easy trade.

But when you've actually got a position on, it's not quite so clear-cut anymore.

We're in a position where we're long asset A and short asset C. Now the next day, we run our features again and obtain our expected return predictions:

- Asset A: +0.5%
- Asset B: -0.12%
- Asset C: -0.1%

Now we're expecting the same return for asset A – a 0.5% increase. Asset C is also the same – a 0.1% decrease.

But our prediction for asset B is now -0.12%. It has the most negative expected return.

So if we had no positions on, we ***wouldn't*** enter the positions we actually have. We would buy asset A, but we'd sell asset B.

But we are currently long asset A and short asset C.

If trading were free, we would get out of asset C and sell asset B instead. But trading isn't free.

By switching positions, we would get an increase in expected returns of 0.02%. But returns are uncertain and there's usually a huge amount of variance associated with our prediction.

On the other hand, know for a fact that trading out of C and into B is going to cost us money.

Now the question is, is the expected return that we get from shifting position significantly bigger than the known costs that we pay from shifting position? And in this case, we probably wouldn't do the trade because our trading costs are very likely to be bigger than the expected return increase that we get from switching positions.

In reality, we need the expected return increase to be significantly bigger than our trading costs because we've got errors in our estimates. *Our estimates are a distribution*. You always want to give yourself some wiggle room.

You can make a similar example for a trade that would reduce portfolio variance. Depending on the costs of trading and the expected reduction in variance, you may be better off sticking with an existing position, especially if the existing exposure is correlated with the ideal exposure.

The main takeaway is that the positions that we have are a significant consideration (in addition to return and risk predictions) in what trading we should do to rebalance into new positions.

Like the variance reduction step, you can incorporate trading costs using simple heuristic rules or explicit optimisation. Again, I will show you some examples of this in a future article.

Managing Trade-offs

The approach I've outlined here is essentially an exercise in maximising return on investment by smart management of the trade-offs between expected returns, risk and trading costs.

Finding some predictive features is the most important part of the process because without a good return prediction, you simply don't have a trade. Minimising variance and reducing turnover is not alpha.

But having a good return prediction is usually not enough – you need to be clever in how you harness it such that you maximise your long-term returns in the face of trading costs and whatever other constraints you're under.

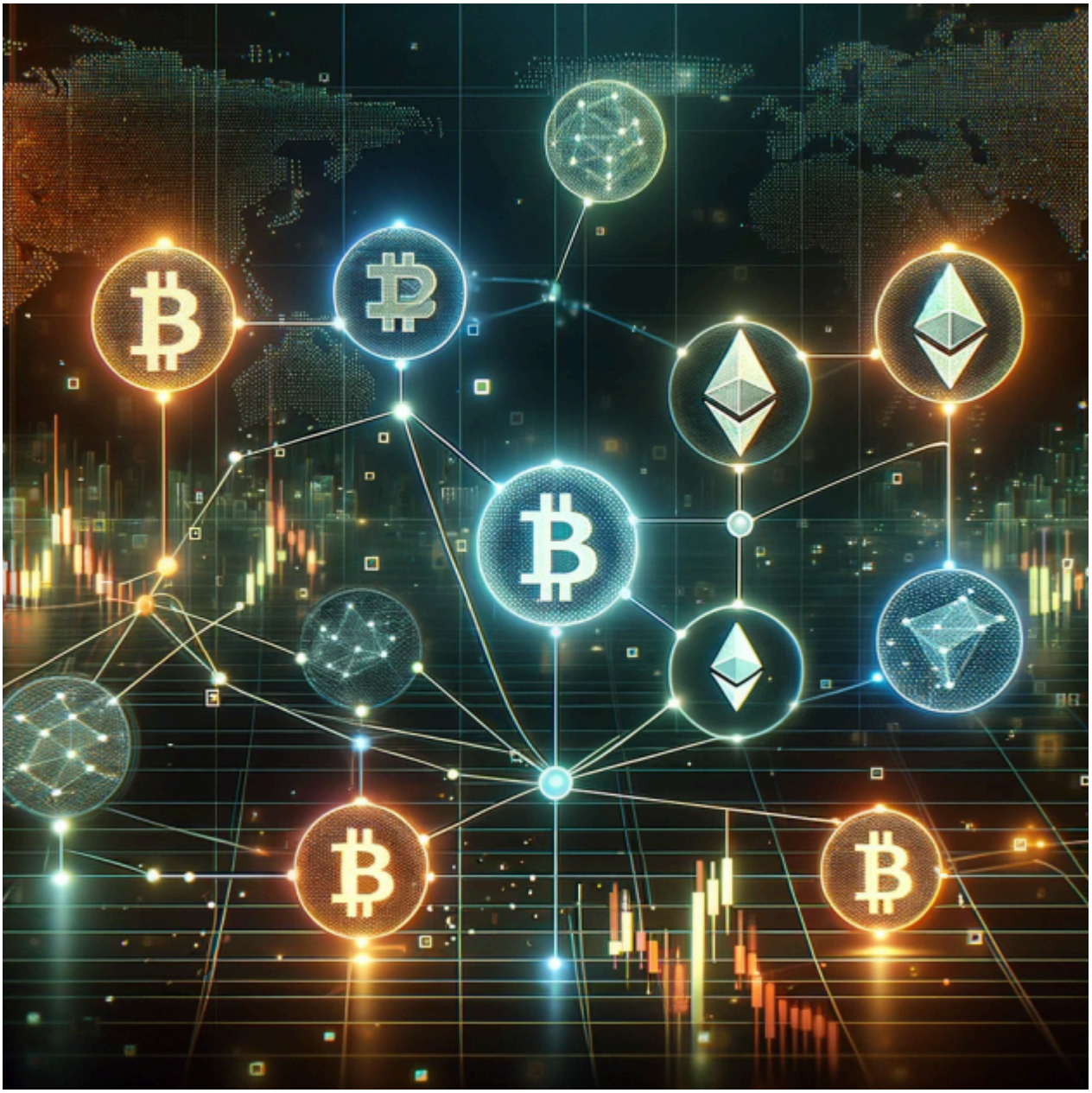
Statistical Arbitrage in Practice

In this article, I outlined a general approach to statistical arbitrage trading. But I left out many of the practical details, including:

- Ideas for predictive features
- Examples of exploring these features and quantifying their information coefficients and decay characteristics
- Combining alphas with different ICs and decay characteristics
- Using heuristics to navigate the return-risk-cost trade-off
- Using constrained optimisation to navigate the return-risk-cost trade-off

We'll explore many of these in the remainder of the book.

Ideas for Crypto Stat Arb Features



In this article, I'll brainstorm some ideas for predictive features that you could potentially use in a crypto stat arb model.

The ideas draw insights from recent discussions and market observations, but of course, you should do your own research.

In future articles, I'll pick some of these features and seek to quantify and combine them, and finally demonstrate how to navigate the return-risk trade-offs using heuristics and explicit optimisation.

But first, let’s brainstorm some ideas.

Understanding Relative Movements

The classic stat arb approach capitalises on relative movements between similar assets.

In particular, fading price-insensitive buying or selling that pushes an asset’s price away from fair value tends to be a good bet.

Trend Effects and Momentum

Apart from relative movements, identifying predictable short-term and long-term trend effects can be useful, particularly in less efficient markets.

An interesting example is the concept of crowded spreads breaking. When many traders occupy a spread, and it diverges further and further, it can lead to violent momentum as everyone gets forced out of it.

If you’re skilled, you can build these effects into your stat arb models.

Lead-Lag Relationships

The crypto market sometimes exhibits specific lead-lag relationships, where the price movement of one asset is predicted by the movement of another.

An interesting example was the lead-lag relationship between Bitcoin and the S&P 500.

I don't know if this still exists, but when I last looked in 2022, it was observable in the data.

If you look at the intraday correlations of those things, you'll see that what probably happened was that someone put SPX returns into their market-making fair value algorithm, which ended up creating a structural lead-lag correlation effect between these two assets.

You can imagine that if you've got people moving quotes on the basis of recent moves in some asset, then you've effectively physically generated a lead-lag relationship between those things – because a big part of price movement is quote revision.

Volume and Price Movement

Another important idea, especially in less liquid markets, is understanding the volume required to move price.

By analyzing the relationship between the historical volume of aggressive buying or selling and the resultant price change, you may be able to infer the presence of invisible supply or demand.

For example, you might look at the historical data and find that, on average, $100k of aggressive buying over the last minute created a price change of 0.2% (those numbers don't mean anything; they're just to make the example more real).

Now, imagine you see $1 million of aggressive buying, and it only moves price 0.2% over the next minute. That would suggest that there is an invisible supply in the market that's being reloaded as an iceberg order or similar.

All things being equal, you'd rather fade that move – because unless more demand comes in, you infer that there's an imbalance.

On the other hand, we did some analysis back in 2022 that suggested that, *on average*, crypto price moves on larger volumes were more likely to continue.

Often, these trend/reversion effects play out over different time horizons, and how you incorporate them into your model requires some careful analysis.

Carry

Carry is also very interesting in crypto.

You would naively expect that futures trading at a premium to spot are, on average, more likely to go down, while those trading at a discount tend to go up.

However, when we looked at carry in crypto, we found that total returns (price change plus funding) were correlated with the futures premium, with some potentially interesting and different dynamics in the tails.

So, carry is potentially an interesting feature to include in your crypto stat arb models.

In addition, *changes* in carry might be worth looking at.

For example, say you're looking at a certain crypto spot market and its perpetual contract. The perpetual gets bid up 1%. You go and check what's happened in the spot market. You'd probably think differently about the continuation of that move if the spot hadn't moved than if they'd both been bid up together.

Seasonal Patterns

Lastly, seasonal patterns can be interesting, especially in less liquid assets. These can be due to cultural factors or known trading activities like rebalance trades. These patterns can be useful inputs to stat arb models.

Conclusion

In this article, we brainstormed some potential predictive features for a crypto stat arb model.

We came up with:

- Relative price moves
- Predictable trend effects
- Lead-lag relationships
- Volume and price change
- Carry
- Seasonal patterns

The next step would be to do some data analysis that attempts to understand these features in detail, ideally quantifying their historical strength and decay characteristics. We'll tackle this next.

Quantifying and Combining Crypto Alphas

In this article, I'll take some crypto stat arb features from the previous chapter and show you how you might quantify their strength and decay characteristics and then combine them into a trading signal.

Ultimately, combining signals is about making sensible decisions about how you want each signal to contribute to the calculation of portfolio weights. And to make those decisions, you need to understand your signals:

- Do they need to be scaled or transformed in some way to be useful?
- How strongly are they predictive of forward returns?
- How quickly does this predictive power decay?
- How noisy is the signal? Does it bounce around all over the place or is it relatively stable?
- How correlated are your signals to one another?

One you understand these questions, you can start to consider how to combine your signals. And there are no right or wrong answers, just different paths through the various trade-offs depending on your goals and constraints. When considering costs, the questions around signal decay and stability become critical.

And you might combine your signals using simple heuristics (equal weight, equal volatility contribution), or by selecting coefficients based on your understanding of how the signals behave. You can also use explicit optimisation techniques, but you don't have to.

In this article, we'll combine some cross-sectional signals with a time-series one, which has the effect of tipping the portfolio net long or short - which may or may not be acceptable depending on your constraints.

I'll include all the code and link to the data so that you can follow along if you like.

Let's get to it.

First, load some libraries and set our session options:

```
# session  options
options(repr.plot.width = 14, repr.plot.height=7, warn = -1)

library(tidyverse)
library(tibbletime)
library(roll)
library(patchwork)

# chart options
theme_set(theme_bw())
theme_update(text = element_text(size = 20))
```

```
── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
✓ dplyr     1.1.4     ✓ readr     2.1.4
✓ forcats   1.0.0     ✓ stringr   1.5.1
✓ ggplot2   3.4.4     ✓ tibble    3.2.1
✓ lubridate 1.9.3     ✓ tidyr     1.3.1
✓ purrr     1.0.2
── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becc

Attaching package: 'tibbletime'


The following object is masked from 'package:stats':

    filter
```

Next load some data. This dataset includes daily price, volume, and funding data for crypto perpetual futures contracts traded on Binance since late 2019.

You can get the data here.

```
perps <- read_csv("https://github.com/Robot-Wealth/r-quant-recipes/raw/master/quantifying-c
head(perps)
```

```
Rows: 187251 Columns: 11
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr  (1): ticker
dbl  (9): open, high, low, close, dollar_volume, num_trades, taker_buy_volum...
date (1): date

ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

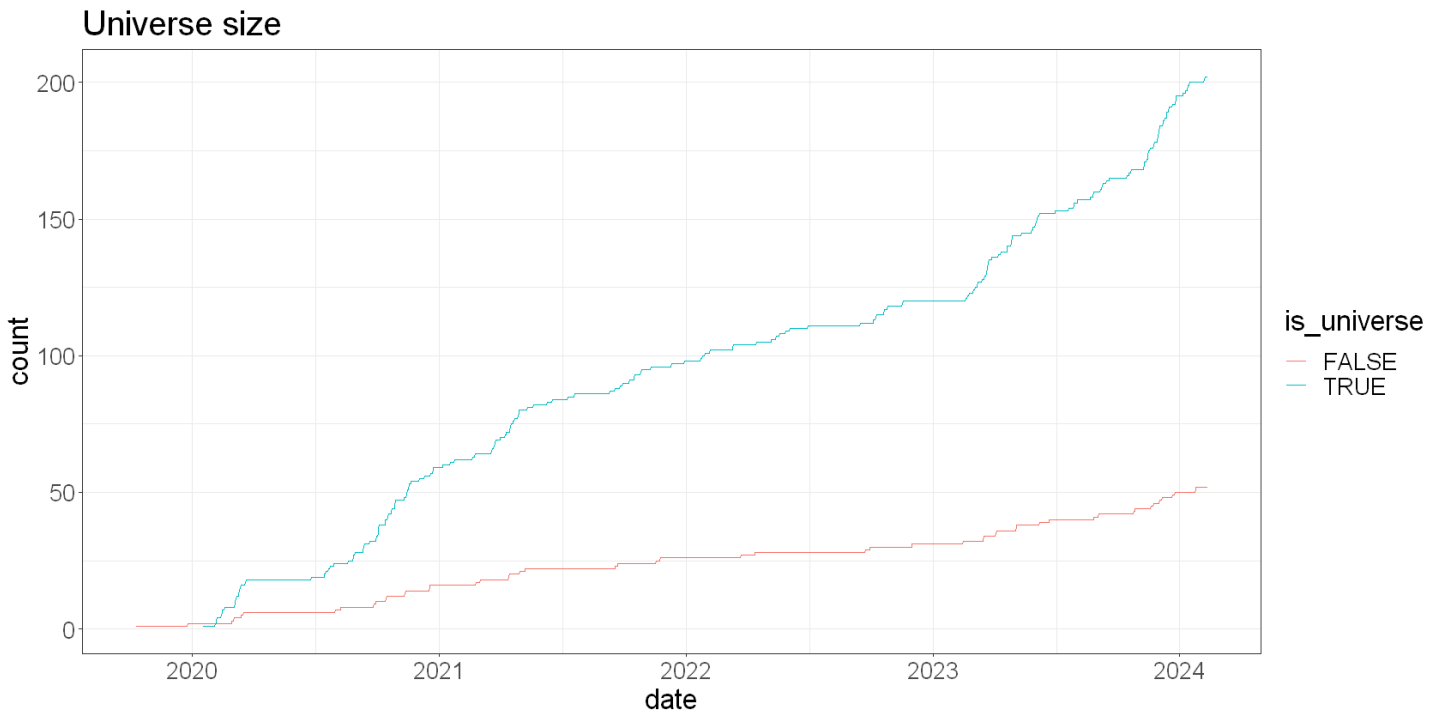
A tibble: 6 × 11

ticker	date	open	high	low	close	dollar_volume	num_trades	ta
<chr>	<date>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	
BTCUSDT	2019-09-11	10172.13	10293.11	9884.31	9991.84	85955369	10928	
BTCUSDT	2019-09-12	9992.18	10365.15	9934.11	10326.58	157223498	19384	
BTCUSDT	2019-09-13	10327.25	10450.13	10239.42	10296.57	189055129	25370	
BTCUSDT	2019-09-14	10294.81	10396.40	10153.51	10358.00	206031349	31494	
BTCUSDT	2019-09-15	10355.61	10419.97	10024.81	10306.37	211326874	27512	
BTCUSDT	2019-09-16	10306.79	10353.81	10115.00	10120.07	208211376	29030	

First, let's constrain our universe by just removing the bottom 20% by trailing 30-day dollar volume.

How many perps are in our universe?

```r
universe <- perps %>%
  group_by(ticker) %>%
  mutate(trail_volume = roll_mean(dollar_volume, 30)) %>%
  # also calculate returns for later
  mutate(
    total_fwd_return_simple = dplyr::lead(funding_rate, 1) + (dplyr::lead(close, 1) - close
    total_fwd_return_simple_2 = dplyr::lead(total_fwd_return_simple, 1),  # next day again
    total_fwd_return_log = log(1 + total_fwd_return_simple)
  ) %>%
  na.omit() %>%
  group_by(date) %>%
  mutate(
    volume_decile = ntile(trail_volume, 10),
    is_universe = volume_decile >= 3
  )

universe %>%
  group_by(date, is_universe) %>%
  summarize(count = n(), .groups = "drop") %>%
  ggplot(aes(x=date, y=count, color = is_universe)) +
  geom_line() +
  labs(
    title = 'Universe size'
)
```



Some simple features

We'll look at three very simple features implied by our brainstorming session.

- Breakout - closeness to recent 20 day highs: (9.5 = new highs today / -9.5 = new highs 20 days ago)
- Carry - funding over the last 24 hours
- Momentum - how much has price changed over the last 10 days

My intent is to use the carry and momentum features as cross-sectional predictors of returns. That is, does relative carry/momentum predict out/under-performance?

And I'll use the breakout feature as a time-series overlay. I hope to use it to predict forward returns for each asset in the time-series, not in the cross-section.

This is mostly to demonstrate how to combine cross-sectional and time-series features into a single long-short strategy. Adding time-series features to a long-short stat arb basket has the effect of making the basket net long or net short rather than delta neutral, so you may not want to do it, depending on your constraints.

First we calculate the raw features and lag them by one observation:

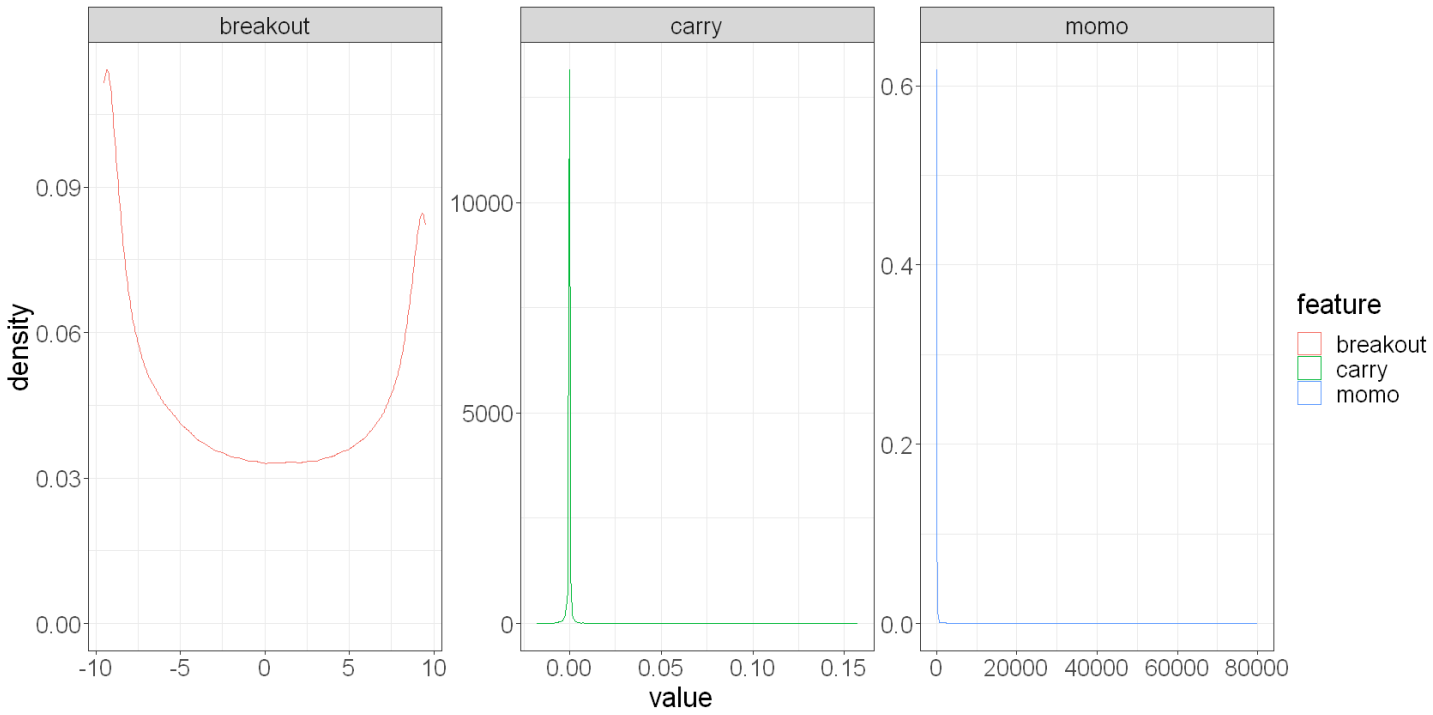
```r
rolling_days_since_high_20 <- purrr::possibly(
  tibbletime::rollify(
    function(x) {
      idx_of_high <- which.max(x)
      days_since_high <- length(x) - idx_of_high
      days_since_high
    },
    window = 20, na_value = NA),
  otherwise = NA
)

features <- universe %>%
  group_by(ticker) %>%
  arrange(date) %>%
  mutate(
    # we won't lag features here because we're using forward returns
    breakout = 9.5 - rolling_days_since_high_20(close),  # puts this feature on a scale -9.
    momo = close - lag(close, 10)/close,
    carry = funding_rate
  ) %>%
  ungroup() %>%
  na.omit()

head(features)
```

ticker	date	open	high	low	close	dollar_volume	num_trades	taker_l
<chr>	<date>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	
BTCUSDT	2019-10-29	9389.98	9569.10	9156.03	9354.63	1207496201	238440	
BTCUSDT	2019-10-30	9352.65	9492.86	8975.72	9102.91	1089606006	267844	
BTCUSDT	2019-10-31	9103.86	9438.64	8933.00	9218.70	851802094	229216	
BTCUSDT	2019-11-01	9218.70	9280.00	9011.00	9084.37	816001159	233091	
BTCUSDT	2019-11-02	9084.37	9375.00	9050.27	9320.00	653539543	204338	
BTCUSDT	2019-11-03	9319.00	9366.69	9105.00	9180.97	609237501	219662	

Exploring our features

The first thing you'll generally want to do is get a handle on the characteristics of our raw features. How are they distributed? Do their distributions imply the need for scaling?

```
features %>%
  filter(is_universe) %>%
  pivot_longer(c(breakout, momo, carry), names_to = "feature") %>%
  ggplot(aes(x = value, colour = feature)) +
  geom_density() +
  facet_wrap(~feature, scales = "free")
```



We see that the breakout feature is not too badly behaved. Carry and momentum both have significant tails however, so we're going to want to scale those features somehow.

We'll scale them cross sectionally by:

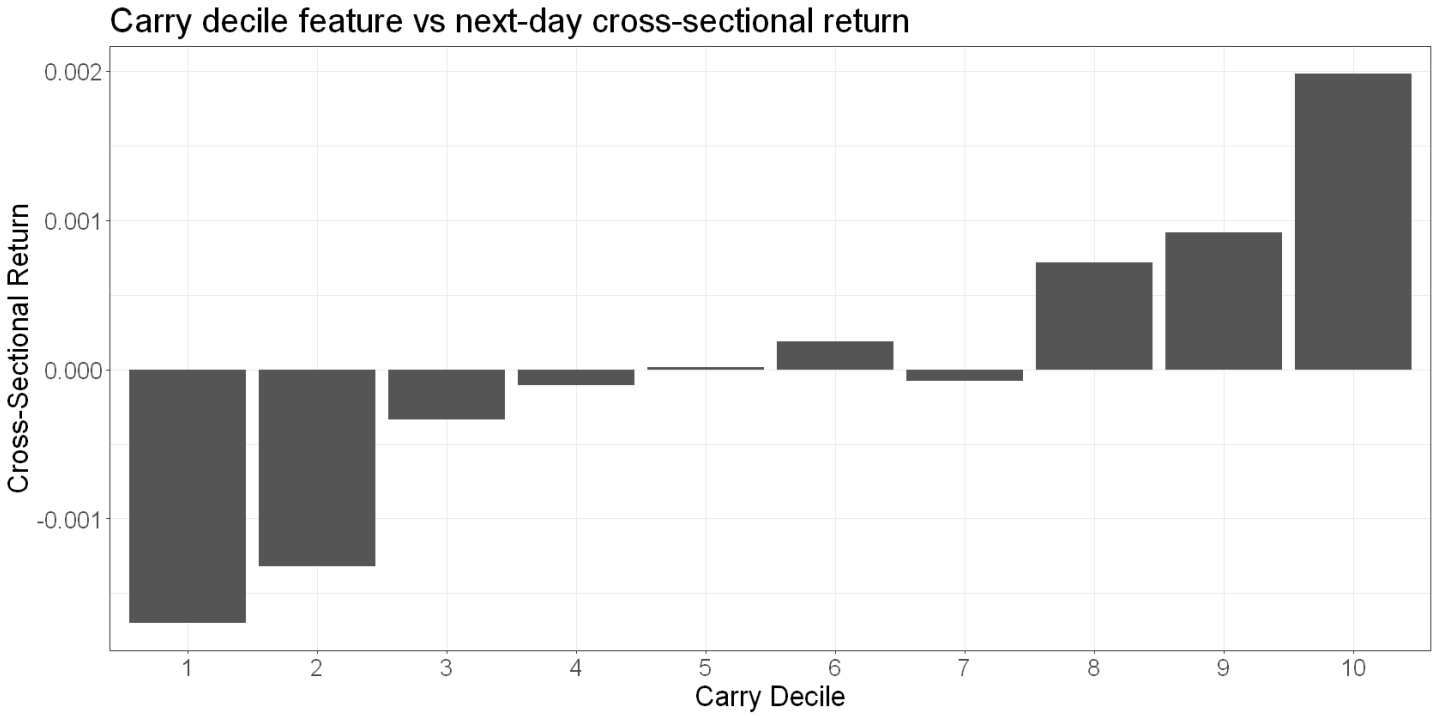
- calculating a zscore: the number of standard deviations each asset is away from the daily mean
- sorting into deciles: sort the assets into 10 equal buckets by the value of the feature each day - this collapses the feature value down to a rank for each asset each day (multiple assets will have the same rank).

We'll also calculate relative returns so that we can explore our features' predictive utility in the time-series and the cross-section.

```
features_scaled <- features %>%
  filter(is_universe) %>%
  group_by(date) %>%
  mutate(
    demeaned_fwd_returns = total_fwd_return_simple - mean(total_fwd_return_simple),
    zscore_carry = (carry - mean(carry, na.rm = TRUE)) / sd(carry, na.rm = TRUE),
    decile_carry = ntile(carry, 10),
    zscore_momo = (momo - mean(momo, na.rm = TRUE)) / sd(momo, na.rm = TRUE),
    decile_momo = ntile(momo, 10),
  ) %>%
  na.omit() %>%
  ungroup()
```

Next we would plot our features against next day returns. For example, here's a factor plot of the `decile_carry` feature against next day relative returns:

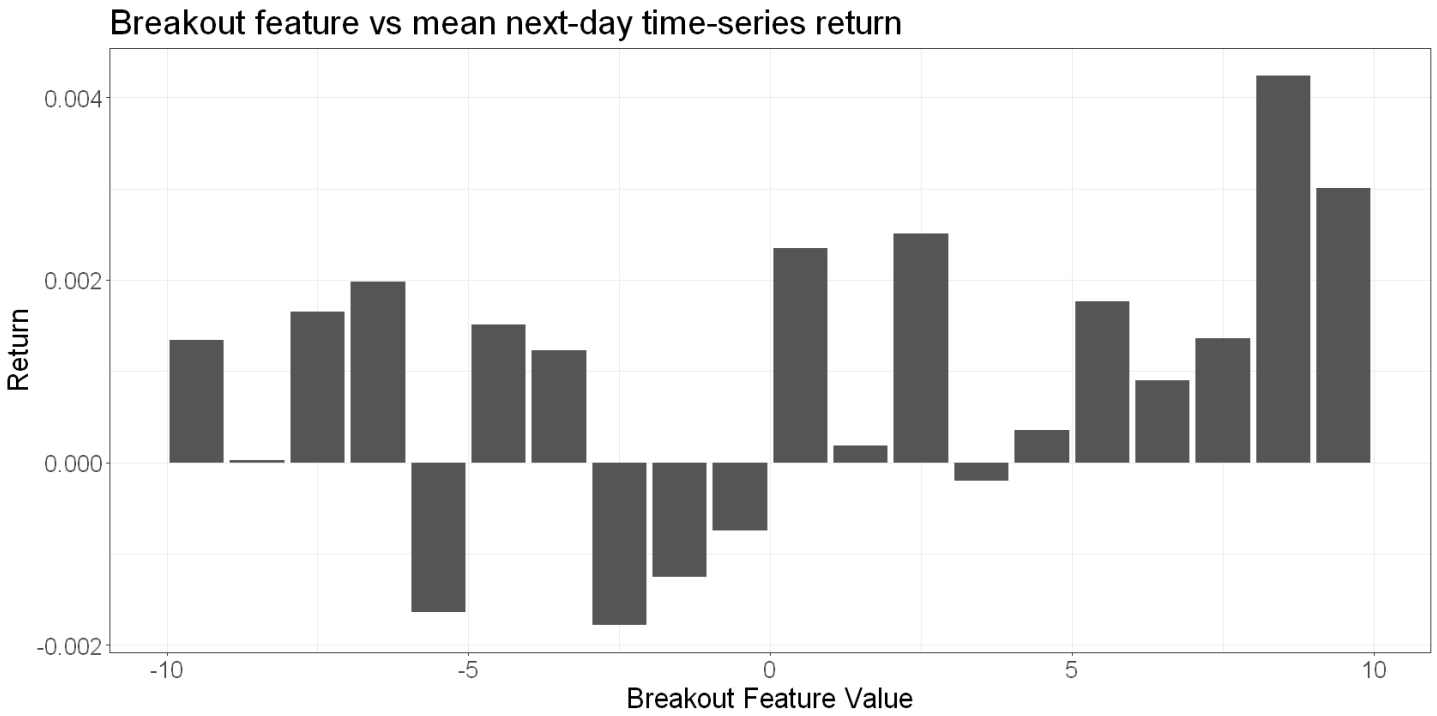
```
features_scaled %>%
  group_by(decile_carry) %>%
  summarise(
    mean_return = mean(mean(demeaned_fwd_returns))
  ) %>%
  ggplot(aes(x = factor(decile_carry), y = mean_return)) +
  geom_bar(stat = "identity") +
  labs(
    x = "Carry Decile",
    y = "Cross-Sectional Return",
    title = "Carry decile feature vs next-day cross-sectional return"
  )
```



We see that it's a good discriminator of the sign of next day returns, and the relationship is noisily linear, but much stronger in the tails.

And here's a plot of the breakout feature against mean next day time-series returns:

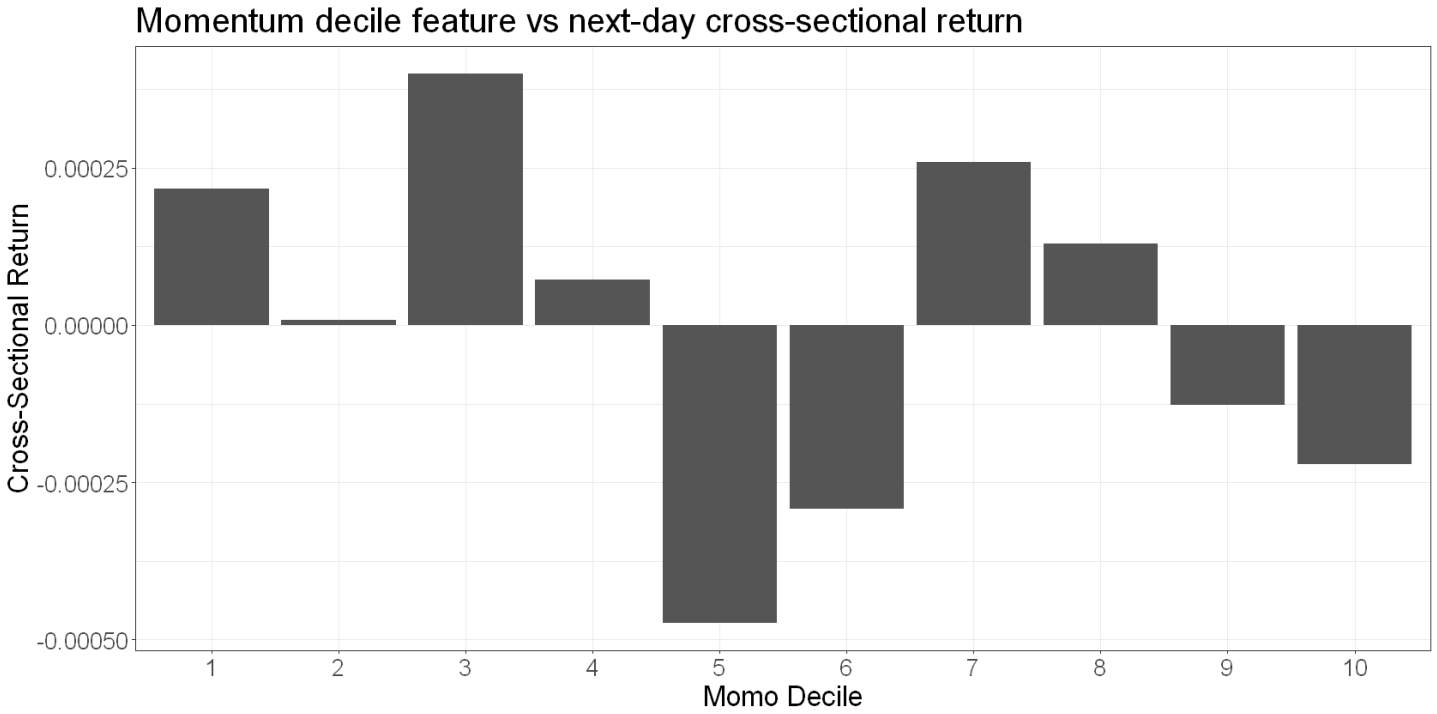
```
features_scaled %>%
  group_by(breakout) %>%
  summarise(
    mean_return = mean(total_fwd_return_simple)
  ) %>%
  ggplot(aes(x = breakout, y = mean_return)) +
  geom_bar(stat = "identity") +
  labs(
    x = "Breakout Feature Value",
    y = "Return",
    title = "Breakout feature vs mean next-day time-series return"
  )
```



We see a pretty noisy and non-linear relationship, but high values of the breakout feature are generally associated with higher returns.

Here's the momentum factor:

```
features_scaled %>%
  group_by(decile_momo) %>%
  summarise(
    mean_return = mean(demeaned_fwd_returns)
  ) %>%
  ggplot(aes(x = factor(decile_momo), y = mean_return)) +
  geom_bar(stat = "identity") +
  labs(
    x = "Momo Decile",
    y = "Cross-Sectional Return",
    title = "Momentum decile feature vs next-day cross-sectional return"
  )
```



We see a very noisy relationship between 10-day cross-sectional momentum and next day relative returns. Interestingly, we see a roughly negative relationship, implying a slight mean-reversion effect rather than momentum.

More work to do

There are all sorts of ways you can view your features.

Here, we've only looked at the mean returns to each discrete feature value, which is useful because by aggregating returns, we get a clear view of the on-average relationships. On the downside however, this approach hides nearly all of the variance. So in practice you'd also rely on other tools - for instance scatterplots and timeseries plots of cumulative next day returns to the feature.

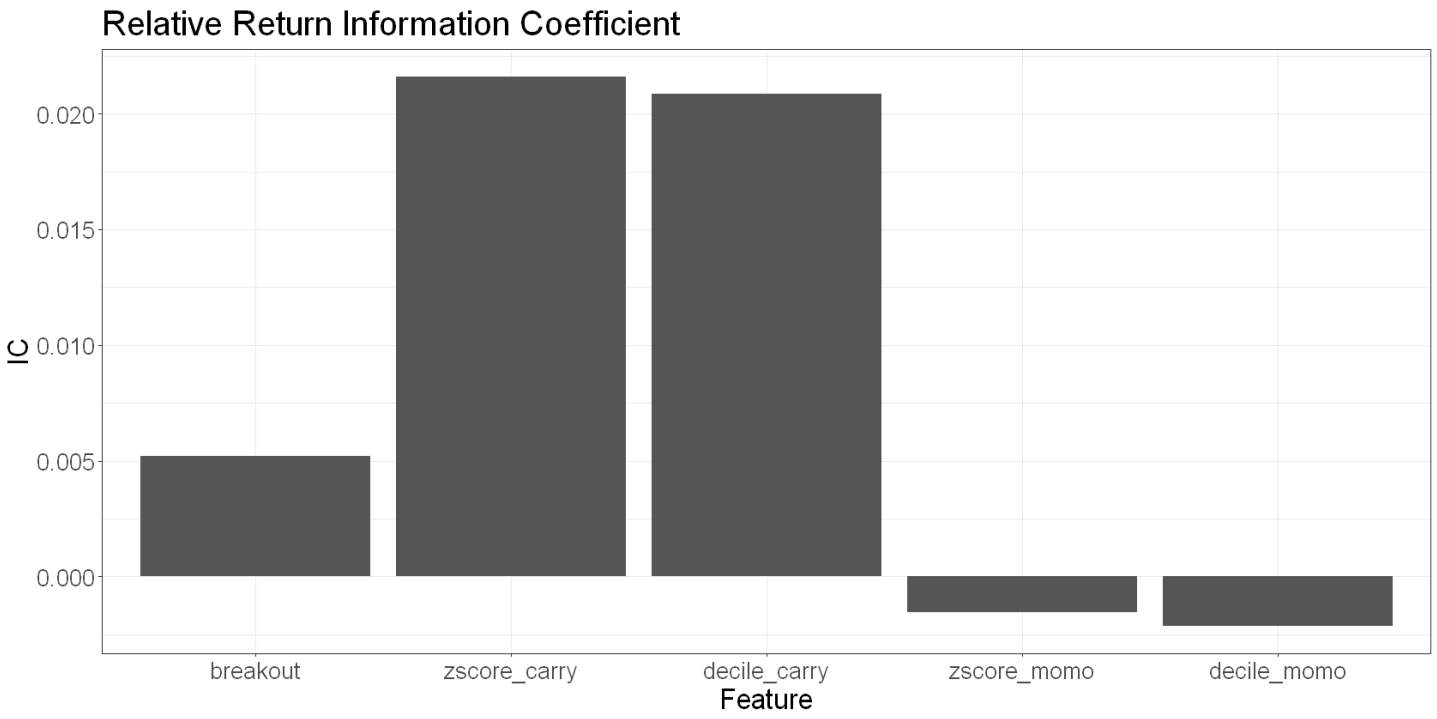
We've also not looked at turnover implied by our signals. This is crucial when you start to consider costs.

You'd also want to look at look at the stability of the factors' performance over time. For example, you may want to down-weight factors that outperformed in the past but have levelled off recently. What we've done here hides that level of detail.

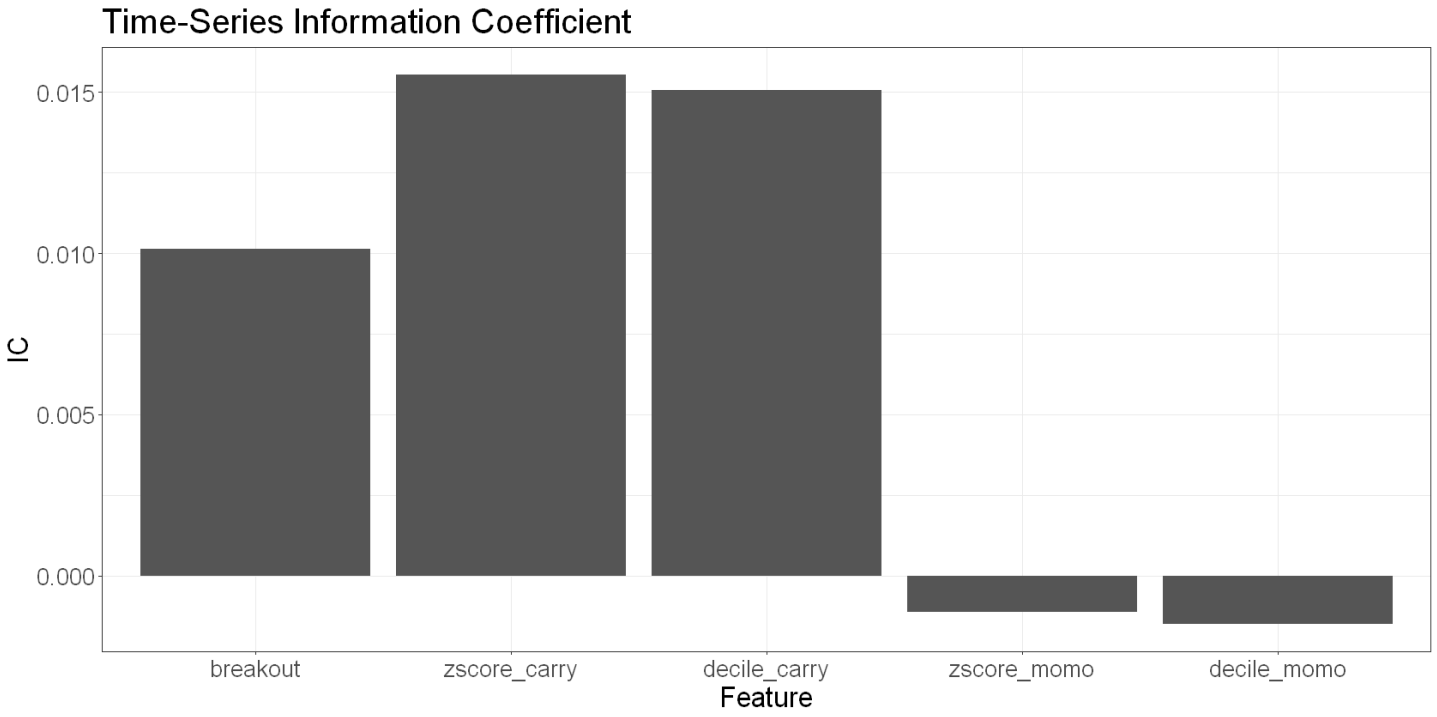
Information coefficient

The information coefficient is simply the correlation of the feature with next day returns. It provides a single metric for quantifying the strength of the signal. We can calculate it for time-series returns and cross-sectional returns:

```
features_scaled %>%
  pivot_longer(c(breakout, zscore_carry, zscore_momo, decile_carry, decile_momo), names_to
  group_by(feature) %>%
  summarize(IC = cor(value, demeaned_fwd_returns))  %>%
  ggplot(aes(x = factor(feature, levels = c('breakout', 'zscore_carry', 'decile_carry', 'zs
  geom_bar(stat = "identity") +
  labs(
    x = "Feature",
    y = "IC",
    title = "Relative Return Information Coefficient"
  )
```



```r
features_scaled %>%
  pivot_longer(c(breakout, zscore_carry, zscore_momo, decile_carry, decile_momo), names_to
  group_by(feature) %>%
  summarize(IC = cor(value, total_fwd_return_simple))  %>%
  ggplot(aes(x = factor(feature, levels = c('breakout', 'zscore_carry', 'decile_carry', 'zs
  geom_bar(stat = "identity") +
  labs(
    x = "Feature",
    y = "IC",
    title = "Time-Series Information Coefficient"
  )
```

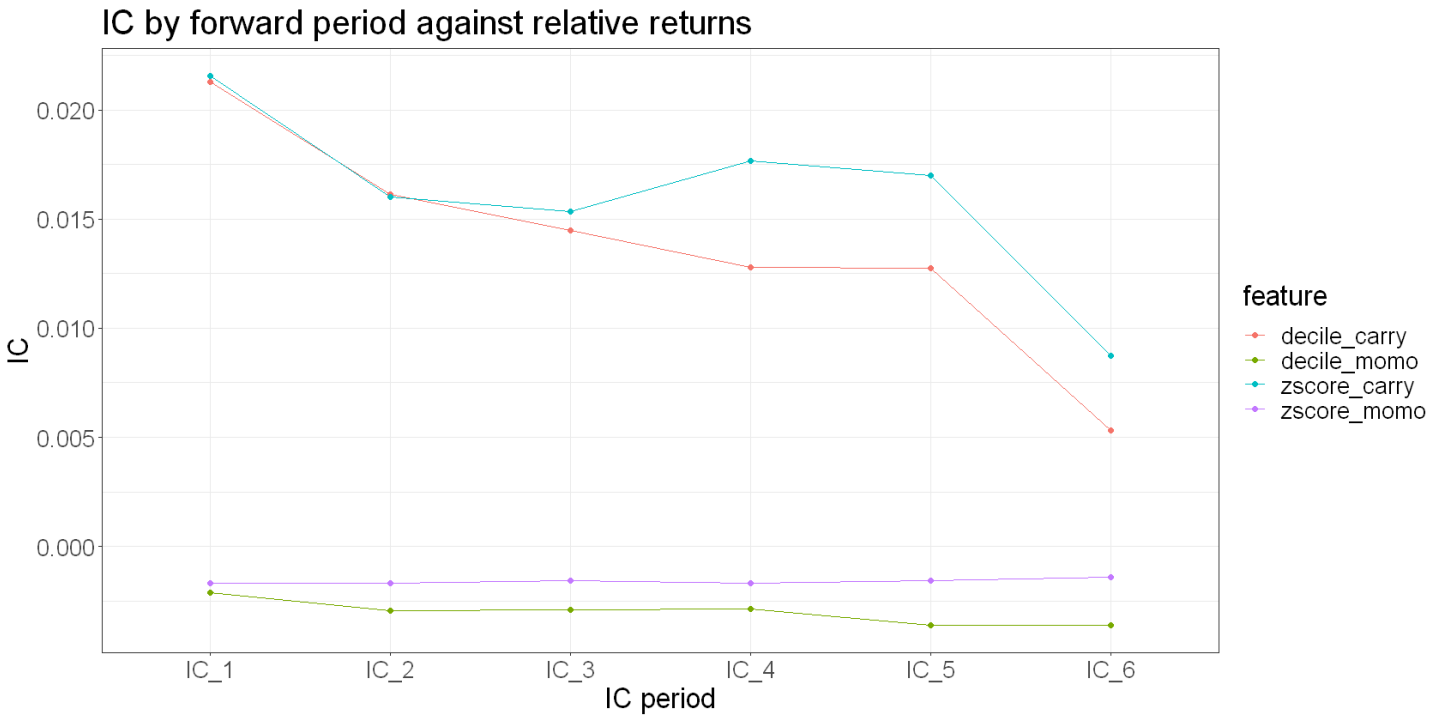


Decay characteristics

Next, it's useful to understand how quickly your signals decay. Signals that decay quickly need to be acted upon without delay, while those that decay slower are more forgiving to execute.

We'll look at relative return IC for our cross-sectional features and the time-series return IC for our breakout feature.

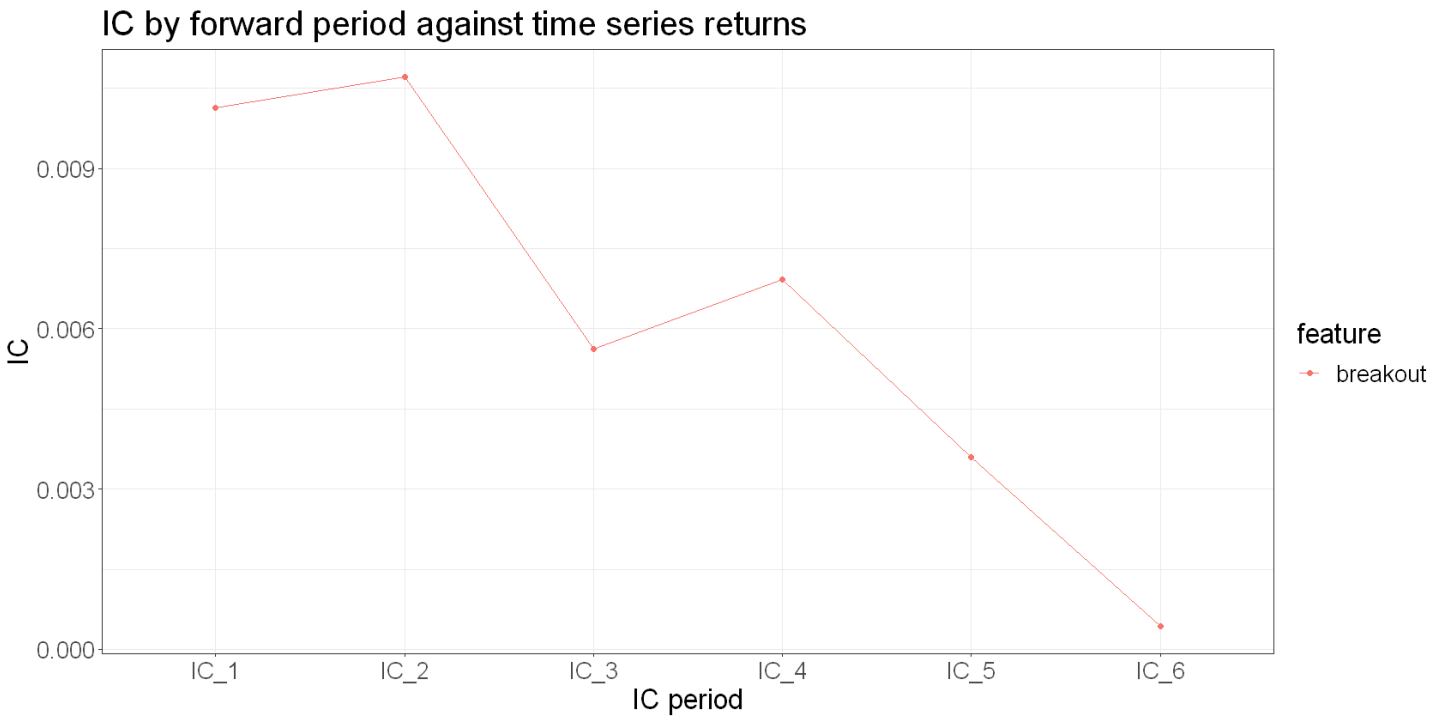
```
features_scaled %>%
  filter(is_universe) %>%
  group_by(ticker) %>%
  arrange(date) %>%
  mutate(
    demeaned_fwd_returns_2 = lead(demeaned_fwd_returns, 1),
    demeaned_fwd_returns_3 = lead(demeaned_fwd_returns, 2),
    demeaned_fwd_returns_4 = lead(demeaned_fwd_returns, 3),
    demeaned_fwd_returns_5 = lead(demeaned_fwd_returns, 4),
    demeaned_fwd_returns_6 = lead(demeaned_fwd_returns, 5),
  ) %>%
  na.omit() %>%
  ungroup() %>%
  pivot_longer(c(zscore_carry, zscore_momo, decile_carry, decile_momo), names_to = "feature
  group_by(feature) %>%
  summarize(
    IC_1 = cor(value, demeaned_fwd_returns),
    IC_2 = cor(value, demeaned_fwd_returns_2),
    IC_3 = cor(value, demeaned_fwd_returns_3),
    IC_4 = cor(value, demeaned_fwd_returns_4),
    IC_5 = cor(value, demeaned_fwd_returns_5),
    IC_6 = cor(value, demeaned_fwd_returns_6),
  )  %>%
  pivot_longer(-feature, names_to = "IC_period", values_to = "IC") %>%
  ggplot(aes(x = factor(IC_period), y = IC, colour = feature, group = feature)) +
  geom_line() +
  geom_point() +
  labs(
    title = "IC by forward period against relative returns",
    x = "IC period"
  )
```



We see some potentially interesting things:

- Our momentum decile feature shows only a slightly negative IC on day 1, but it decays quite slowly.
- Our carry decile feature on the other hand has a significant and sticky IC.

```r
features_scaled %>%
  group_by(ticker) %>%
  mutate(
    fwd_returns_2 = lead(total_fwd_return_simple, 1),
    fwd_returns_3 = lead(total_fwd_return_simple, 2),
    fwd_returns_4 = lead(total_fwd_return_simple, 3),
    fwd_returns_5 = lead(total_fwd_return_simple, 4),
    fwd_returns_6 = lead(total_fwd_return_simple, 5),
  ) %>%
  na.omit() %>%
  ungroup() %>%
  pivot_longer(c(breakout), names_to = "feature") %>%
  group_by(feature) %>%
  summarize(
    IC_1 = cor(value, total_fwd_return_simple),
    IC_2 = cor(value, fwd_returns_2),
    IC_3 = cor(value, fwd_returns_3),
    IC_4 = cor(value, fwd_returns_4),
    IC_5 = cor(value, fwd_returns_5),
    IC_6 = cor(value, fwd_returns_6),
  )  %>%
  pivot_longer(-feature, names_to = "IC_period", values_to = "IC") %>%
  ggplot(aes(x = factor(IC_period), y = IC, colour = feature, group = feature)) +
  geom_line() +
  geom_point() +
  labs(
    title = "IC by forward period against time series returns",
    x = "IC period"
  )
```

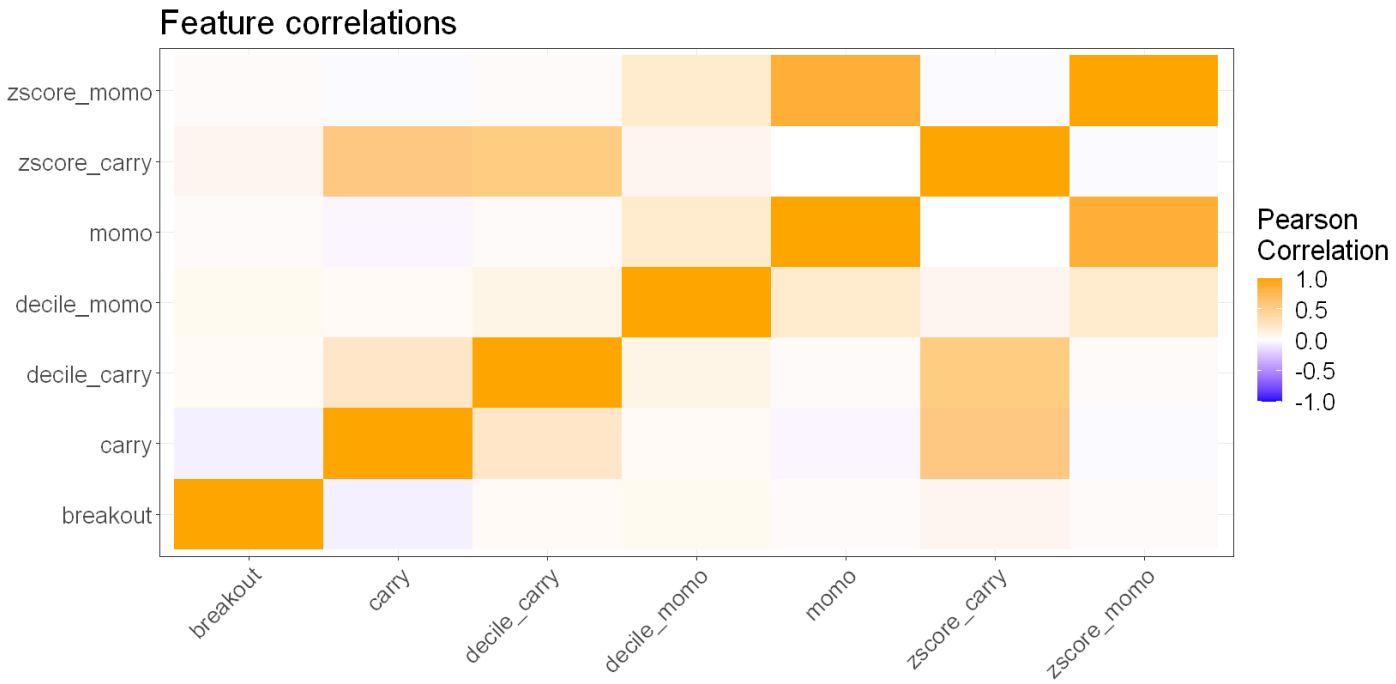


The breakout feature decays quite slowly too. It’s has a larger IC value than momentum, but less than carry.

Feature correlation

Next let's look at the correlation between our features. Ideally, they'll be uncorrelated or anti-correlated to provide a diversification effect.

```
features_scaled %>%
  select('breakout', 'momo', 'carry', 'zscore_carry', 'decile_carry', 'zscore_momo', 'decil
  as.matrix() %>%
  cor() %>%
  as.data.frame() %>%
  rownames_to_column("feature1") %>%
  pivot_longer(-feature1, names_to = "feature2", values_to = "corr") %>%
  ggplot(aes(x = feature1 , y = feature2, fill = corr)) +
  geom_tile() +
  scale_fill_gradient2(
    low = "blue",
    high = "orange",
    mid = "white",
    midpoint = 0,
    limit = c(-1,1),
    name="Pearson\nCorrelation"
  ) +
  labs(
    x = "",
    y = "",
    title = "Feature correlations"
  ) +
  theme(axis.text.x = element_text(angle = 45, vjust = 1, size = 16, hjust = 1))
```



This isn't a bad result. We see that our breakout and carry features are negatively correlated, and our momentum feature is almost uncorrelated with everything. That means that we should get some nice diversification from combining these signals.

Combining our signals

Based on what we've seen and what we understand about our signals, I think it makes sense to:

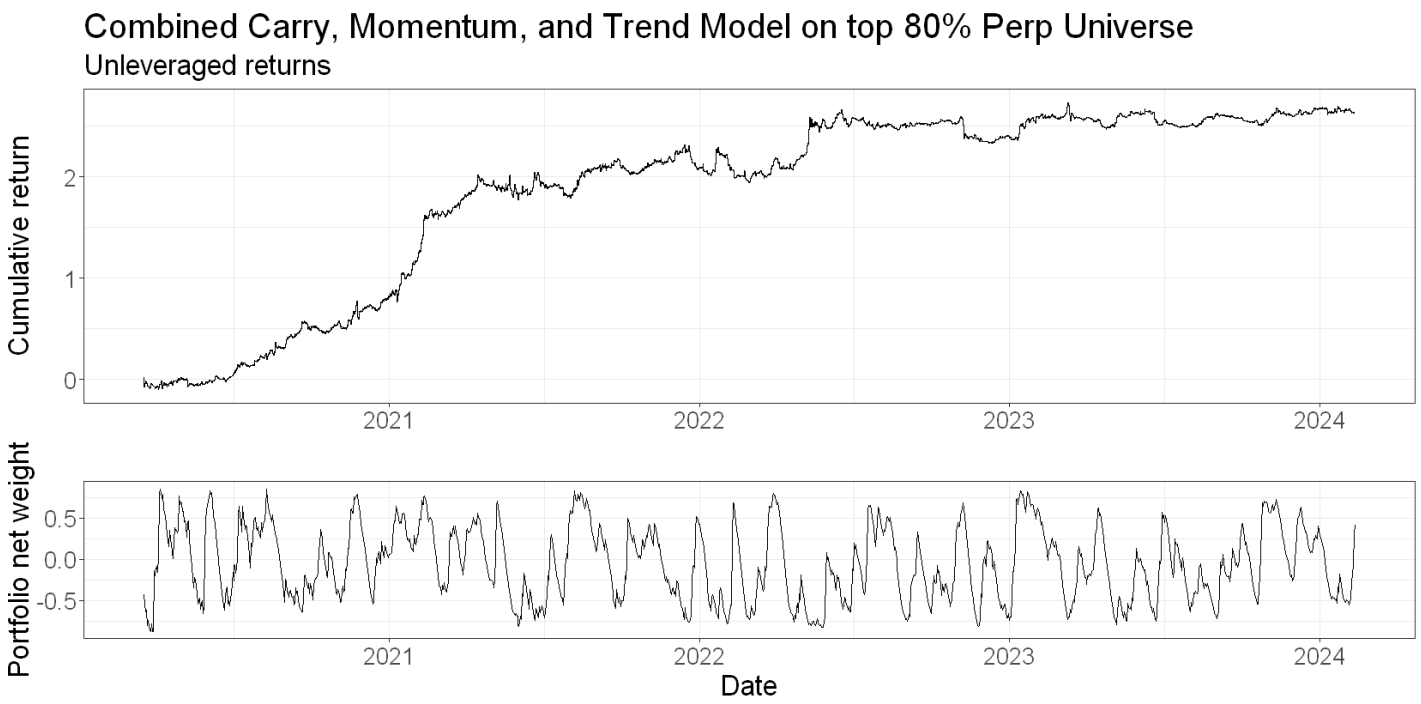
- Overweight the `decile_carry` signal, as it has a high IC, decays slowly, and is anti-correlated with our other cross-sectional feature.
- Underweight the `decile_momo` signal, as it has a small negative IC.
- Overweight the `breakout` signal slightly. It has a moderate IC, decays slowly, and is uncorrelated with the other signals. Note that it also tilts the portfolio net long or short since it is used in the time series, not the cross-section.

You might want to weight these signals differently depending on your objectives and constraints, as well as what you understood from a more thorough analysis of each signal.

In any event, here's a plot of the cumulative returns to our weighted signals over time.

I want to stress that this isn't a backtest - it makes zero attempt to address the real-world issues such as costs, turnover, or universe restriction. It simply plots the returns to our combined features as if you could do so frictionlessly.

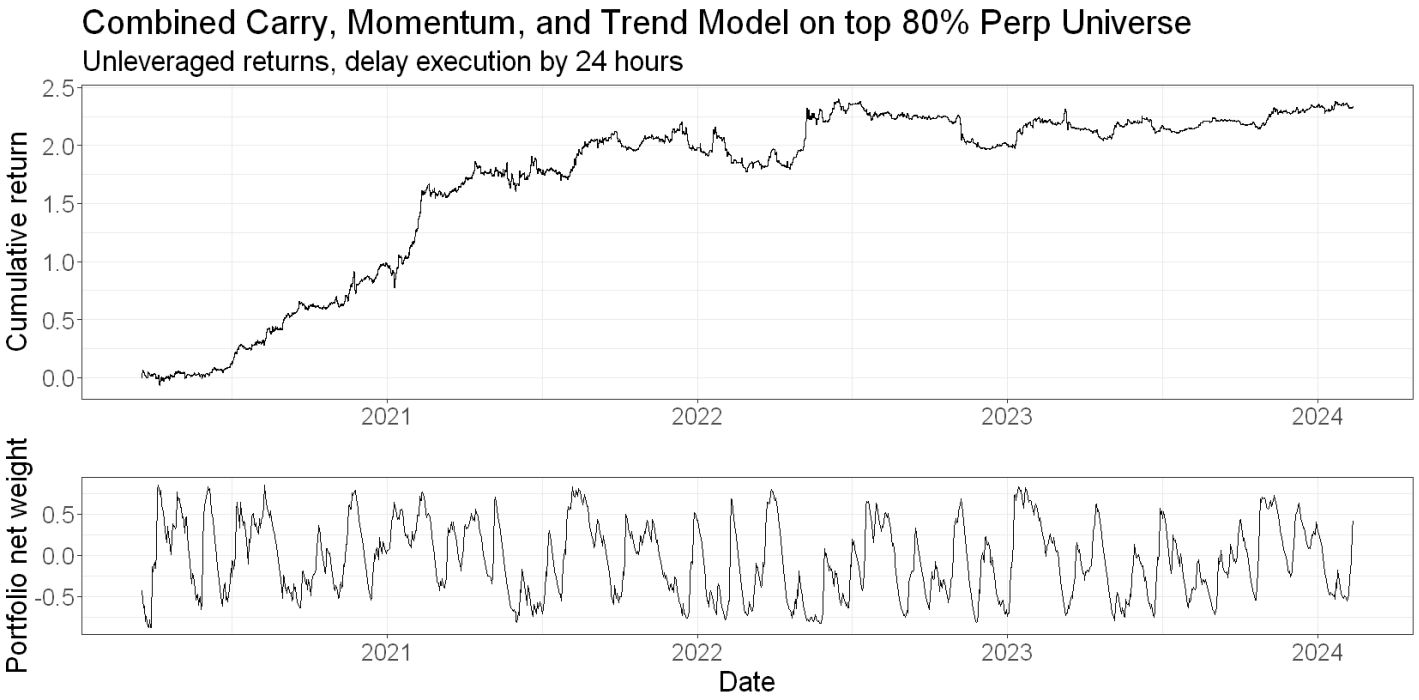
```r
# start simulation from date we first have n tickers in the universe
min_trading_universe_size <- 10
start_date <- features %>%
  group_by(date, is_universe) %>%
  summarize(count = n(), .groups = "drop") %>%
  filter(count >= min_trading_universe_size) %>%
  head(1) %>%
  pull(date)

model_df <- features %>%
  filter(is_universe) %>%
  filter(date >= start_date) %>%
  group_by(date) %>%
  mutate(
    carry_decile = ntile(carry, 10),
    carry_weight = (carry_decile - 5.5),  # will run -4.5 to 4.5
    momo_decile = ntile(momo, 10),
    momo_weight = -(momo_decile - 5.5),  # will run -4.5 to 4.5
    breakout_weight = breakout / 2,   # TODO: probably don't actually want to allow a short
    combined_weight = (0.5*carry_weight + 0.2*momo_weight + 0.3*breakout_weight),
    # scale weights so that abs values sum to 1 - no leverage condition
    scaled_weight = if_else(combined_weight == 0, 0, combined_weight/sum(abs(combined_weigh
  )

returns_plot <- model_df %>%
  summarize(returns = scaled_weight * total_fwd_return_simple, .groups = "drop") %>%
  mutate(logreturns = log(returns+1)) %>%
  ggplot(aes(x=date, y=cumsum(logreturns))) +
  geom_line() +
  labs(
    title = 'Combined Carry, Momentum, and Trend Model on top 80% Perp Universe',
    subtitle = "Unleveraged returns",
    x = "",
    y = "Cumulative return"
  )

weights_plot <- model_df %>%
  group_by(date) %>%
  summarise(total_weight = sum(scaled_weight)) %>%
  ggplot(aes(x = date, y = total_weight)) +
  geom_line() +
  labs(x = "Date", y = "Portfolio net weight")

returns_plot / weights_plot + plot_layout(heights = c(2,1))
```



The portfolio net weight plot shows that the breakout feature tilts the portfolio net long or short.

Note that because of the way we've aligned features and returns, the plot assumes you calculate the feature values at the daily close and act on them at that same price. While this is flattering, it's not entirely inaccurate for a 24-7 market like crypto. For stocks, you'd *definitely* want to put another day's gap between your feature and your forward return. This is illustrative for crypto too. Here's what the returns would look like if we delayed acting on them for 24 hours:

```
returns_plot <- model_df %>%
  summarize(returns = scaled_weight * total_fwd_return_simple_2, .groups = "drop") %>%
  mutate(logreturns = log(returns+1)) %>%
  ggplot(aes(x=date, y=cumsum(logreturns))) +
  geom_line() +
  labs(
    title = 'Combined Carry, Momentum, and Trend Model on top 80% Perp Universe',
    subtitle = "Unleveraged returns, delay execution by 24 hours",
    x = "",
    y = "Cumulative return"
  )

weights_plot <- model_df %>%
  group_by(date) %>%
  summarise(total_weight = sum(scaled_weight)) %>%
  ggplot(aes(x = date, y = total_weight)) +
  geom_line() +
  labs(x = "Date", y = "Portfolio net weight")

returns_plot / weights_plot + plot_layout(heights = c(2,1))
```


Combined Carry, Momentum, and Trend Model on top 80% Perp Universe
Unleveraged returns, delay execution by 24 hours
Cumulative return
2.5
2.0
1.5
1.0
0.5
0.0
2021
2022
2023
2024
Portfolio net weight
0.5
0.0
-0.5
2021
2022
2023
2024
Date

Conclusions

In this article, we looked at ways to understand our signals, and to combine them using a weighting scheme derived from our understanding.

And let's be honest - the work we did here to understand our signals was very shallow. There's much more you can and should do in order to understand your signals. For example:

- Scatterplots and time-series plots that don't hide the variance
- Stability of factor ICs over time
- Stability of the signal itself and the implications for turnover
- Impact of taking the tails only
- etc

In the next article, we'll tackle the questions of turnover and cost and use a heuristic approach to help us navigate these trade-offs. We'll also be a little more careful with universe selection (we don't want to be trading the top 80% of perpetual contracts).

In a future article, we'll also explore how we can use mean-variance optimisation to achieve the same goal.

Navigating Cost Tradeoffs using Heuristics

Every time we trade, we incur a cost. We pay a commission to the exchange or broker, we cross spreads, and we might even have market impact to contend with.

A common issue in quant trading is to find an edge, only to discover that if you executed it naively, you'd get killed with costs.

In this article, I'll show you an example of using a simple heuristic that helps you do an optimal amount of trading to realise your edge in the face of costs.

A heuristic is a simple rule or shortcut to help make decisions. And this one doesn't involve any fancy math or portfolio optimisation and is quite effective.

Previously, we combined three crypto features that resulted in some nice theoretical returns using an untradeable model (it was based on a universe of 200+ perpetual futures contracts and didn't consider costs - its purpose was to simply explore and understand the features, not simulate trading them in the real world).

This time we'll consider costs and constrain the universe so that it would be operationally feasible to trade. And we'll use an accurate backtest to explore the various trade offs.

Let's get to it.

First, we load the relevant libraries and set our session options.

You'll notice I install and load an R package called `rsims`.

This is a backtesting engine for R that I developed. You can find it on GitHub here.

A lot of people reading this will ask "Why do we need another backtester?" Which is a fair reaction. I wrote this one because there was nothing that really did what I needed:

- I wanted something that ran super fast to allow me to quickly explore real-world trade-offs in execution (it does a 10-year simulation on a universe of 250 assets in 650 milliseconds on my machine).
- I wanted something that would take the outputs of a typical quant research process in R and use them as inputs to a backtest with minimal intervening steps.

I'll show you how to use it.

```
# session  options
options(repr.plot.width = 14, repr.plot.height=7, warn = -1)

pacman::p_load_current_gh("Robot-Wealth/rsims", dependencies = TRUE)  # this will take some
library(rsims)
library(tidyverse)
library(tibbletime)
library(roll)
library(patchwork)

# chart options
theme_set(theme_bw())
theme_update(text = element_text(size = 20))
```

Next, read our perpetual futures data. This is the same data used previously in this series, but this time we are a bit more careful with our universe:

- Remove any stablecoins
- Trade only the top 30 coins by trailing 30-day volume

```
perps <- read_csv("https://github.com/Robot-Wealth/r-quant-recipes/raw/master/quantifying-c
head(perps)
```

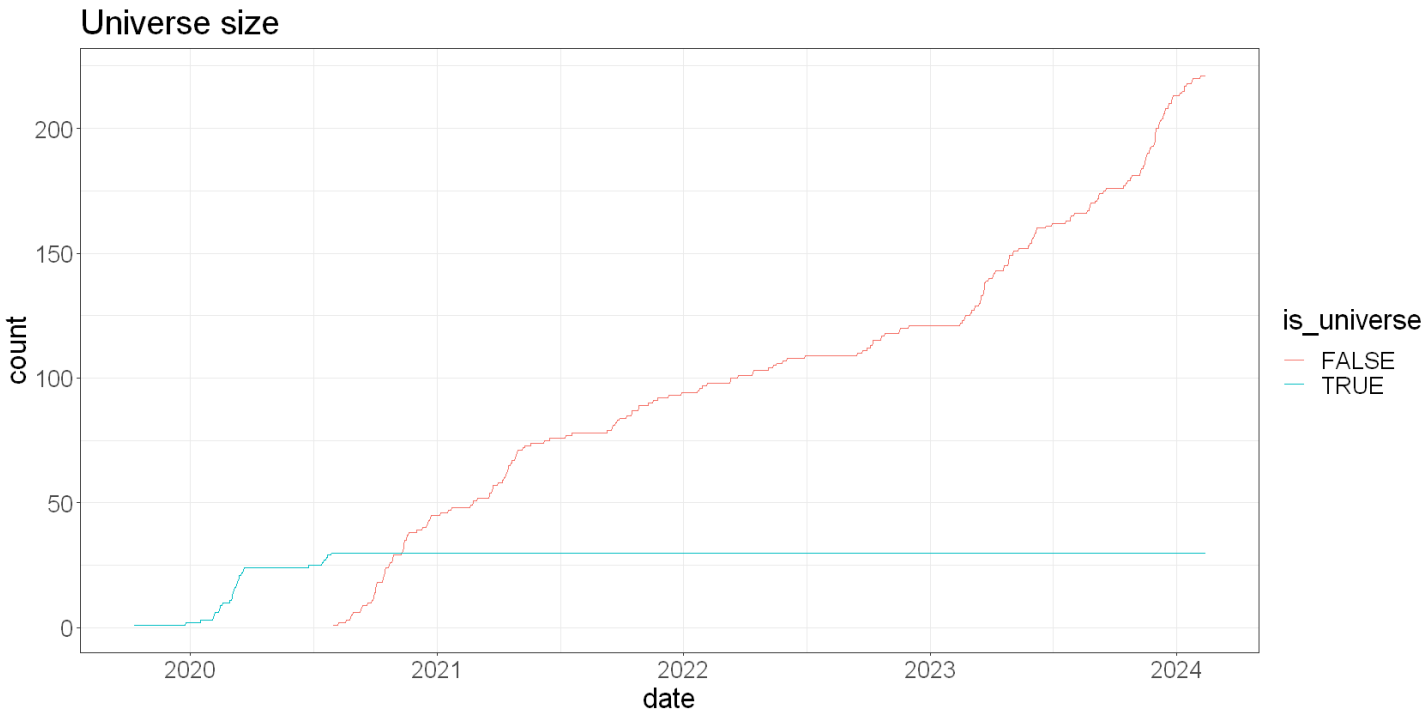
```
Rows: 187251 Columns: 11
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr  (1): ticker
dbl  (9): open, high, low, close, dollar_volume, num_trades, taker_buy_volum...
date (1): date

ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

A tibble: 6 × 11

ticker	date	open	high	low	close	dollar_volume	num_trades	ta
<chr>	<date>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	
BTCUSDT	2019-09-11	10172.13	10293.11	9884.31	9991.84	85955369	10928	
BTCUSDT	2019-09-12	9992.18	10365.15	9934.11	10326.58	157223498	19384	
BTCUSDT	2019-09-13	10327.25	10450.13	10239.42	10296.57	189055129	25370	
BTCUSDT	2019-09-14	10294.81	10396.40	10153.51	10358.00	206031349	31494	
BTCUSDT	2019-09-15	10355.61	10419.97	10024.81	10306.37	211326874	27512	
BTCUSDT	2019-09-16	10306.79	10353.81	10115.00	10120.07	208211376	29030	

```
# remove stablecoins
# list of stablecoins from defi llama
url <- "https://stablecoins.llama.fi/stablecoins?includePrices=true"
response <- httr::GET(url)

stables <- response %>%
  httr::content(as = "text", encoding = "UTF-8") %>%
  jsonlite::fromJSON(flatten = TRUE) %>%
  pluck("peggedAssets") %>%
  pull(symbol)

# sort(stables)

perps <- perps %>%
  filter(!ticker %in% glue::glue("{stables}USDT"))
```

```
# just get the top 30 by trailing 30-day volume
trading_universe_size <- 30

universe <- perps %>%
  group_by(ticker) %>%
  mutate(trail_volume = roll_mean(dollar_volume, 30)) %>%
  na.omit() %>%
  group_by(date) %>%
  mutate(
    volume_rank = row_number(-trail_volume),
    is_universe = volume_rank <= trading_universe_size
  )

universe %>%
  group_by(date, is_universe) %>%
  summarize(count = n(), .groups = "drop") %>%
  ggplot(aes(x=date, y=count, color = is_universe)) +
  geom_line() +
  labs(
    title = 'Universe size'
)
```



The plot shows that we have 30 assets in our universe from mid 2020 (and of course, those assets will change through time).

Next, calculate our features as before:

```
# calculate features
rolling_days_since_high_20 <- purrr::possibly(
  tibbletime::rollify(
    function(x) {
      idx_of_high <- which.max(x)
      days_since_high <- length(x) - idx_of_high
      days_since_high
    },
    window = 20, na_value = NA),
  otherwise = NA
)

features <- universe %>%
  group_by(ticker) %>%
  arrange(date) %>%
  mutate(
    breakout = lag(9.5 - rolling_days_since_high_20(close)),  # puts this feature on a scal
    momo = lag(close - lag(close, 10)/close),
    carry = lag(funding_rate)
  ) %>%
  ungroup() %>%
  na.omit()

head(features)
```

ticker	date	open	high	low	close	dollar_volume	num_trades	taker_b
<chr>	<date>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	
BTCUSDT	2019-10-30	9352.65	9492.86	8975.72	9102.91	1089606006	267844	
BTCUSDT	2019-10-31	9103.86	9438.64	8933.00	9218.70	851802094	229216	
BTCUSDT	2019-11-01	9218.70	9280.00	9011.00	9084.37	816001159	233091	
BTCUSDT	2019-11-02	9084.37	9375.00	9050.27	9320.00	653539543	204338	
BTCUSDT	2019-11-03	9319.00	9366.69	9105.00	9180.97	609237501	219662	
BTCUSDT	2019-11-04	9180.97	9320.00	9073.00	9292.66	631763431	211571	

Next combine our features into target weights as before, but we also need prices and funding rates for backtesting purposes (previously we just summed total log returns, but this time we're actually going to backtest on prices and funding rates).

```r
# calculate target weights
# filter on is_universe so that we calculate features only for stuff that's in the universe
# (we'd have to do this differently if any of these calcs depended on past data, eg if we w
# then, join on original prices for backtesting

# tickers that were ever in the universe
universe_tickers <- features %>%
  filter(is_universe) %>%
  pull(ticker) %>%
  unique()

# print(length(universe_tickers))

# start simulation from date we first have n tickers in the universe
start_date <- features %>%
  group_by(date, is_universe) %>%
  summarize(count = n(), .groups = "drop") %>%
  filter(count >= trading_universe_size) %>%
  head(1) %>%
  pull(date)

# calculate weights
model_df <- features %>%
  filter(is_universe) %>%
  group_by(date) %>%
  mutate(
    carry_decile = ntile(carry, 10),
    carry_weight = (carry_decile - 5.5),  # will run -4.5 to 4.5
    momo_decile = ntile(momo, 10),
    momo_weight = -(momo_decile - 5.5),  # will run -4.5 to 4.5
    breakout_weight = breakout / 2,
    combined_weight = (0.5*carry_weight + 0.2*momo_weight + 0.3*breakout_weight),
    # scale weights so that abs values sum to 1 - no leverage condition
    scaled_weight = combined_weight/sum(abs(combined_weight))
  )  %>%
  select(date, ticker, scaled_weight) %>%
  # join back onto df of prices for all tickers that were ever in the universe
  # so that we have prices before and after a ticker comes into or out of the universe
  # for backtesting purposes
  right_join(
    features %>%
      filter(ticker %in% universe_tickers) %>%
      select(date, ticker, close, funding_rate),
    by = c("date", "ticker")
  ) %>%
  # give anything with a NA weight (due to the join) a zero
  replace_na(list(scaled_weight = 0)) %>%
  arrange(date, ticker) %>%
  filter(date >= start_date)
```

Introducing rsims

Event based simulators are typically slow because each loop through the trading logic depends on the output of the previous one.

`rsims` is so fast because it takes a quasi-event based approach and relies on the user to correctly date-align prices, target weights and funding rates. Specifically, it assumes that `target_weights` are date-aligned with `prices` such that the price at which you assume you trade into a target weight has the same index value.

That means that you need to lag your features relative to your prices. And for Binance perpetuals, you'll need to do the opposite - lag prices relative to funding rates. You also need to ensure that funding rate represents funding to *long* positions, which may not be the default for many exchanges.

This takes a little thought and some data wrangling upstream, however, this is often a natural output of the research process, so this workflow can be quite efficient.

`rsims` also leverages all the usual tricks to speed things up - using efficient data structures (matrixes instead of dataframes), preallocating memory, and offloading any bottlenecks to C++ rather than R.

A simple way to get the date-aligned prices, weights, and funding matrixes from our dataframes is to use `dplyr::pivot_wider`. If you use your date column as the `id_cols` argument, you're guaranteed to have the other variables correctly date-aligned (with NA for any variable missing on a particular date).

```r
# get weights as a wide matrix
# note that date column will get converted to unix timestamp
backtest_weights <- model_df %>%
  pivot_wider(id_cols = date, names_from = ticker, values_from = c(close, scaled_weight)) %
  select(date, starts_with("scaled_weight")) %>%
  data.matrix()

# NA weights should be zero
backtest_weights[is.na(backtest_weights)] <- 0

head(backtest_weights, c(5, 5))

# get prices as a wide matrix
# note that date column will get converted to unix timestamp
backtest_prices <- model_df %>%
  pivot_wider(id_cols = date, names_from = ticker, values_from = c(close, scaled_weight)) %
  select(date, starts_with("close_")) %>%
  data.matrix()

head(backtest_prices, c(5, 5))

# get funding as a wide matrix
# note that date column will get converted to unix timestamp
backtest_funding <- model_df %>%
  pivot_wider(id_cols = date, names_from = ticker, values_from = c(close, funding_rate)) %>
  select(date, starts_with("funding_rate_")) %>%
  data.matrix()

head(backtest_funding, c(5, 5))
```

A matrix: 5 × 5 of type dbl

date	scaled_weight_ADAUSDT	scaled_weight_ALGOUSDT	scaled_weight_ATOMUSDT	scal
18577	-0.05340376	0	-0.08157277	
18578	-0.04981025	0	-0.03083491	
18579	-0.04385542	0	-0.04385542	
18580	0.01767389	0	-0.04047891	
18581	0.02675386	0	0.00000000	

A matrix: 5 × 5 of type dbl

date	close_ADAUSDT	close_ALGOUSDT	close_ATOMUSDT	close_AVAXUSDT
18577	0.10815	0.2823	5.069	3.5881
18578	0.10467	0.2647	5.091	3.4029
18579	0.10643	0.2759	5.200	3.5428
18580	0.10298	0.2668	5.068	3.4840
18581	0.10298	0.2712	5.088	3.6066

A matrix: 5 × 5 of type dbl

date	funding_rate_ADAUSDT	funding_rate_ALGOUSDT	funding_rate_ATOMUSDT	fundin
18577	-0.00125424	-0.00030000	-0.00030000	
18578	-0.00065454	-0.00038655	-0.00030000	
18579	-0.00061317	-0.00045249	-0.00030000	
18580	-0.00030000	-0.00092204	-0.00039684	
18581	-0.00132094	-0.00133458	0.00000783	

Next, prior to running some simulations, here are some helper objects and functions for modelling costs on Binance and plotting results:

```r
# fees - reasonable approximation of actual binance costs (spread + market impact + commiss
fees <- tribble(
  ~tier, ~fee,
  0, 0.,  # use for cost-free simulations
  1, 0.0015,
  2, 0.001,
  3, 0.0008,
  4, 0.0007,
  5, 0.0006,
  6, 0.0004,
  7, 0.0002
)

# make a nice plot with some summary statistics
# plot equity curve from output of simulation
plot_results <- function(backtest_results, weighting_protocol = "0.5/0.2/0.3 Carry/Momo/Bre
  margin <- backtest_results %>%
    group_by(Date) %>%
    summarise(Margin = sum(Margin, na.rm = TRUE))

  cash_balance <- backtest_results %>%
    filter(ticker == "Cash") %>%
    select(Date, Value) %>%
    rename("Cash" = Value)

  equity <- cash_balance %>%
    left_join(margin, by = "Date") %>%
    mutate(Equity = Cash + Margin)

  fin_eq <- equity %>%
    tail(1) %>%
    pull(Equity)

  init_eq <- equity %>%
    head(1) %>%
    pull(Equity)

  total_return <- (fin_eq/init_eq - 1) * 100
  days <- nrow(equity)
  ann_return <- total_return * 365/days
  sharpe <- equity %>%
    mutate(returns = Equity/lag(Equity)- 1) %>%
    na.omit() %>%
    summarise(sharpe = sqrt(365)*mean(returns)/sd(returns)) %>%
    pull()

  equity %>%
    ggplot(aes(x = Date, y = Equity)) +
      geom_line() +
      labs(
        title = "Crypto Stat Arb Simulation",
        subtitle = glue::glue(
          "{weighting_protocol}, costs {commission_pct*100}% of trade value, trade buffer =
```

```
          {round(total_return, 1)}% total return, {round(ann_return, 1)}% annualised, Sharp
        )
      )
}

# calculate sharpe ratio from output of simulation
calc_sharpe <- function(backtest_results) {
  margin <- backtest_results %>%
    group_by(Date) %>%
    summarise(Margin = sum(Margin, na.rm = TRUE))

  cash_balance <- backtest_results %>%
    filter(ticker == "Cash") %>%
    select(Date, Value) %>%
    rename("Cash" = Value)

  equity <- cash_balance %>%
    left_join(margin, by = "Date") %>%
    mutate(Equity = Cash + Margin)

equity %>%
    mutate(returns = Equity/lag(Equity)- 1) %>%
    na.omit() %>%
    summarise(sharpe = sqrt(355)*mean(returns)/sd(returns)) %>%
    pull()
}
```

Introducing our heuristic for reducing trading: the no-trade buffer

Before we get to our simulations, let's discuss our heuristic for reducing our trading.

Finding alphas is one thing, but an extremely important skill in trading is managing them together, in particular managing turnover. The no-trade buffer is a simple heuristic for managing this.

Since we always trade uncertain edges in the face of certain costs, our goal is to do only the minimum amount of trading required to harness the edge.

In the no-trade buffer approach, positions are rebalanced once they deviate from their target by more than a parameter, `trade_buffer`. That is, we have a "no-trade region" around our target position that is twice `trade_buffer`, and we trade only when our positions get more out of whack.

@macrocephalopod on Twitter/X gives a derivation of the trade buffer parameter that allows it to be extended to cases where you want to include co-movement of assets or force more aversion to trading.

Rebalancing happens slightly differently depending on the commission model used:

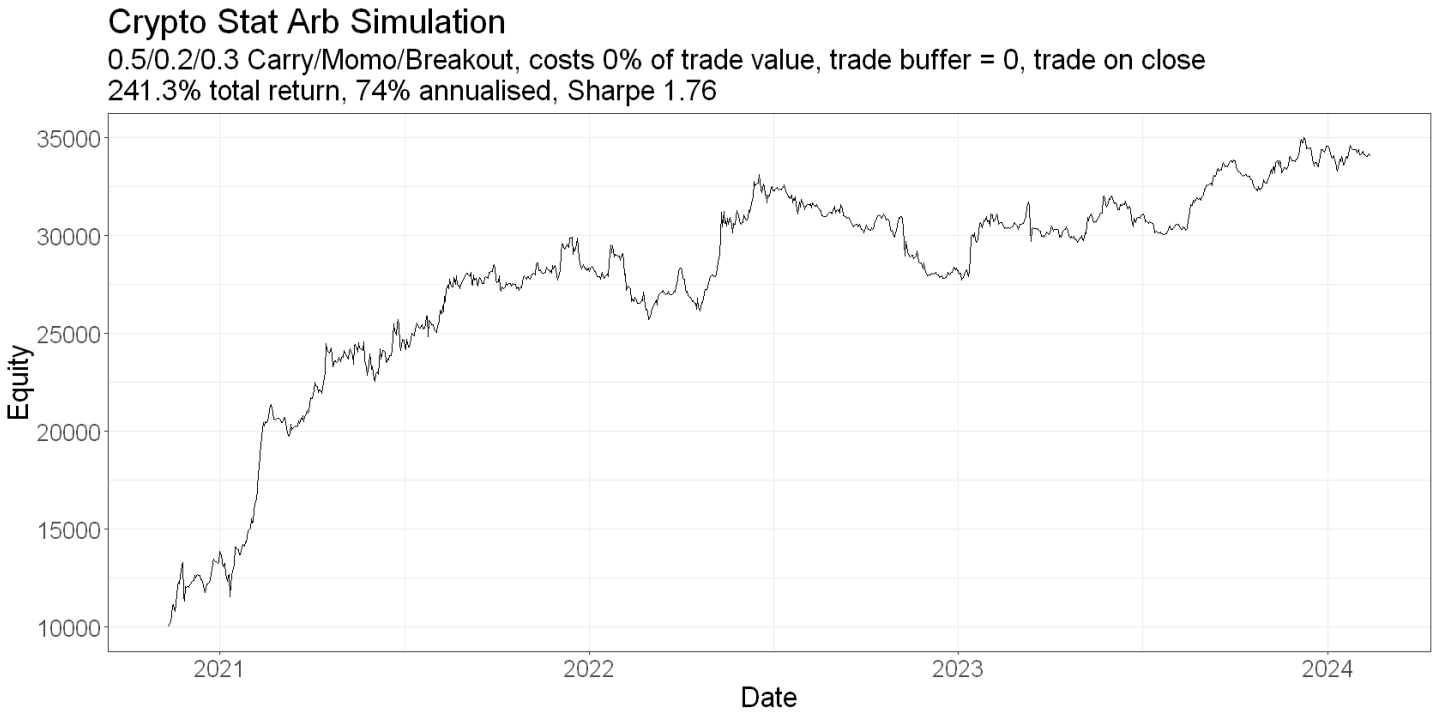
- For minimum commission trading (for instance equities trading with Interactive Brokers), rebalance back to the target weight
- For fixed commission trading (for instance futures and most crypto exchanges), rebalance back to the target weight plus/minus the trade buffer

There are a few different ways to decide on the `trade_buffer` parameter. For instance, you can set it to target a certain portfolio turnover. Or, you can find the value that optimises the Sharpe ratio of a backtest that includes costs. Here we'll take the latter approach.

`rsims` implements the no-trade buffer approach for crypto perpetual futures with the backtesting function `rsims::fixed_commission_backtest_with_funding`.

First, let's simulate the cost-free version with the `trade_buffer` set to zero. This is the same as trading the signal precisely - always rebalancing back to our target positions. We invest a constant $10,000 and don't use leverage.

```r
# cost-free, no trade buffer
# simulation parameters
initial_cash <- 10000
fee_tier <- 0
capitalise_profits <- FALSE  # remain fully invested?
trade_buffer <- 0.
commission_pct <- fees$fee[fees$tier==fee_tier]
margin <- 0.05

# simulation
results_df <- fixed_commission_backtest_with_funding(
  prices = backtest_prices,
  target_weights = backtest_weights,
  funding_rates = backtest_funding,
  trade_buffer = trade_buffer,
  initial_cash = initial_cash,
  margin = margin,
  commission_pct = commission_pct,
  capitalise_profits = capitalise_profits
  ) %>%
  mutate(ticker = str_remove(ticker, "close_")) %>%
  # remove coins we don't trade from results
  drop_na(Value)

plot_results(results_df)
```

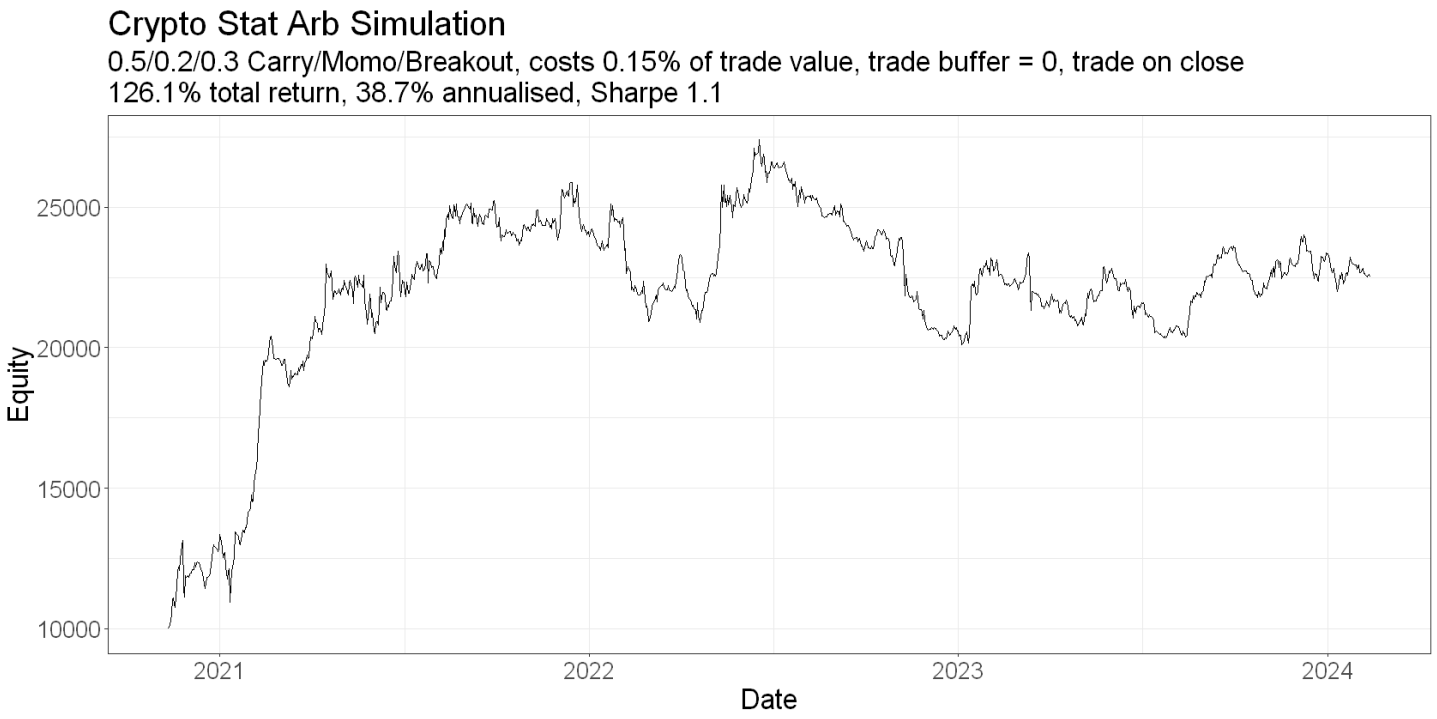


```r
# check that actual weights match intended (can trade fractional contracts, so should be eq
results_df %>%
  left_join(model_df %>% select(ticker, date, scaled_weight), by = c("ticker", "Date" = "da
  group_by(Date) %>%
  mutate(
    actual_weight = Value/(initial_cash)
  )  %>%
  filter(scaled_weight != 0) %>%
  tail(10)
```

ticker	Date	Close	Position	Value	Margin	Funding
<chr>	<date>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>
CFXUSDT	2024-02-13	0.2192000	-1.070045e+03	-234.55378	11.727689	0.06922776
ARBUSDT	2024-02-13	2.0327000	5.347353e+01	108.69565	5.434783	-0.02059611
BLURUSDT	2024-02-13	0.6817000	-2.601519e+02	-177.34554	8.867277	0.09084505
SUIUSDT	2024-02-13	1.7847000	3.814524e+02	680.77803	34.038902	-0.20855505
UMAUSDT	2024-02-13	4.2210000	5.014700e+01	211.67048	10.583524	0.34515420
SEIUSDT	2024-02-13	0.7359000	3.809218e+02	280.32037	14.016018	-0.20655005
TIAUSDT	2024-02-13	19.2523000	3.892667e+01	749.42792	37.471396	0.42387837
ORDIUSDT	2024-02-13	67.5660000	-2.794115e+00	-188.78719	9.439359	0.54223212
PYTHUSDT	2024-02-13	0.5799000	-1.085171e+02	-62.92906	3.146453	-0.03434807
1000SATSUSDT	2024-02-13	0.0004505	-1.180991e+06	-532.03661	26.601831	0.58351629

Next, let's add costs but keep our `trade_buffer` at zero. We'll assume we pay 0.15% of the value of each trade in costs.

```
# explore costs-turnover tradeoffs
# with costs, no trade buffer
fee_tier <- 1.
commission_pct <- fees$fee[fees$tier==fee_tier]

# simulation
results_df <- fixed_commission_backtest_with_funding(
  prices = backtest_prices,
  target_weights = backtest_weights,
  funding_rates = backtest_funding,
  trade_buffer = trade_buffer,
  initial_cash = initial_cash,
  margin = margin,
  commission_pct = commission_pct,
  capitalise_profits = capitalise_profits
  ) %>%
  mutate(ticker = str_remove(ticker, "close_")) %>%
  # remove coins we don't trade from results
  drop_na(Value)

results_df %>%
  plot_results()
```

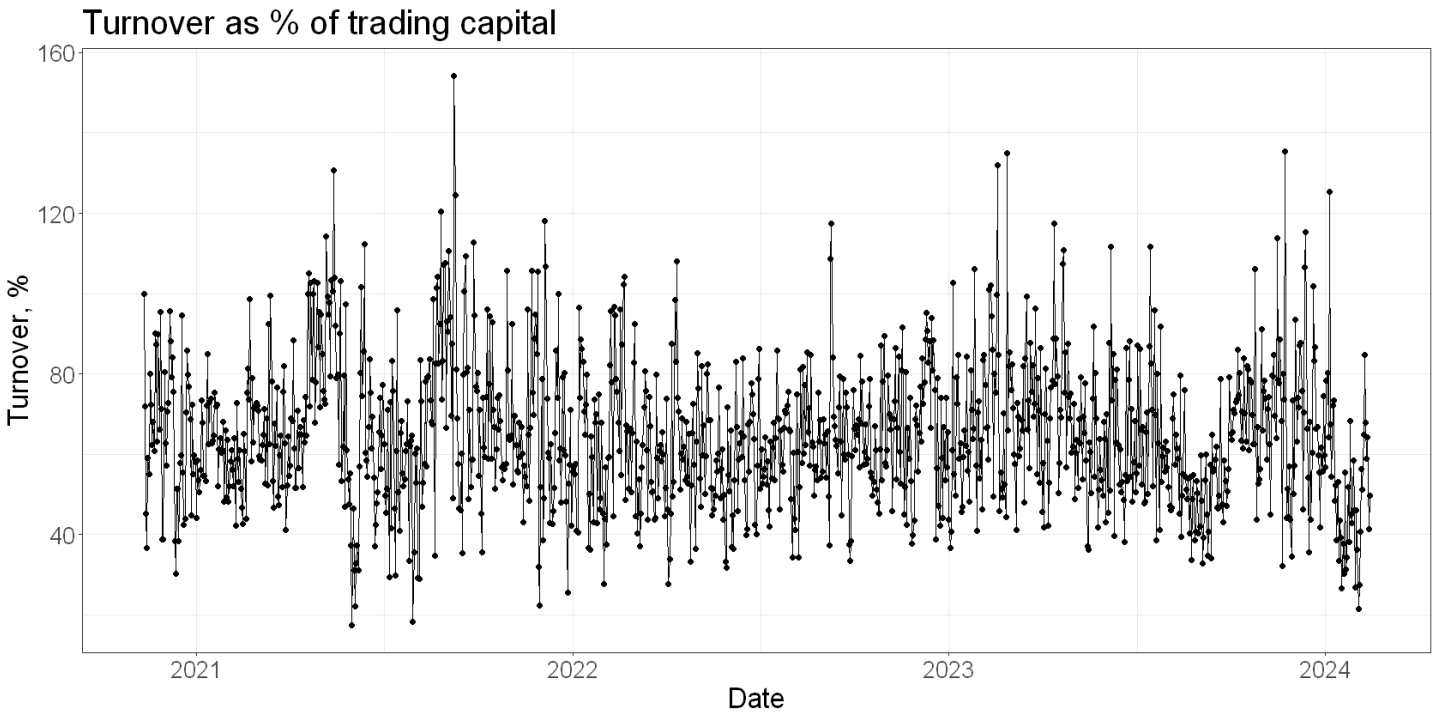


The impact of costs is obvious.

Let's have a look at the turnover of our portfolio.

First, here's the daily turnover as a percentage of our trading capital:

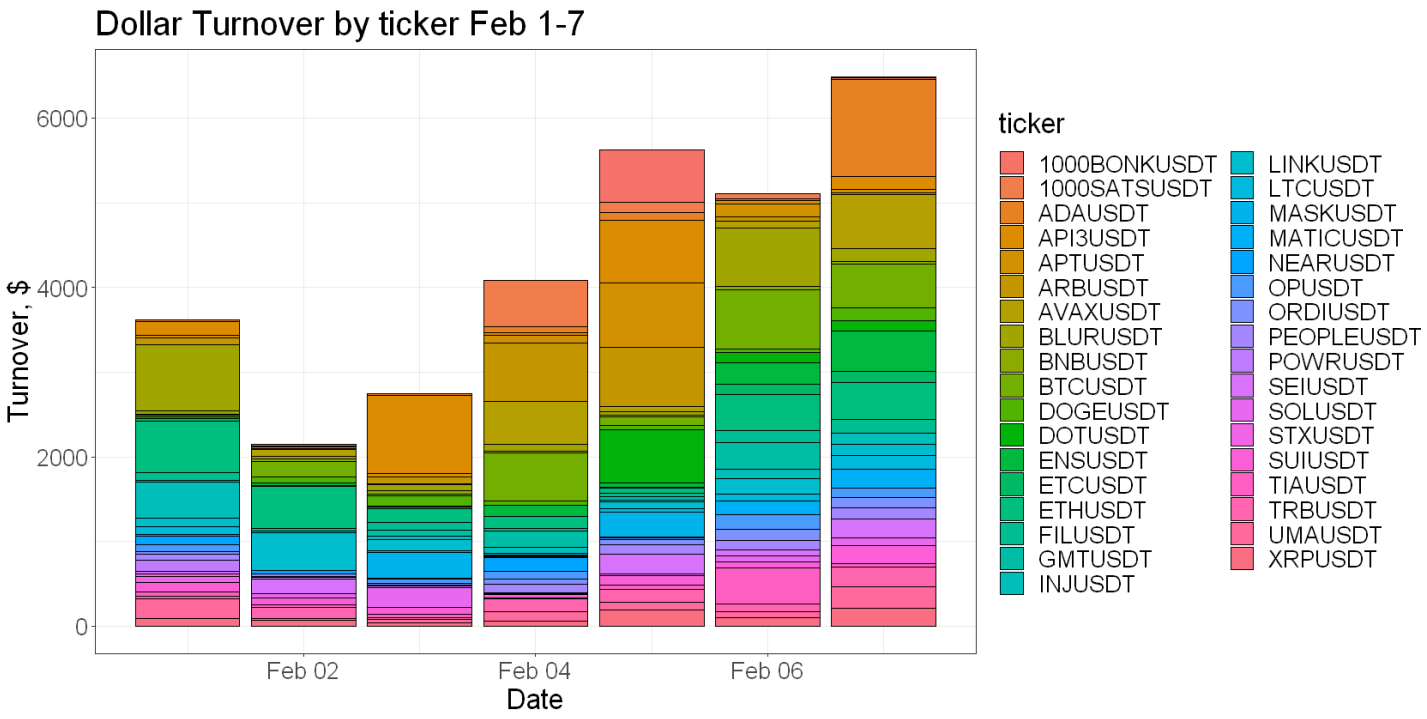
```
results_df %>%
  filter(ticker != "Cash") %>%
  group_by(Date) %>%
  summarise(Turnover = 100*sum(abs(TradeValue))/initial_cash) %>%
  ggplot(aes(x = Date, y = Turnover)) +
    geom_line() +
    geom_point() +
    labs(
      title = "Turnover as % of trading capital",
      y = "Turnover, %"
    )
```



You can see that we regularly turnover than half the portfolio on a single day! That's going to cost an absolute ton.

Here's a plot of dollar turnover by ticker for a few days in February. You can see that we do a ton of small rebalances, which is operationally tedious, expensive, and unlikely to be very valuable.

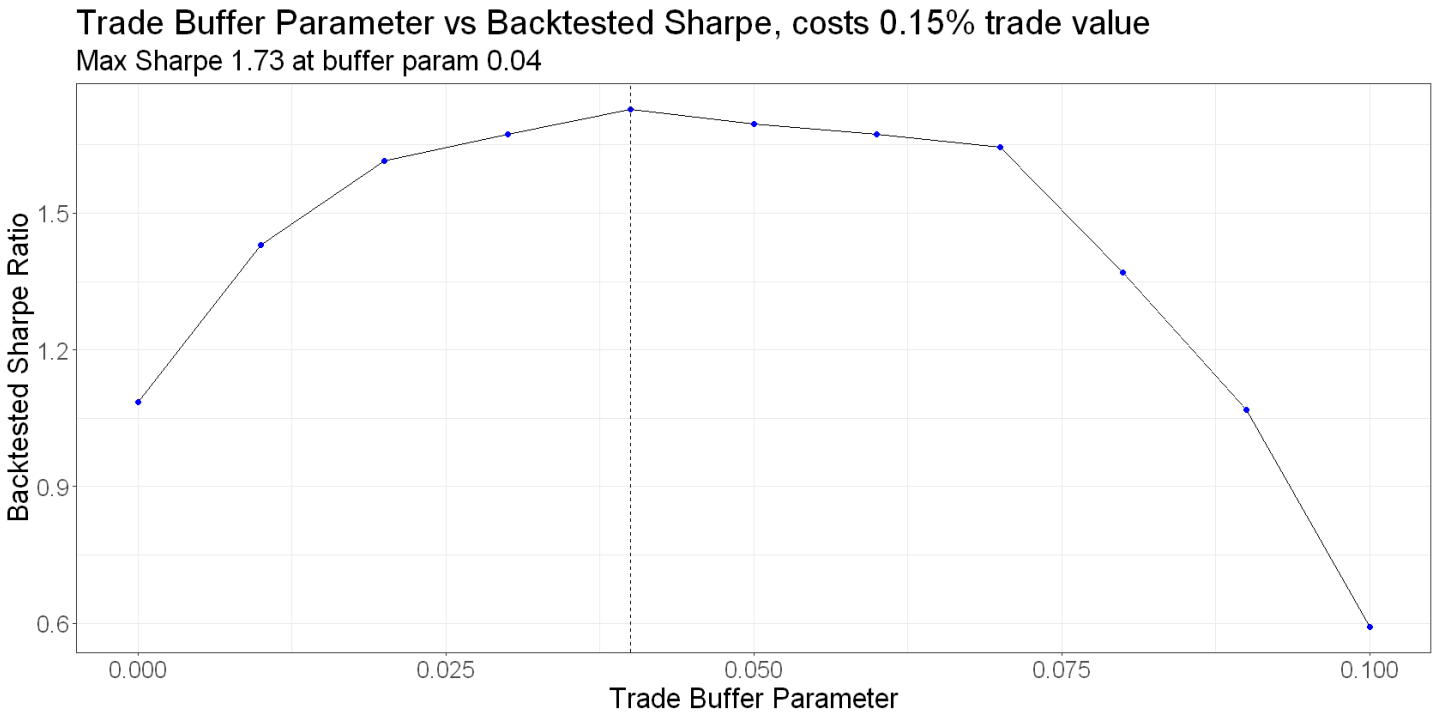
```
# on a particular few days
results_df %>%
  filter(Date >= "2024-02-01", Date <= "2024-02-07") %>%
  filter(abs(TradeValue) > 0) %>%
  ggplot(aes(x = Date, y = abs(TradeValue), fill = ticker)) +
    geom_bar(stat = "identity", position = "stack", colour = "black") +
    labs(
        title = "Dollar Turnover by ticker Feb 1-7",
        y = "Turnover, $"
    )
```



We can reduce this turnover by introducing the no-trade buffer.

To find a sensible value, find the `trade_buffer` that maximised the historical after-cost Sharpe ratio:

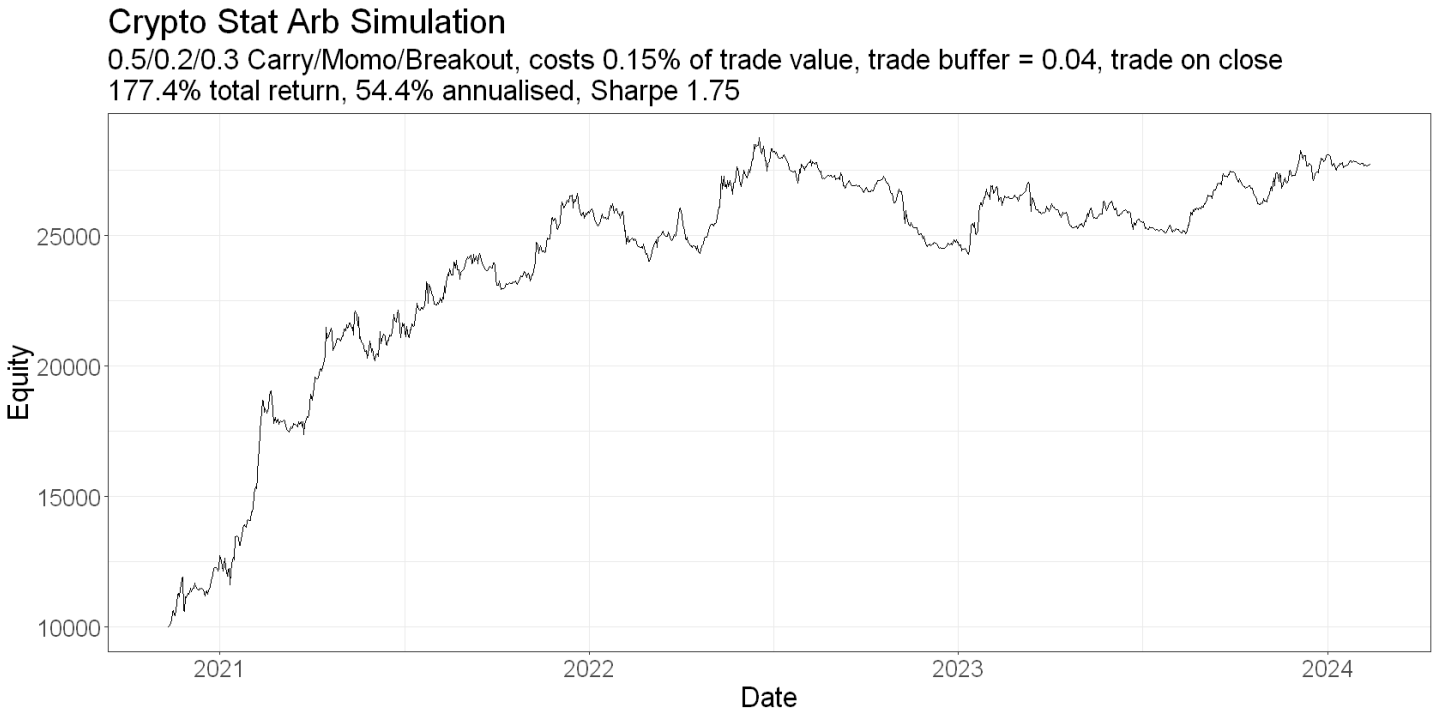
```r
# find appropriate trade buffer by optimising historical sharpe
sharpes <- list()
trade_buffers <- seq(0, 0.1, by = 0.01)
for(trade_buffer in trade_buffers) {
  sharpes <- c(
    sharpes,
    fixed_commission_backtest_with_funding(
      prices = backtest_prices,
      target_weights = backtest_weights,
      funding_rates = backtest_funding,
      trade_buffer = trade_buffer,
      initial_cash = initial_cash,
      margin = margin,
      commission_pct = commission_pct,
      capitalise_profits = capitalise_profits
    ) %>%
      calc_sharpe()
  )
}

sharpes <- unlist(sharpes)
data.frame(
  trade_buffer = trade_buffers,
  sharpe = sharpes
) %>%
  ggplot(aes(x = trade_buffer, y = sharpe)) +
    geom_line() +
    geom_point(colour = "blue") +
    geom_vline(xintercept = trade_buffers[which.max(sharpes)], linetype = "dashed") +
    labs(
      x = "Trade Buffer Parameter",
      y = "Backtested Sharpe Ratio",
      title = glue::glue("Trade Buffer Parameter vs Backtested Sharpe, costs {commission_pc
      subtitle = glue::glue("Max Sharpe {round(max(sharpes), 2)} at buffer param {trade_buf
    )
```



A value of 0.04 maximised our historical after-cost Sharpe. Historical performance was quite stable across a decent range.

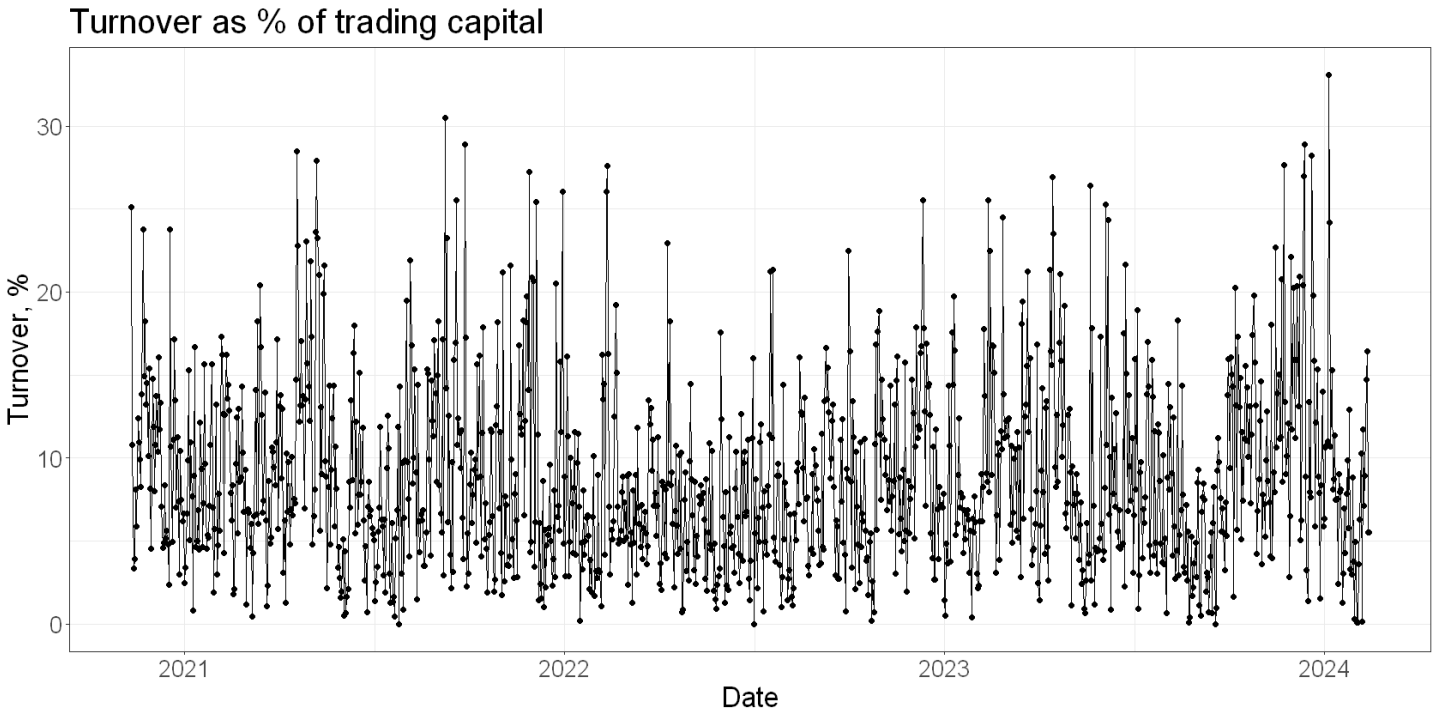
You might pick a value a little higher than 0.04 to mitigate the risk that your out-of-sample performance will be worse than your in-sample (almost always a good assumption). But for now, let's just simulate 0.04:

```
# get back original with costs simulation results
initial_cash <- 10000
capitalise_profits <- FALSE  # remain fully invested?
trade_buffer <- 0.04
fee_tier <- 1.
commission_pct <- fees$fee[fees$tier==fee_tier]

# simulation
results_df <- fixed_commission_backtest_with_funding(
    prices = backtest_prices,
    target_weights = backtest_weights,
    funding_rates = backtest_funding,
    trade_buffer = trade_buffer,
    initial_cash = initial_cash,
    margin = margin,
    commission_pct = commission_pct,
    capitalise_profits = capitalise_profits
  ) %>%
  mutate(ticker = str_remove(ticker, "close_")) %>%
  # remove coins we don't trade from results
  drop_na(Value)

# simulation results
results_df %>%
  plot_results()
```



Let's take a look at the effect on turnover:

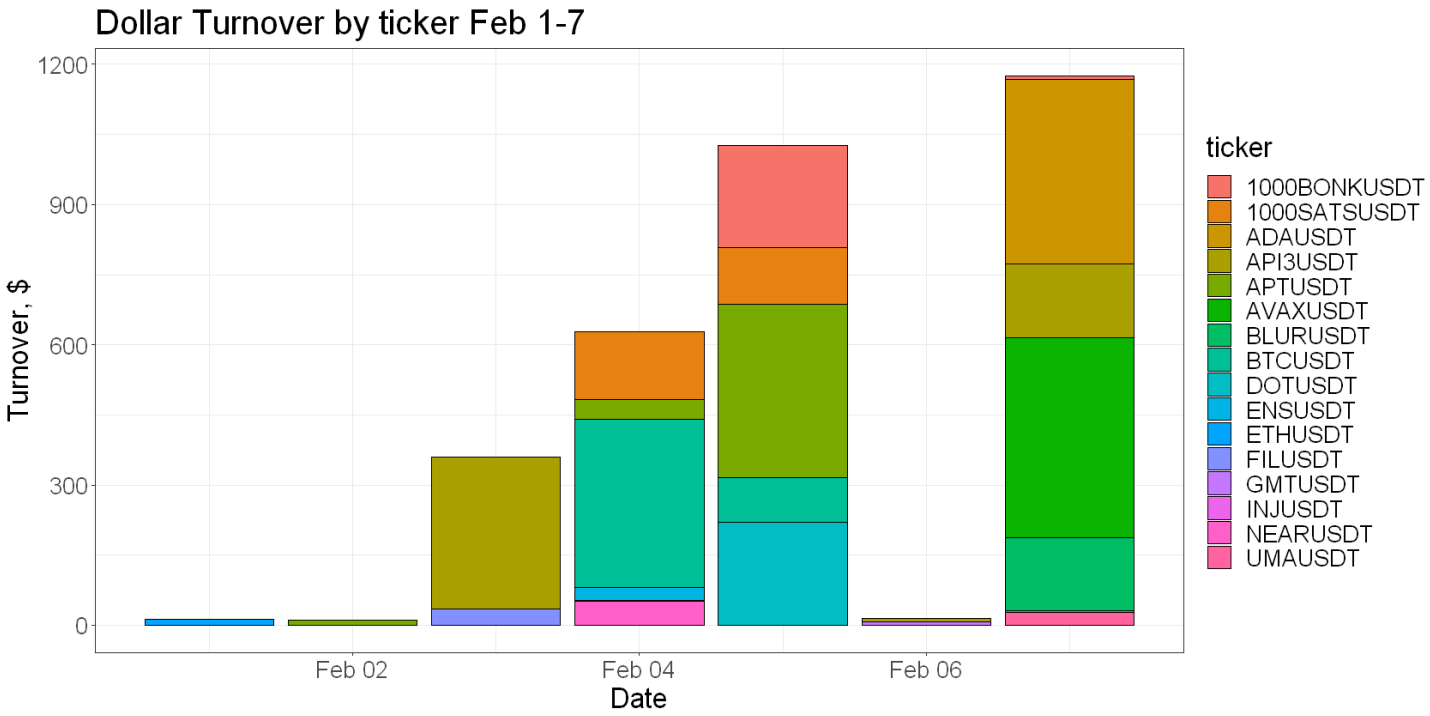
```
results_df %>%
  filter(ticker != "Cash") %>%
  group_by(Date) %>%
  summarise(Turnover = 100*sum(abs(TradeValue))/initial_cash) %>%
  ggplot(aes(x = Date, y = Turnover)) +
    geom_line() +
    geom_point() +
    labs(
      title = "Turnover as % of trading capital",
      y = "Turnover, %"
    )
```



You can see that turnover has come way down.

This is obvious when we plot dollar turnover by ticker for a few days in February. We only do a handful of rebalances each day - this is both cost-effective and operationally efficient.

```
# on a particular few days
results_df %>%
  filter(Date >= "2024-02-01", Date <= "2024-02-07") %>%
  filter(abs(TradeValue) > 0) %>%
  ggplot(aes(x = Date, y = abs(TradeValue), fill = ticker)) +
    geom_bar(stat = "identity", position = "stack", colour = "black") +
    labs(
        title = "Dollar Turnover by ticker Feb 1-7",
        y = "Turnover, $"
    )
```


Dollar Turnover by ticker Feb 1-7
Turnover, $
1200
900
600
300
0
Feb 02
Feb 04
Feb 06
Date
ticker
1000BONKUSDT
1000SATSUSDT
ADAUSDT
API3USDT
APTUSDT
AVAXUSDT
BLURUSDT
BTCUSDT
DOTUSDT
ENSUSDT
ETHUSDT
FILUSDT
GMTUSDT
INJUSDT
NEARUSDT
UMAUSDT

Conclusion

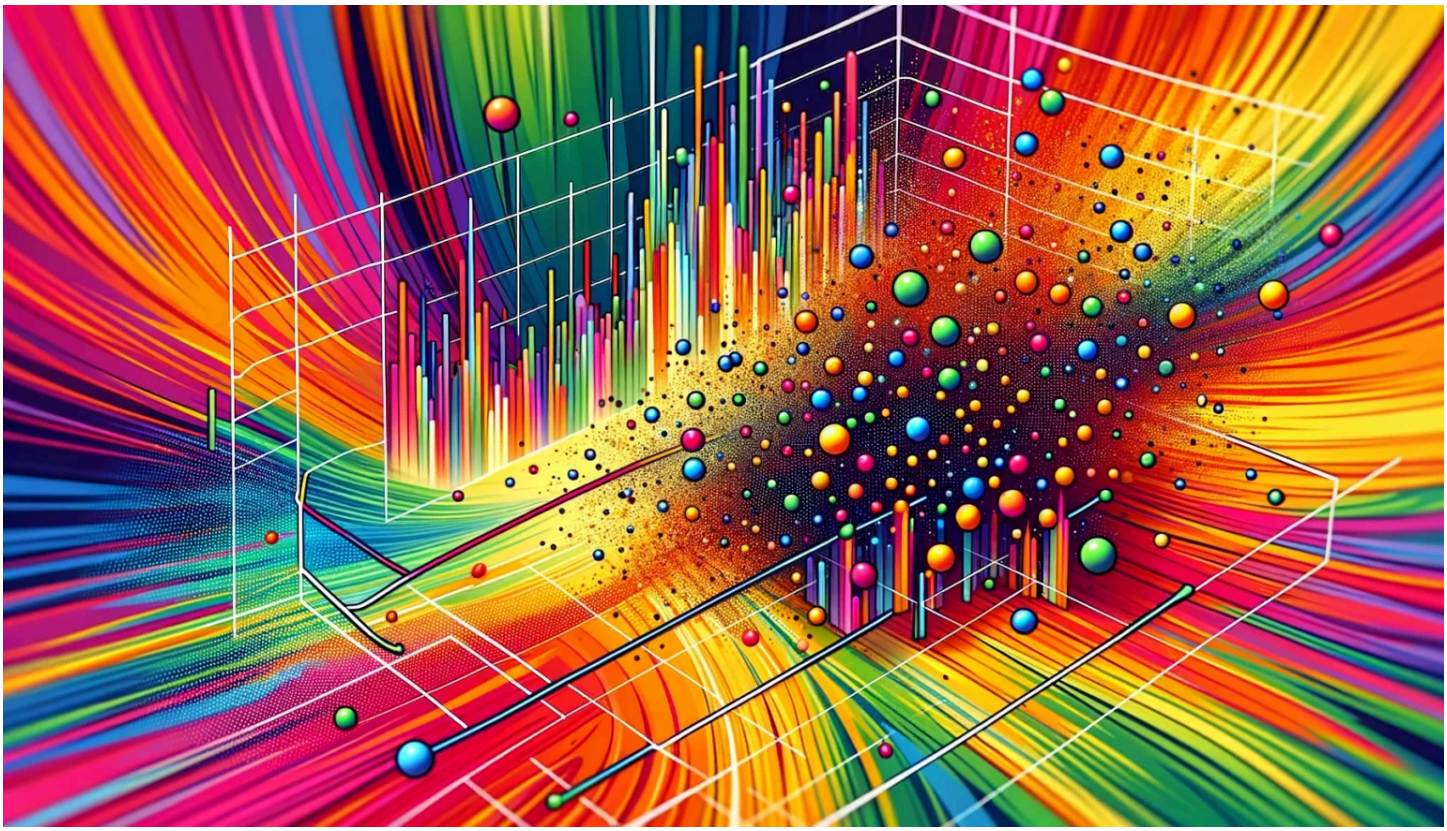
In this chapter, we saw an example of managing portfolio turnover with the heuristic no-trade buffer rule that stops you trading until your positions get out of whack with your target positions by a certain amount.

This is a simple and effective approach that requires no fancy math or optimisation techniques.

We also introduced the `rsims` package for doing fast, accurate backtests in R. The benefit of `rsims` is that it naturally takes the outputs of a quant research process as the inputs for the backtest, leading to an efficient workflow.

In the next several chapters, we'll explore another approach for managing trading decisions using optimisation techniques.

How to Model Features as Expected Returns



Modeling features as expected returns can be a useful way to develop trading strategies, but it requires some care.

The main advantage is that it directly aligns with the objective of predicting and capitalising on future returns. This can make optimisation and implementation more intuitive. It also facilitates direct comparison between features and provides a common framework for incorporating new signals or reassessing existing ones.

Finally, expected return models can be integrated with common risk models such as covariance estimates, and enable a direct comparison with trading costs, opening up the possibility to use mathematical optimisation techniques to manage the trade-off between return, risk, and turnover in the face of real world constraints.

However, a little care is needed in the modeling process. It is all too easy to overfit models to past data, and the risk of mis-specifying a model (choosing a model that doesn't reflect the underlying data) is real.

You can somewhat mitigate these risks by keeping things simple and having a good understanding of your futures that informs your choice of model. I'll share some examples below.

In this article, we'll take the carry, momentum, and breakout features used in the previous articles and model them as expected returns to use in a trading strategy. I'll show you some simple tricks to avoid the most common pitfalls.

First, load libraries and set session options:

```
# session  options
options(repr.plot.width = 14, repr.plot.height=7, warn = -1)

library(tidyverse)
library(tidyfit)
library(tibbletime)
library(roll)
library(patchwork)
pacman::p_load_current_gh("Robot-Wealth/rsims", dependencies = TRUE)

# chart options
theme_set(theme_bw())
theme_update(text = element_text(size = 20))
```

```
── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
✓ dplyr     1.1.4     ✓ readr     2.1.4
✓ forcats   1.0.0     ✓ stringr   1.5.1
✓ ggplot2   3.4.4     ✓ tibble    3.2.1
✓ lubridate 1.9.3     ✓ tidyr     1.3.1
✓ purrr     1.0.2
── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to becc

Attaching package: 'tibbletime'


The following object is masked from 'package:stats':

    filter
```

Next, load the data we've been using in this series. It consists of daily perpetual futures contract prices from Binance. You can get it here.

```
perps <- read_csv("https://github.com/Robot-Wealth/r-quant-recipes/raw/master/quantifying-c
head(perps)
```

```
Rows: 187251 Columns: 11
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr  (1): ticker
dbl  (9): open, high, low, close, dollar_volume, num_trades, taker_buy_volum...
date (1): date

ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

A tibble: 6 × 11

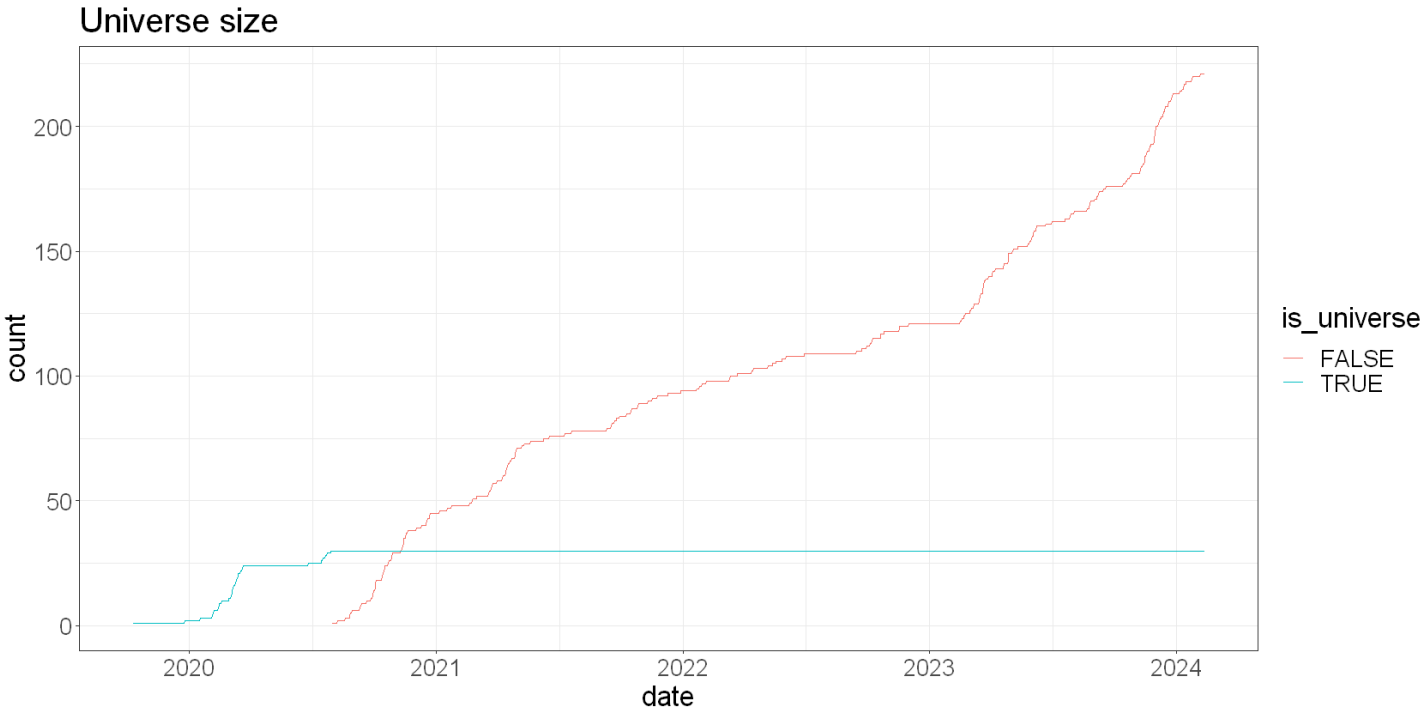
ticker	date	open	high	low	close	dollar_volume	num_trades	ta
<chr>	<date>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	
BTCUSDT	2019-09-11	10172.13	10293.11	9884.31	9991.84	85955369	10928	
BTCUSDT	2019-09-12	9992.18	10365.15	9934.11	10326.58	157223498	19384	
BTCUSDT	2019-09-13	10327.25	10450.13	10239.42	10296.57	189055129	25370	
BTCUSDT	2019-09-14	10294.81	10396.40	10153.51	10358.00	206031349	31494	
BTCUSDT	2019-09-15	10355.61	10419.97	10024.81	10306.37	211326874	27512	
BTCUSDT	2019-09-16	10306.79	10353.81	10115.00	10120.07	208211376	29030	

For the purposes of this example, we'll create the same crypto universe that we used last time - the top 30 Binance perpetual futures contracts by trailing 30-day dollar-volume, with stables and wrapped tokens removed.

We'll also calculate returns at this step for later use.

```r
# get same universe as before - top 30 by rolling 30-day dollar volume, no stables

# remove stablecoins
# list of stablecoins from defi llama
url <- "https://stablecoins.llama.fi/stablecoins?includePrices=true"
response <- httr::GET(url)

stables <- response %>%
  httr::content(as = "text", encoding = "UTF-8") %>%
  jsonlite::fromJSON(flatten = TRUE) %>%
  pluck("peggedAssets") %>%
  pull(symbol)

# sort(stables)

perps <- perps %>%
  filter(!ticker %in% glue::glue("{stables}USDT"))
```

```r
# get the top 30 by trailing 30-day volume
trading_universe_size <- 30

universe <- perps %>%
  group_by(ticker) %>%
  # also calculate returns for later
  mutate(
    total_return_simple = funding_rate + (close - lag(close, 1))/lag(close, 1),
    total_return_log = log(1 + total_return_simple),
    total_fwd_return_simple = dplyr::lead(funding_rate, 1) + (dplyr::lead(close, 1) - close
    total_fwd_return_log = log(1 + total_fwd_return_simple)
  ) %>%
  mutate(trail_volume = roll_mean(dollar_volume, 30)) %>%
  na.omit() %>%
  group_by(date) %>%
  mutate(
    volume_rank = row_number(-trail_volume),
    is_universe = volume_rank <= trading_universe_size,
  )

universe %>%
  group_by(date, is_universe) %>%
  summarize(count = n(), .groups = "drop") %>%
  ggplot(aes(x=date, y=count, color = is_universe)) +
  geom_line() +
  labs(
    title = 'Universe size'
)
```



Next, calculate our features as before. We have:

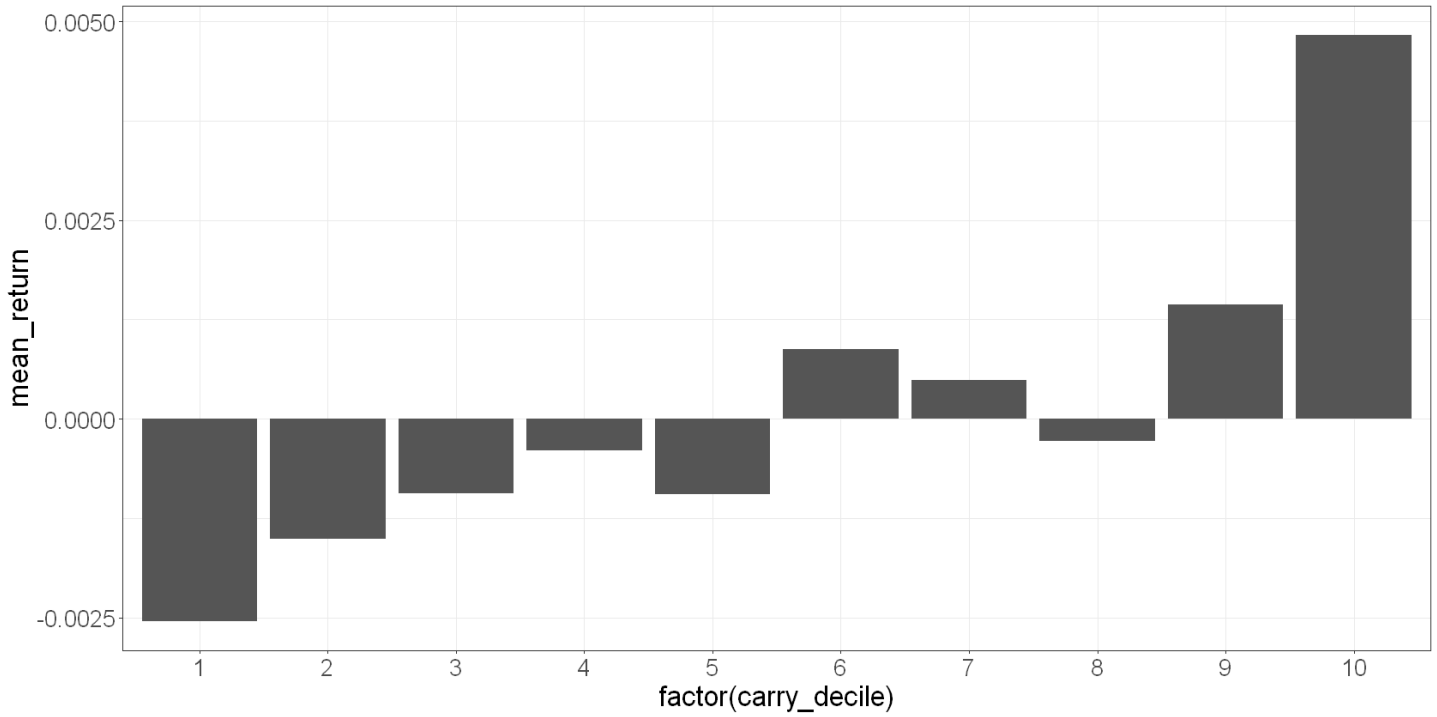
- Short-term (10-day) cross-sectional momentum (buckted into deciles by date)
- Short-term (1-day) cross-sectional carry (also bucketed into deciles by date)
- A breakout feature defined as the number of days since the 20-day high which we use as a time-series return predictor.

```
# calculate features as before
rolling_days_since_high_20 <- purrr::possibly(
  tibbletime::rollify(
    function(x) {
      idx_of_high <- which.max(x)
      days_since_high <- length(x) - idx_of_high
      days_since_high
    },
    window = 20, na_value = NA),
  otherwise = NA
)

features <- universe %>%
  group_by(ticker) %>%
  arrange(date) %>%
  mutate(
    breakout = 9.5 - rolling_days_since_high_20(close),  # puts this feature on a scale -9.
    momo = close - lag(close, 10)/close,
    carry = funding_rate
  ) %>%
  ungroup() %>%
  na.omit()

# create a model df on our universe with momo and carry features scaled into deciles
model_df <- features %>%
  filter(is_universe) %>%
  group_by(date) %>%
  mutate(
    carry_decile = ntile(carry, 10),
    momo_decile = ntile(momo, 10),
    # also calculate demeaned return for everything in our universe each day for later
    demeaned_return = total_return_simple - mean(total_return_simple, na.rm = TRUE),
    demeaned_fwd_return = total_fwd_return_simple - mean(total_fwd_return_simple, na.rm = T
  ) %>%
  ungroup()

  # start simulation from date we first have n tickers in the universe
start_date <- features %>%
  group_by(date, is_universe) %>%
  summarize(count = n(), .groups = "drop") %>%
  filter(count >= trading_universe_size) %>%
  head(1) %>%
  pull(date)
```

Now that we have our features, we can model expected returns.

In the article on *Quantifying and Combining Alphas*, we saw that the carry feature showed a very rough and noisily linear relationship with forward cross-sectional returns. It looks even nicer, although not perfectly linear, on our cut-down universe:

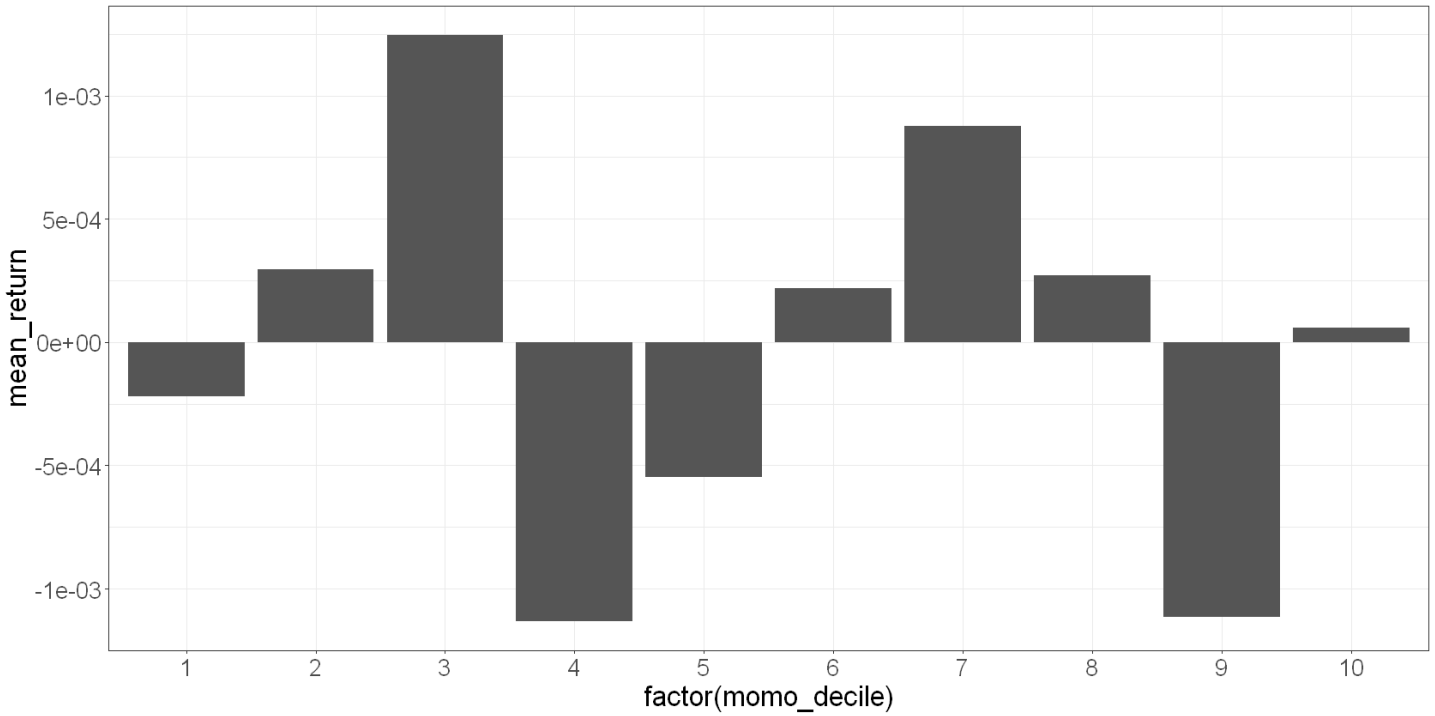
```
model_df %>%
  group_by(carry_decile) %>%
  summarise(mean_return = mean(demeaned_fwd_return)) %>%
  ggplot(aes(x = factor(carry_decile), y = mean_return)) +
  geom_bar(stat = "identity")
```



While not perfectly linear in expected returns, it would be entirely acceptable to model this feature with a simple linear model. In fact, it's probably the *optimal* thing to do because it will limit your ability to overfit to the past. Having said that, to the extent that the outlying return to the tenth decile is real and stable through time, you *might* explicitly account for it in your model - I'd probably just keep things simple though.

In contrast, the momentum feature doesn't look quite so nice on our cut-down universe. In fact, it looks quite random:

```
model_df %>%
  group_by(momo_decile) %>%
  summarise(mean_return = mean(demeaned_fwd_return)) %>%
  ggplot(aes(x = factor(momo_decile), y = mean_return)) +
  geom_bar(stat = "identity")
```

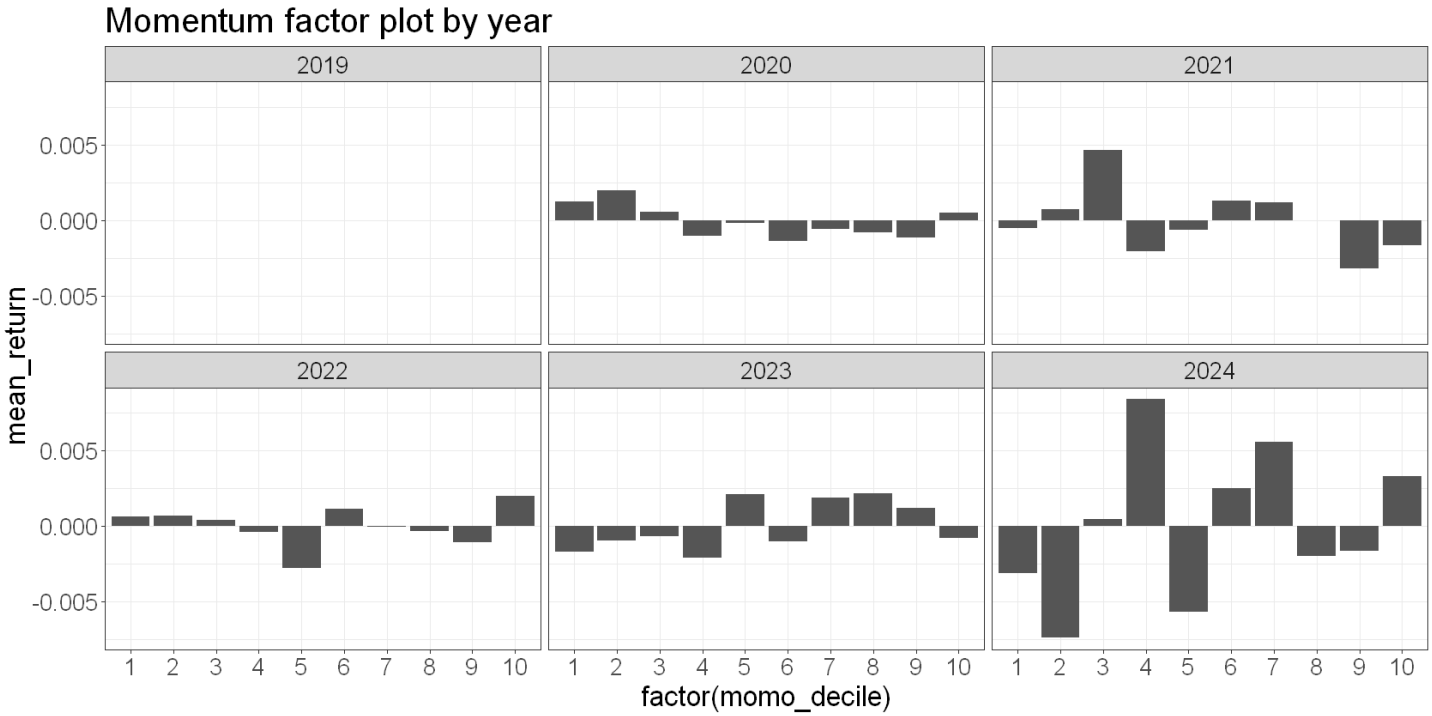


This is interesting. Perhaps there's some liquidity dependency here given that the factor looks rather non-predictive on our more-liquid universe (although it didn't look so good on our larger universe either).

But at this point, it's hard to make the case for a linear model doing this feature justice.

Let's see how this feature's relationship with forward returns has changed over time. We'll plot a factor plot for each year in our data set:

```
model_df %>%
  mutate(Year = year(date)) %>%
  group_by(momo_decile, Year) %>%
  summarise(mean_return = mean(demeaned_fwd_return, na.rm = TRUE), .groups = "drop") %>%
  ggplot(aes(x = factor(momo_decile), y = mean_return)) +
  geom_bar(stat = "identity") +
  facet_wrap(~Year) +
  labs(
    title = "Momentum factor plot by year"
  )
```



Interesting. You could argue that in 2020, 2021, and (maybe) 2022, the momentum feature showed a noisy inverse relationship, implying mean reversion. In 2023 and 2024, that looks to have flipped.

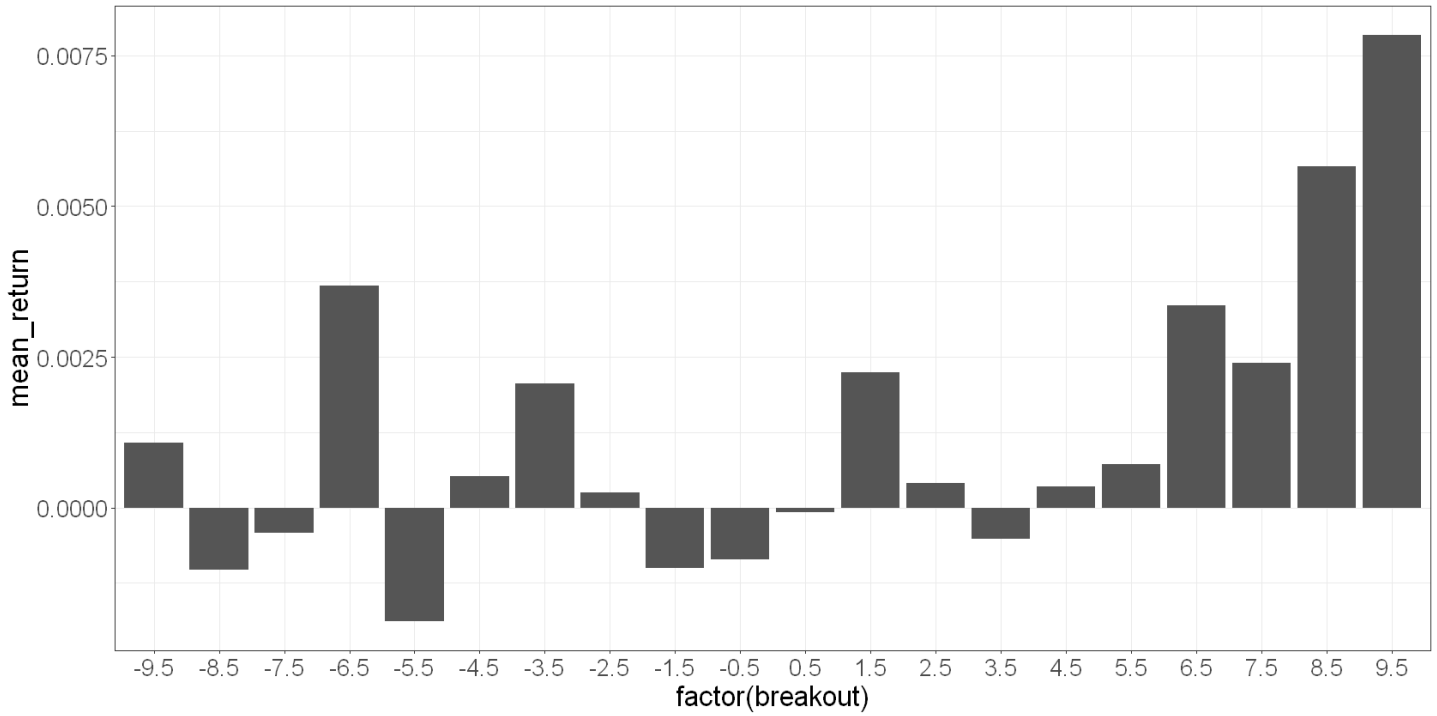
When we take all the years together as in the first plot, it looks like a big random mess. But digging a bit deeper reveals a (potentially) real but changing relationship instead. And you could make a case that a linear model was a *reasonable* choice for each year in the plot above... it's just that a different model seems appropriate for each year.

One significant benefit of the approach we'll use here is that you can update your model of expected returns through time, and thus capture changing relationships such as this.

As an aside, this is the sort of analysis you'd do for all of your features - you really want to understand them in as much detail as possible. But this is already going to be a long article, so let's press on.

Next let's check what our breakout feature looks like:

```
model_df %>%
  group_by(breakout) %>%
  summarise(mean_return = mean(total_fwd_return_simple)) %>%
  ggplot(aes(x = factor(breakout), y = mean_return)) +
  geom_bar(stat = "identity")
```

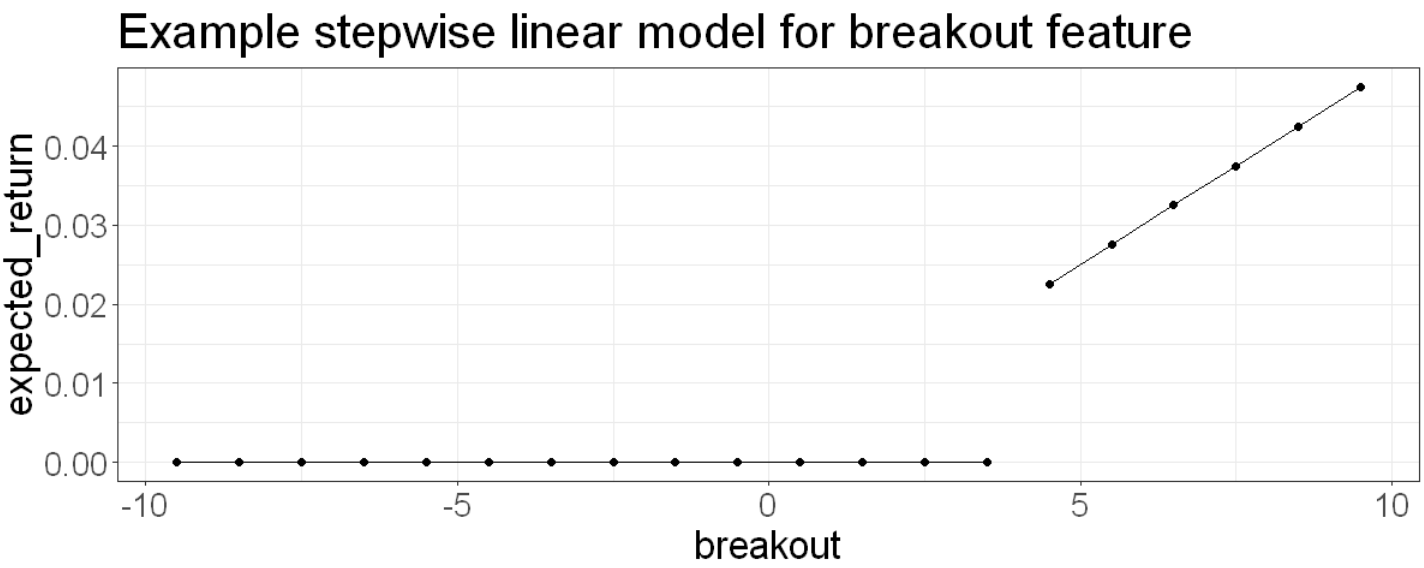


This provides another good example of thinking about a model that adequately describes our feature.

You can see that for the values -9.5 through about 3.5, the relationship with forward returns looks entirely random.

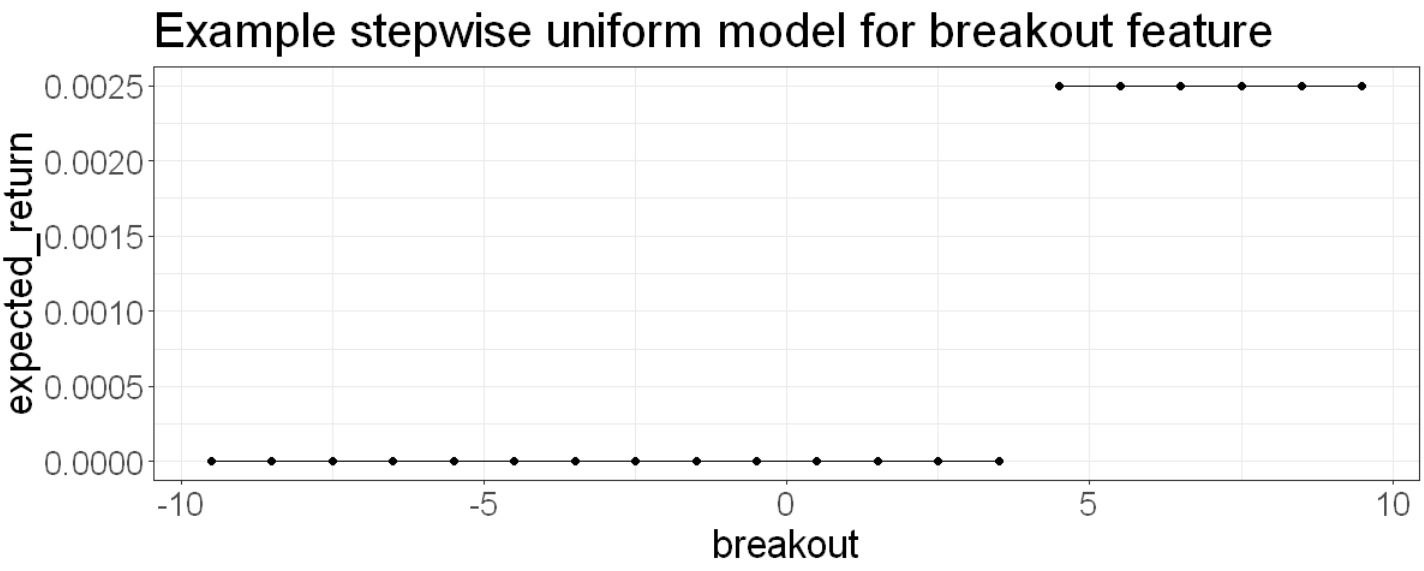
But from about 4.5 onwards, it looks quite non-random. You could argue for a linear model on this portion of the feature's range. This leads to a stepwise linear model of expected returns, where expected returns are zero for -9.5 through 3.5 and linear from 3.5 through 9.5.

```
options(repr.plot.width = 10, repr.plot.height=4)
data.frame(breakout = seq(from = -9.5, to = 9.5, by = 1)) %>%
    mutate(expected_return = case_when(breakout <= 3.5 ~ 0, TRUE ~ breakout * 0.005)) %>%
    mutate(groups = case_when(breakout <= 3.5 ~ 0, TRUE ~ 1)) %>%
    ggplot(aes(x = breakout, y = expected_return, group = groups)) +
    geom_line() +
    geom_point() +
    labs(
        title = "Example stepwise linear model for breakout feature"
    )
```



It would also be quite acceptable to model this feature's expected returns as zero from -9.5 through 3.5 and the mean of the expected returns to the remaining deciles above 3.5:

```
data.frame(breakout = seq(from = -9.5, to = 9.5, by = 1)) %>%
    mutate(expected_return = case_when(breakout <= 3.5 ~ 0, TRUE ~ 0.0025)) %>%
    mutate(groups = case_when(breakout <= 3.5 ~ 0, TRUE ~ 1)) %>%
    ggplot(aes(x = breakout, y = expected_return, group = groups)) +
    geom_line() +
    geom_point() +
    labs(
        title = "Example stepwise uniform model for breakout feature"
    )
# session  options
options(repr.plot.width = 14, repr.plot.height=7)
```



This has the advantage of simplicity, but I do think that it makes sense that this feature would behave linearly at the top end (that is, expected returns are higher the closer we are to the 20-day high).

Now that we have a reasonable specification for our models, let's go ahead and build them.

This is slightly complicated by the fact that we should do this on a rolling basis, only including information that was available at the time.

This leads to rolling estimates for our model coefficients.

These rolling regressions were once a bit of a pain to set up, but the `tidymodels` ecosystem now makes this quite simple and intuitive and integrates with the rest of the `tidyverse`.

This modeling ecosystem is incredibly rich and provides a single interface to many model specifications. The `caret` package was an early attempt at this, but `tidymodels` takes it further with tight integration with the rest of the `tidyverse` and a consistent interface regardless of the model type.

An example of this richness is that we can estimate a standard linear model using ordinary least squares with `lm`. But we can also estimate one with robust standard errors (for example, accounting for autocorrelation and heteroscedasicity) by simply passing the `vcov.` argument, which will instead fit the model using the `sandwich` package - all without changing the model interface. Very cool. This won't make much difference in this application, but I'll include it in the example.

We'll fit our model on 90-day windows of data, and refit every 10 days. 90 days doesn't sound like a lot of data, but in general I find that fitting to the recent past tends to work better.

```
# use a 90-day window and refit every 10 days
is_days <- 90
step_size <- trading_universe_size*10

# rolling model for cross-sectional features
roll_xs_coeffs_df <- model_df %>%
  filter(date >= start_date) %>%
  regress(
    demeaned_fwd_return ~ carry_decile + momo_decile,
    m("lm", vcov. = "HAC"),
    .cv = "sliding_index",
    .cv_args = list(lookback = days(is_days), step = step_size, index = "date"),
    .force_cv = TRUE,
    .return_slices = TRUE
  )

# rolling model for time series features
breakout_cutoff <- 5.5  # below this level, we set our expected return to zero
roll_ts_coeffs_df <- model_df %>%
  filter(date >= start_date) %>%
  # setting regression weights to zero when breakout < breakout_cutoff will give these data
  mutate(regression_weights = case_when(breakout < breakout_cutoff ~ 0, TRUE ~ 1)) %>%
  regress(
    total_fwd_return_simple ~ breakout,
    m("lm", vcov. = "HAC"),
    .weights = "regression_weights",
    .cv = "sliding_index",
    .cv_args = list(lookback = days(is_days), step = step_size, index = "date"),
    .force_cv = TRUE,
    .return_slices = TRUE
  )
```

This results in a nested dataframe that contains the model objects and various metadata:

```
roll_xs_coeffs_df %>% head
roll_ts_coeffs_df %>% select(-settings) %>% head
```

A tidyfit.models: 6 × 7

model	estimator_fct	size (MB)	grid_id	model_object	settings	slice_id
<chr>	<chr>	<dbl>	<chr>	<list>	<list>	<chr>
lm	stats::lm	1.339600	#0010000	<environment: 0x000001ee56a9a2f0>	HAC	2021-02-11
lm	stats::lm	1.345544	#0010000	<environment: 0x000001ee48593470>	HAC	2021-02-21
lm	stats::lm	1.358320	#0010000	<environment: 0x000001ee551fa4a8>	HAC	2021-03-03
lm	stats::lm	1.363056	#0010000	<environment: 0x000001ee554e3170>	HAC	2021-03-13
lm	stats::lm	1.363000	#0010000	<environment: 0x000001ee48e2b898>	HAC	2021-03-23
lm	stats::lm	1.363136	#0010000	<environment: 0x000001ee4769c6a0>	HAC	2021-04-02

A tidyfit.models: 6 × 6

model	estimator_fct	size (MB)	grid_id	model_object	slice_id
<chr>	<chr>	<dbl>	<chr>	<list>	<chr>
lm	stats::lm	1.224192	#0010000	<environment: 0x000001ee47e4b2d0>	2021-02-11
lm	stats::lm	1.236392	#0010000	<environment: 0x000001ee548f9590>	2021-02-21
lm	stats::lm	1.233184	#0010000	<environment: 0x000001ee54a51478>	2021-03-03
lm	stats::lm	1.233880	#0010000	<environment: 0x000001ee54dc9dc0>	2021-03-13
lm	stats::lm	1.238816	#0010000	<environment: 0x000001ee5519bf28>	2021-03-23
lm	stats::lm	1.235976	#0010000	<environment: 0x000001ee5569ce28>	2021-04-02

`slice_id` is the date the model goes out of sample - so we'll need to make sure that we align our model coefficients to avoid using them on the data they were fitted on.

This requires a little data wrangling:

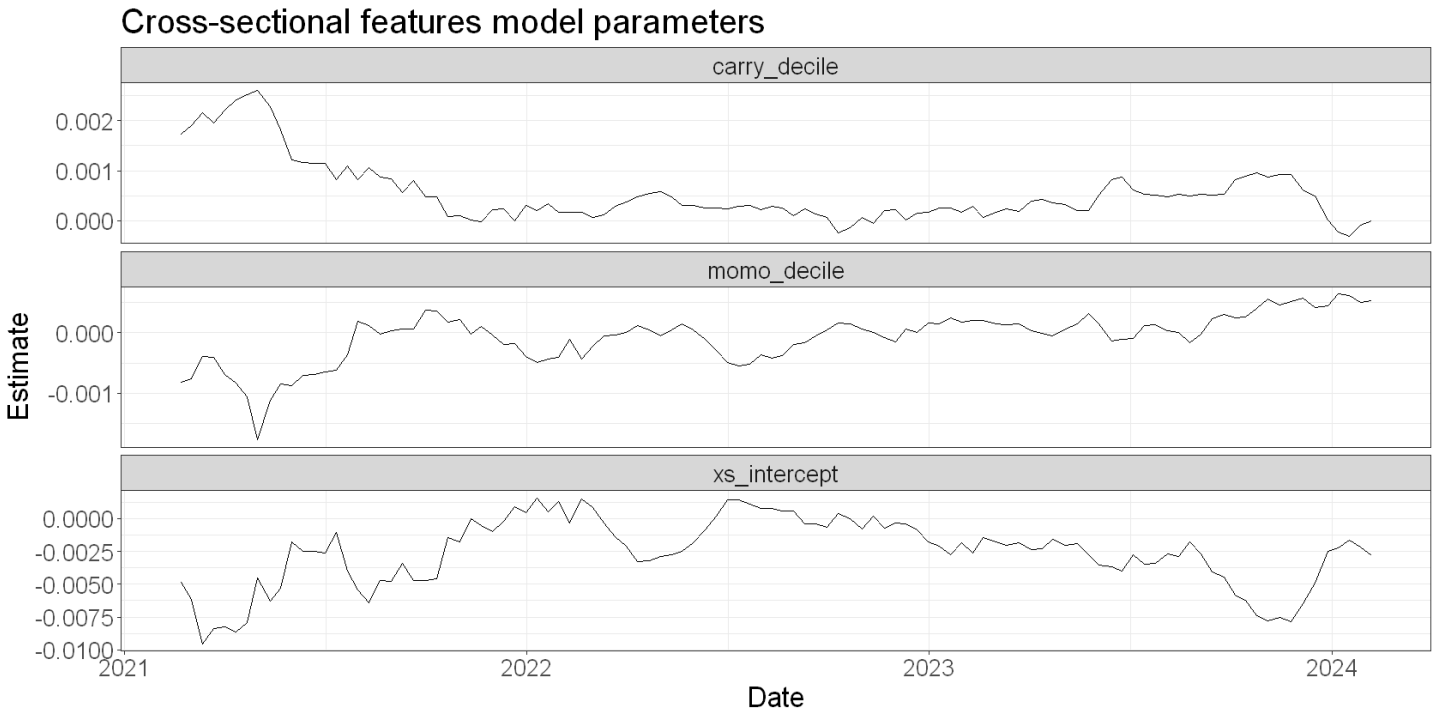
```r
# for this to work, need to install.packages("sandwich", "lmtest")
xs_coefs <- roll_xs_coeffs_df %>%
  coef()

xs_coefs_df <- xs_coefs %>%
  ungroup() %>%
  select(term, estimate, slice_id) %>%
  pivot_wider(id_cols = slice_id, names_from = term, values_from = estimate) %>%
  mutate(slice_id = as_date(slice_id)) %>%
  # need to lag slice id to make it oos
  # slice_id_oos is the date we start using the parameters
  mutate(slice_id_oos = lead(slice_id)) %>%
  rename("xs_intercept" = `(Intercept)`)

ts_coefs <- roll_ts_coeffs_df %>%
  coef()

ts_coefs_df <- ts_coefs %>%
  ungroup() %>%
  select(term, estimate, slice_id) %>%
  pivot_wider(id_cols = slice_id, names_from = term, values_from = estimate) %>%
  mutate(slice_id = as_date(slice_id)) %>%
  # need to lag slice id to make it oos
  # slice_id_oos is the date we start using the parameters
  mutate(slice_id_oos = lead(slice_id)) %>%
  rename("ts_intercept" = `(Intercept)`)

xs_coefs_df %>% head
# ts_coefs_df %>% head
```

A tibble: 6 × 5

slice_id	xs_intercept	carry_decile	momo_decile	slice_id_oos
<date>	<dbl>	<dbl>	<dbl>	<date>
2021-02-11	-0.004872208	0.001732322	-0.0008239738	2021-02-21
2021-02-21	-0.006129608	0.001902553	-0.0007640190	2021-03-03
2021-03-03	-0.009538180	0.002148154	-0.0003897220	2021-03-13
2021-03-13	-0.008374302	0.001953655	-0.0004141515	2021-03-23
2021-03-23	-0.008275046	0.002211137	-0.0006898811	2021-04-02
2021-04-02	-0.008613423	0.002413307	-0.0008298451	2021-04-12

Here's a plot of our cross-sectional features' regression coefficients through time:

```r
# plot cross-sectional estimates
xs_coefs_df %>%
  select(-slice_id) %>%
  pivot_longer(cols = -slice_id_oos, names_to = "coefficient", values_to = "estimate") %>%
  ggplot(aes(x = slice_id_oos, y = estimate)) +
  geom_line() +
  facet_wrap(~coefficient, ncol = 1, scales = "free_y") +
  labs(
    title = "Cross-sectional features model parameters",
    x = "Date",
    y = "Estimate"
  )

# plot time-series estimates
# ts_coefs_df %>%
#   select(-slice_id) %>%
#   pivot_longer(cols = -slice_id_oos, names_to = "coefficient", values_to = "estimate") %>
#   ggplot(aes(x = slice_id_oos, y = estimate)) +
#   geom_line() +
#   facet_wrap(~coefficient, ncol = 1, scales = "free_y") +
#   labs(
#     title = "Time-series features model parameters",
#     x = "Date",
#     y = "Estimate"
#   )
```



The estimates for the model coefficients for our carry and momentum features change over time to reflect the changing relationship with forward returns.

In particular, notice how the momentum coefficient flipped sign a few times, but especially from mid-2022, which is in line with our understanding of how the feature evolved.

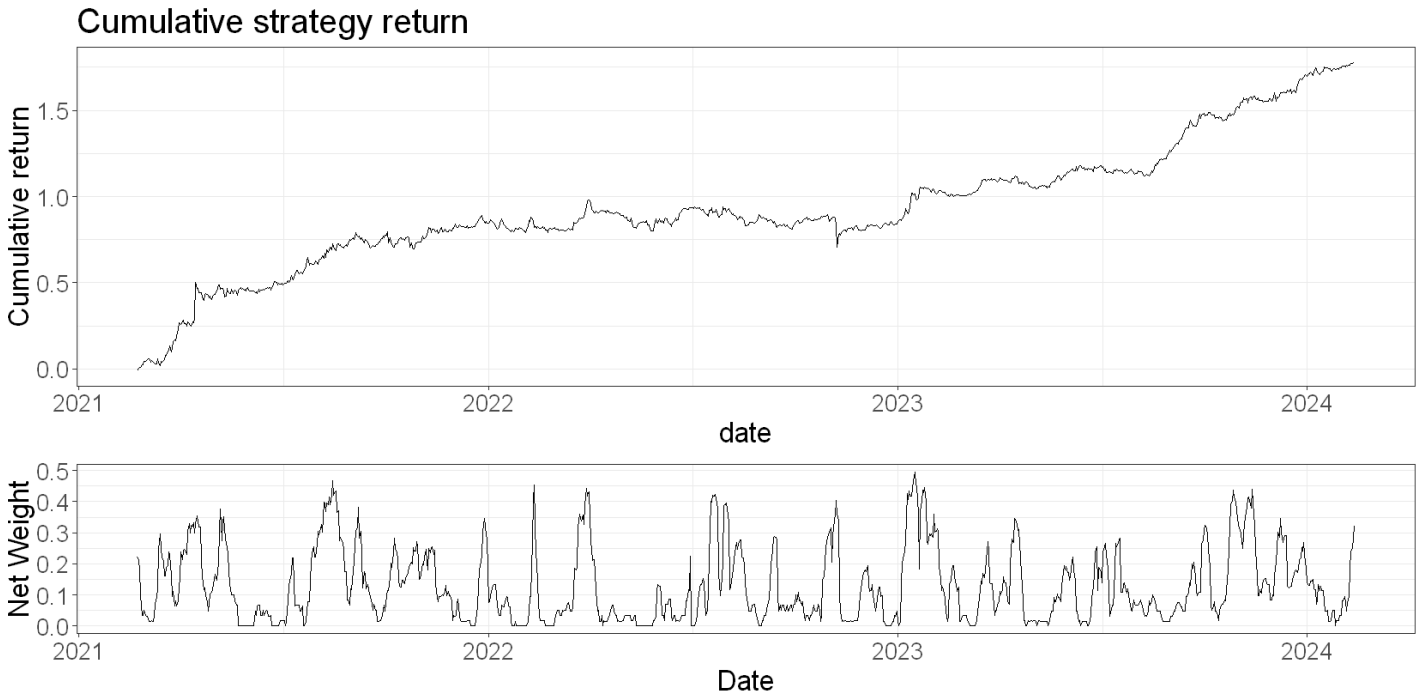
Now we can plot a time series of returns to a frictionless trading strategy based on these expected return estimates. This isn't a backtest - it makes no attempt to address real-world issues such as costs and turnover. It simply plots the returns to our predictions of expected returns over time.

I won't actually use the linear model of the breakout feature - instead I'll just set its expected return to 0.002 when it's greater than 5 and 0 otherwise. I don't like that the breakout coefficients go negative from time to time.

I'll calculate target positions proportional to their cross-sectional return estimates. I'll then let the breakout feature tilt the portfolio net long, but I'll constrain the maximum delta that this feature can add to a position.

```r
# join and fill using slice_id to designate when the model goes oos
exp_return_df <- model_df %>%
  left_join(
    xs_coefs_df %>% left_join(ts_coefs_df, by = c("slice_id", "slice_id_oos")),
    by = join_by(closest(date > slice_id_oos)), suffix = c("_factor", "_coef")
  ) %>%
  na.omit() %>%
  # forecast cross-sectional expected return as
  mutate(expected_xs_return = carry_decile_factor*carry_decile_coef + momo_decile_factor*mo
  # mean expected xs return each day is zero
  # let total expected return be xs return + ts return - allows time series expected return
  mutate(expected_ts_return = case_when(breakout_factor >= 5.5 ~ 0.002, TRUE ~ 0)) %>%
  ungroup()

# long-short the xs expected return
# layer ts expected return on top
# position by expected return

# 1 in the numerator lets it get max 100% long due to breakout
max_ts_pos <- 0.5/trading_universe_size

strategy_df <- exp_return_df %>%
  filter(date >= start_date) %>%
  group_by(date) %>%
  mutate(xs_position = expected_xs_return - mean(expected_xs_return, na.rm = TRUE)) %>%
  # scale positions so that leverage is 1
  group_by(date) %>%
  mutate(xs_position = if_else(xs_position == 0, 0, xs_position/sum(abs(xs_position)))) %>%
  # layer ts expected return prediction
  ungroup() %>%
  mutate(ts_position = sign(expected_ts_return)) %>%
  # constrain maximum delta added by time series prediction
  mutate(ts_position = if_else(ts_position >= 0, pmin(ts_position, max_ts_pos), pmax(ts_pos
  mutate(position = xs_position + ts_position) %>%
  # strategy return
  mutate(strat_return = position*total_fwd_return_simple) %>%
  # scale back to leverage 1
  group_by(date) %>%
  mutate(position = if_else(position == 0, 0, position/sum(abs(position))))

returns_plot <- strategy_df %>%
  group_by(date) %>%
  summarise(total_ret = sum(strat_return)) %>%
  ggplot(aes(x = date, y = cumsum(log(1+total_ret)))) +
  geom_line() +
  labs(
    title = "Cumulative strategy return",
    y = "Cumulative return"
  )

weights_plot <- strategy_df %>%
  summarise(net_pos = sum(position)) %>%
  ggplot(aes(x = date, y = net_pos)) +
```

```
  geom_line() +
  labs(
    x = "Date",
    y = "Net Weight"
  )

returns_plot / weights_plot + plot_layout(heights = c(2,1))
```



Notice that I still had to make some manual adjustments to the positions resulting from the time-series predictions which were quite imprecise (for instance, using the sign of the prediction as the position then scaling it back using a maximum delta constraint).

This is OK, but I'd rather use my predictions a little more directly. I want to model my features as best I can, and then at each decision point, I want to answer the question "Given these expected returns, what's the best portfolio given my constraints?" That's an optimisation problem, and one that we can actually solve without resorting to manual adjustments and heuristics. I'll show you how to do that in the next article.

But first, let's do a more accurate simulation given our target weights and trading costs. We'll use `rsims`, like in the previous example.

First, wrangle our data into matrixes of target positions, prices, and funding rates:

```r
# get weights as a wide matrix
# note that date column will get converted to unix timestamp
backtest_weights <- strategy_df %>%
  pivot_wider(id_cols = date, names_from = ticker, values_from = c(close, position)) %>%  #
  select(date, starts_with("position_")) %>%
  data.matrix()

# NA weights should be zero
backtest_weights[is.na(backtest_weights)] <- 0

head(backtest_weights, c(5, 5))

# get prices as a wide matrix
# note that date column will get converted to unix timestamp
backtest_prices <- strategy_df %>%
  pivot_wider(id_cols = date, names_from = ticker, values_from = c(close, position)) %>%  #
  select(date, starts_with("close_")) %>%
  data.matrix()

head(backtest_prices, c(5, 5))

# get funding as a wide matrix
# note that date column will get converted to unix timestamp
backtest_funding <- strategy_df %>%
  pivot_wider(id_cols = date, names_from = ticker, values_from = c(close, funding_rate)) %>
  select(date, starts_with("funding_rate_")) %>%
  data.matrix()

head(backtest_funding, c(5, 5))
```

A matrix: 5 × 5 of type dbl

date	position_1INCHUSDT	position_AAVEUSDT	position_ADAUSDT	position_ALPHAUSD
18680	0.002694931	0.01054433	0.0762376871	0.01470442
18681	0.027559434	-0.07675613	0.0125204272	-0.01039933
18682	0.040915764	-0.05302819	0.0012411217	0.00209787
18683	0.014953494	-0.02667621	0.0062533668	-0.01029254
18684	-0.023077149	0.01078900	0.0002980306	-0.02370630

A matrix: 5 × 5 of type dbl

date	close_1INCHUSDT	close_AAVEUSDT	close_ADAUSDT	close_ALPHAUSDT
18680	4.3783	399.582	0.98801	1.44859
18681	3.6600	347.445	0.97291	1.24510
18682	3.7108	384.990	1.02610	1.28602
18683	4.1634	379.815	1.12148	1.69307
18684	4.7594	340.026	1.16322	1.34361

A matrix: 5 × 5 of type dbl

date	funding_rate_1INCHUSDT	funding_rate_AAVEUSDT	funding_rate_ADAUSDT	fundin
18680	-0.00136194	-0.00031975	-0.00000653	
18681	-0.00025597	-0.00077154	-0.00030000	
18682	-0.00021475	-0.00030000	-0.00030000	
18683	-0.00040013	-0.00045891	-0.00060473	
18684	-0.00030000	-0.00010783	-0.00030000	

We'll start with a cost-free simulation that trades frictionlessly into our target positions. The result should look a lot like the returns plot above:

```r
# cost-free, no trade buffer
# simulation parameters
initial_cash <- 10000
fee_tier <- 0
capitalise_profits <- FALSE  # remain fully invested?
trade_buffer <- 0.
commission_pct <- 0.
margin <- 0.05

# simulation
results_df <- fixed_commission_backtest_with_funding(
  prices = backtest_prices,
  target_weights = backtest_weights,
  funding_rates = backtest_funding,
  trade_buffer = trade_buffer,
  initial_cash = initial_cash,
  margin = margin,
  commission_pct = commission_pct,
  capitalise_profits = capitalise_profits
  ) %>%
  mutate(ticker = str_remove(ticker, "close_")) %>%
  # remove coins we don't trade from results
  drop_na(Value)
```

```r
# make a nice plot with some summary statistics
# plot equity curve from output of simulation
plot_results <- function(backtest_results, trade_on = "close") {
  margin <- backtest_results %>%
    group_by(Date) %>%
    summarise(Margin = sum(Margin, na.rm = TRUE))

  cash_balance <- backtest_results %>%
    filter(ticker == "Cash") %>%
    select(Date, Value) %>%
    rename("Cash" = Value)

  equity <- cash_balance %>%
    left_join(margin, by = "Date") %>%
    mutate(Equity = Cash + Margin)

  fin_eq <- equity %>%
    tail(1) %>%
    pull(Equity)

  init_eq <- equity %>%
    head(1) %>%
    pull(Equity)

  total_return <- (fin_eq/init_eq - 1) * 100
  days <- nrow(equity)
  ann_return <- total_return * 365/days
  sharpe <- equity %>%
    mutate(returns = Equity/lag(Equity)- 1) %>%
    na.omit() %>%
    summarise(sharpe = sqrt(365)*mean(returns)/sd(returns)) %>%
    pull()

  equity %>%
    ggplot(aes(x = Date, y = Equity)) +
      geom_line() +
      labs(
        title = "Crypto Stat Arb Simulation",
        subtitle = glue::glue(
          "Costs {commission_pct*100}% of trade value, trade buffer = {trade_buffer}, trade
          {round(total_return, 1)}% total return, {round(ann_return, 1)}% annualised, Sharp
        )
      )
}

# calculate sharpe ratio from output of simulation
calc_sharpe <- function(backtest_results) {
  margin <- backtest_results %>%
    group_by(Date) %>%
    summarise(Margin = sum(Margin, na.rm = TRUE))

  cash_balance <- backtest_results %>%
    filter(ticker == "Cash") %>%
```

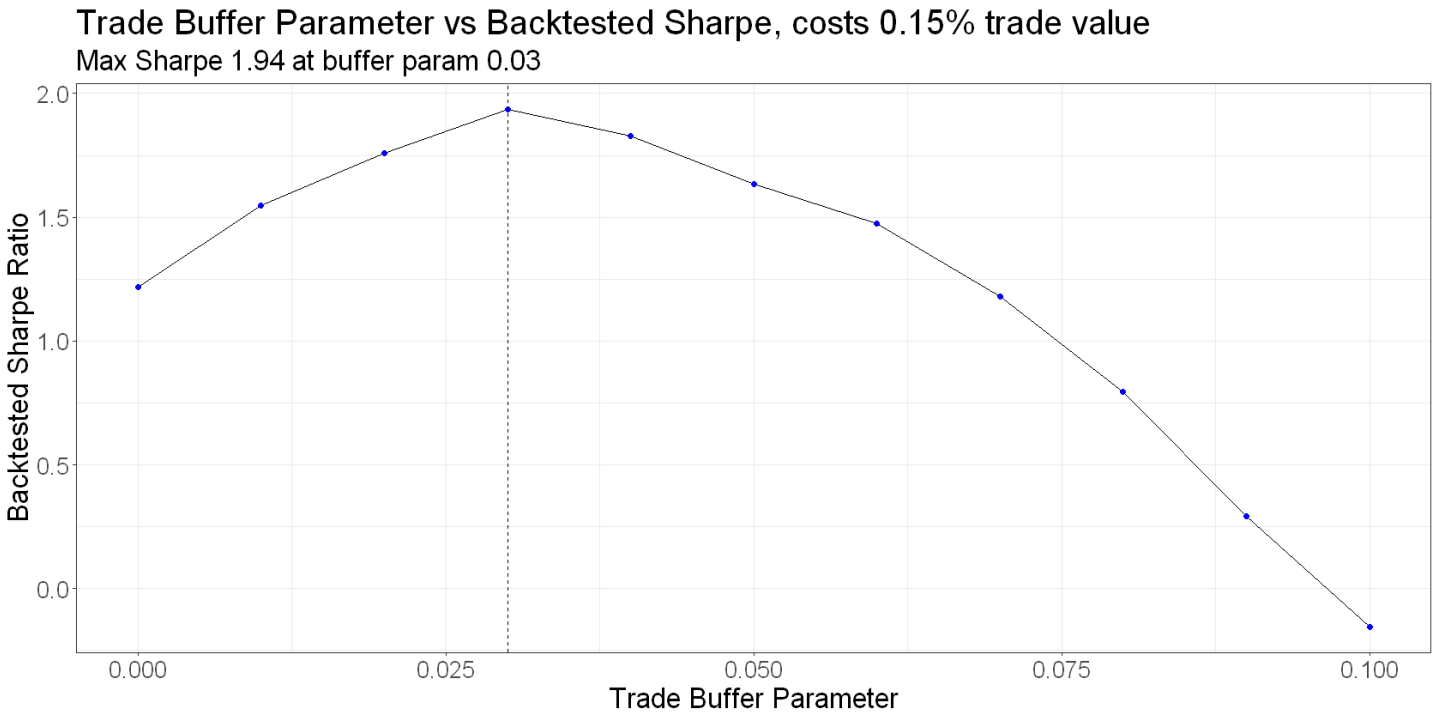
```
    select(Date, Value) %>%
    rename("Cash" = Value)

  equity <- cash_balance %>%
    left_join(margin, by = "Date") %>%
    mutate(Equity = Cash + Margin)

    equity %>%
        mutate(returns = Equity/lag(Equity)- 1) %>%
        na.omit() %>%
        summarise(sharpe = sqrt(355)*mean(returns)/sd(returns)) %>%
        pull()
}

plot_results(results_df)
```

Now we'll add costs:

```
# explore costs-turnover tradeoffs
# with costs, no trade buffer
commission_pct <- 0.0015

# simulation
results_df <- fixed_commission_backtest_with_funding(
  prices = backtest_prices,
  target_weights = backtest_weights,
  funding_rates = backtest_funding,
  trade_buffer = trade_buffer,
  initial_cash = initial_cash,
  margin = margin,
  commission_pct = commission_pct,
  capitalise_profits = capitalise_profits
  ) %>%
  mutate(ticker = str_remove(ticker, "close_")) %>%
  # remove coins we don't trade from results
  drop_na(Value)

results_df %>%
  plot_results()
```

Costs are quite a drag on performance. Let's use the no-trade buffer heuristic from the last article to do the minimum amount of trading to harness the edge:

```r
# find appropriate trade buffer by optimising historical sharpe
sharpes <- list()
trade_buffers <- seq(0, 0.1, by = 0.01)
for(trade_buffer in trade_buffers) {
  sharpes <- c(
    sharpes,
    fixed_commission_backtest_with_funding(
      prices = backtest_prices,
      target_weights = backtest_weights,
      funding_rates = backtest_funding,
      trade_buffer = trade_buffer,
      initial_cash = initial_cash,
      margin = margin,
      commission_pct = commission_pct,
      capitalise_profits = capitalise_profits
    ) %>%
      calc_sharpe()
  )
}

sharpes <- unlist(sharpes)
data.frame(
  trade_buffer = trade_buffers,
  sharpe = sharpes
) %>%
  ggplot(aes(x = trade_buffer, y = sharpe)) +
    geom_line() +
    geom_point(colour = "blue") +
    geom_vline(xintercept = trade_buffers[which.max(sharpes)], linetype = "dashed") +
    labs(
      x = "Trade Buffer Parameter",
      y = "Backtested Sharpe Ratio",
      title = glue::glue("Trade Buffer Parameter vs Backtested Sharpe, costs {commission_pc
      subtitle = glue::glue("Max Sharpe {round(max(sharpes), 2)} at buffer param {trade_buf
    )
```



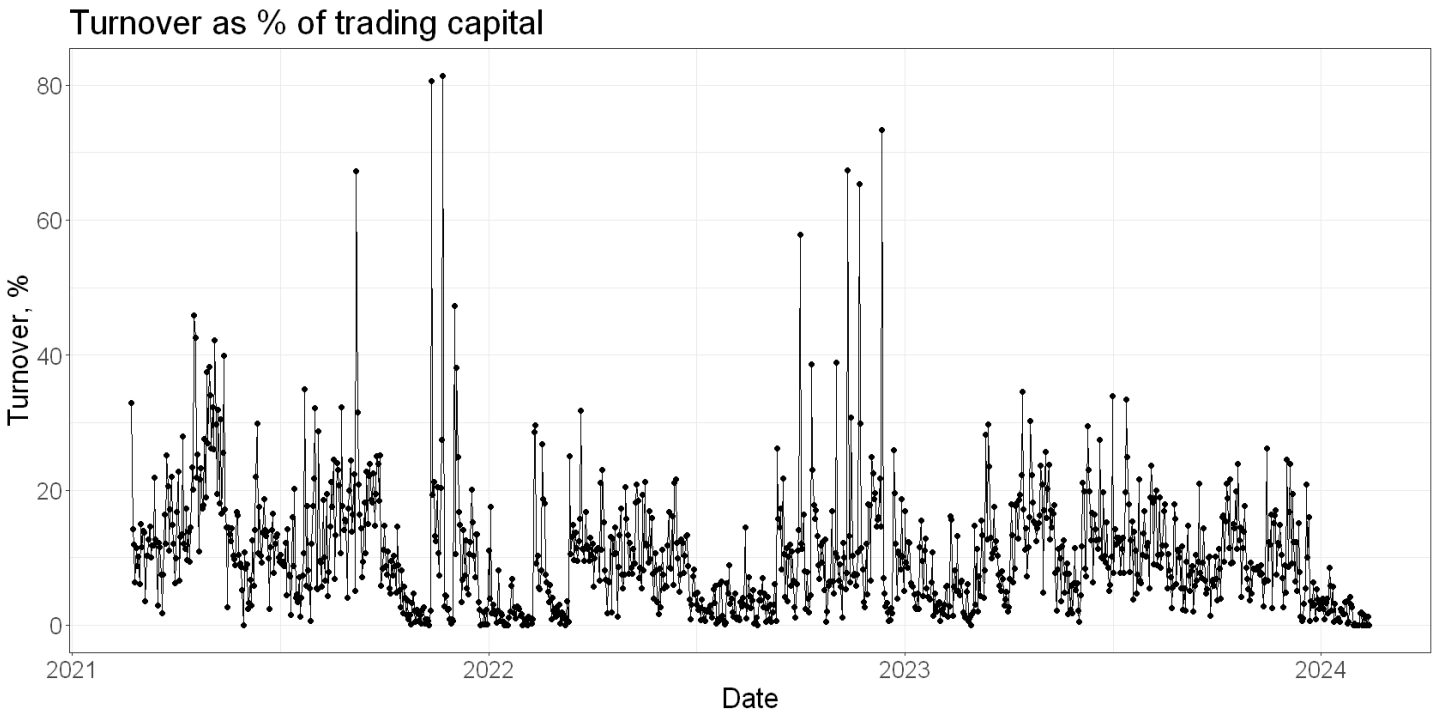
A value of 0.03 maximised our historical after-cost Sharpe. You might pick a value a little higher than 0.03 to mitigate the risk that your out-of-sample performance will be worse than your in-sample (almost always a good assumption). But for now, let's just simulate 0.03:

```
# get back original with costs simulation results
trade_buffer <- 0.03

# simulation
results_df <- fixed_commission_backtest_with_funding(
    prices = backtest_prices,
    target_weights = backtest_weights,
    funding_rates = backtest_funding,
    trade_buffer = trade_buffer,
    initial_cash = initial_cash,
    margin = margin,
    commission_pct = commission_pct,
    capitalise_profits = capitalise_profits
  ) %>%
  mutate(ticker = str_remove(ticker, "close_")) %>%
  # remove coins we don't trade from results
  drop_na(Value)

# simulation results
results_df %>%
  plot_results()
```

Performance is a little higher with this approach (note that this simulation starts later than the previous one because we have to exclude the first in-sample model estimation period - over the same period, this one does better, especially recently).

Turnover is a bit higher, but that's more to do with the lower trade buffer parameter selected:

```
results_df %>%
  filter(ticker != "Cash") %>%
  group_by(Date) %>%
  summarise(Turnover = 100*sum(abs(TradeValue))/initial_cash) %>%
  ggplot(aes(x = Date, y = Turnover)) +
    geom_line() +
    geom_point() +
    labs(
      title = "Turnover as % of trading capital",
      y = "Turnover, %"
    )
```



While this is a good result for such a simple modeling approach, the main benefit of this approach is that your features are all modelled on the same scale, making them directly comparable. It also means that incorporating new signals into your system is straightforward and follows the same modeling process. We also saw that we can potentially capture changing feature dynamics directly in the modeling process.

New biases

It's important to be aware of the new biases that this approach introduces.

For example, there's a degree of future peeking in specifying a model type for each feature. We chose a linear model for carry and momentum because we saw a (noisily) linear relationship between these features and expected returns. And while we estimate the coefficients of these models on a rolling basis that prevents future peeking, the model type itself was chosen based on knowledge of the entire dataset.

We also introduce two new meta-parameters: the length of the model estimation window, and the frequency with which we refit the model. Inevitably, you'll try a bunch of values and intoduce some data snooping that way as well.

I think these are fairly benign biases, but they're real and it's a good idea to assume that performance will deteriorate out of sample as a result.

Conclusion

Modeling features as expected returns provides a consistent framework for making trading decisions, including by structuring them as an optimisation problem.

However, it does require some care.

In particular, you need to have a good understanding of your features:

- How they're distributed
- How their relationship with expected returns changes across their range
- Whether there is a sensible basis for modeling those changes or whether they're just noise
- How their predictive power changes through time

You need to do the grunt work.

You also need to understand the biases you introduce by adopting this approach.

Soon, we'll swap out our heuristic approach for managing turnover with an approach that solves an optimisation problem at each decision step. This allows you to directly compare expected returns with costs, and can also accommodate a risk model and real-world constraints. But before we get to that, in the next chapter I'll help you develop some intuition for how portfolio optimisation works.

Building Intuition of Convex Optimisation in Trading with CVXR

Next, we'll build some intuition for using convex optimisation to manage real-world trade-offs in execution.

Imagine you've quantified some signals that are predictive of next-day returns. You've done the grunt work of the research phase:

- Understood their strength and decay characteristics
- Gotten a handle on the rate at which they change and hence their impact on turnover
- Examined how they changed over time
- Modelled them as expected returns

The next phase is implementation. This answers the question: "Given these signals and my real-world constraints and objectives, what trading decisions am I going to make?"

This sounds straightforward, but it's nuanced.

Often, if you naively trade the positions implied by your signals (for example without considering costs), you end up trading more frequently than you need to and getting killed with costs.

Implementation is all about navigating trade-offs. For example, you have a signal that is noisily correlated with future returns. But you also incur a cost every time you trade.

Maybe you care about ending up with positions concentrated in similar assets as well as maximising your expected returns.

Maybe you have a limit on how net long or short you want to be, or how much leverage you want to take.

Clearly, your signals are only one input into your trading decisions.

In the article *A Simple, Effective Way to Manage Turnover and Not Get Killed by Costs*, we introduced the concept of a "no-trade buffer" that stops you hyperactively trading your signals, and instead only rebalances positions when they get out of whack by some percentage of your target.

The advantage of this approach is that it's simple and intuitive and leads to a predictable, mechanical set of rules for managing your positions.

However, a more explicit optimisation approach has some attractive benefits:

- It accommodates real-world constraints directly
- It enables incorporation of risk models such as covariance estimates, Value-at-Risk, etc
- It's very flexible and scalable: you can add new signals or risk estimates without re-fitting anything

Before we get into the details of how to apply optimisation in the trading context, it's worth building some intuition. Optimisers can seem like a black box, but we can build up some intuition about how they work using some simple examples.

Let's get to it.

```r
# session  options
options(repr.plot.width = 14, repr.plot.height=7, warn = -1)

library(tidyverse)
library(CVXR)

# chart options
theme_set(theme_bw())
theme_update(text = element_text(size = 20))
```

Why convex optimisation?

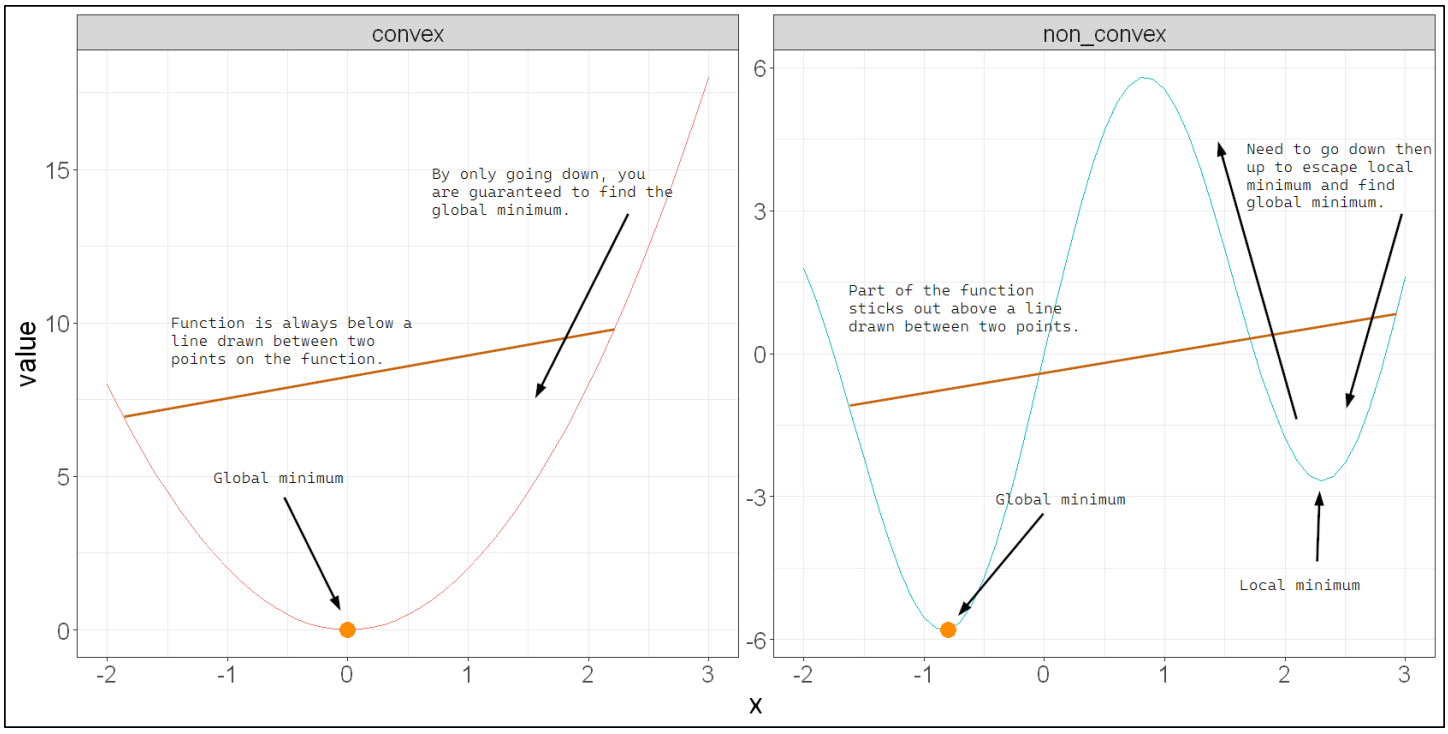
In a nutshell, convex problems are fast to solve, and there's always a clear path to the solution.

They're amenable to solving quickly and accurately because the problem is bounded in such a way that you can't get stuck in a local minimum or maximum, and you get to the right answer by always going in the same mathematical direction.

The common definition of a convex function is a little wordy:

> If we can draw a line segment between any two points on the graph of a function such that there is no point of this graph that is above this line segment between these two points then the function is called a convex function.

This is a visual representation of what it means (right click, open in new tab to make bigger):



Because of the bowl shape, we know a few useful things:

- There's only one optimal spot.
- We can find it without getting lost or stuck at a higher point - we just need to keep going down.

- We can find the best spot efficiently. Since there's a straightforward path to the bottom, we don't have to check every single spot on the function to find the lowest point. We can follow a path that takes us directly there, which saves a lot of time and effort.
- It works every time. No matter where you start on the function, you'll always end up at the bottom if you keep going downhill. This predictability makes it easier for us to solve these problems without surprises.

The non-convex function on the right requires more care and effort. If we find a local minimum, there's no guarantee that we found the global minimum.

Thankfully, many problems can be framed as convex problems, including the trading problem.

Modeling convex problems in R

CVXR is an R package that provides a simple, object-oriented modeling language for convex optimization.

It's strength lies in allowing the user to formulate convex optimization problems in a natural, intuitive syntax. In contrast, many optimisers require restrictive syntax that's heavy on the linear algebra, making it inaccessible to many users.

In CVXR, the user specifies an objective and set of constraints by combining CVXR objects representing constants, variables, and parameters using a library of functions with known mathematical properties.

CVXR then automatically verifies the problem's convexity. Once verified, the problem is automatically converted into standard form and passed to an underlying solver.

Building intuition through examples

Next, we'll explore CVXR and build up our intuition of what the optimiser does via gradually more complex examples.

Maximise portfolio returns given expected returns

In this simplest of problems, we seek only to maximise expected returns. We don't consisder costs or risk. We only add no-leverage and long-only constraints.

We are asking the question: *"Given expected returns for these assets, what weights maximise my portfolio expected return?"*

Under this scenario, since it's considering nothing but expected returns, the optimiser should invest 100% in whatever asset has the highest expected return, and return a weight of 0% for everything else.

We first specify a function for solving this optimisation problem:

```
maximise_returns <- function(expected_returns) {
  # define our unknown weight vector as a CVXR::Variable
    # this is the thing that CVXR will try to solve for
  weights <- Variable(rows = length(expected_returns))
  # define our objective:
    # portfolio returns = sum(weights * expected_returns)
    # we want to maximise this, so use CVXR::Maximise
    # and express weighted sum of returns using linear algebra - transpose the weight vecto
  objective <- Maximize(t(weights) %*% expected_returns)
  # define our constraints as a list:
    # here we have only two constraints:
      # that the sum of the abs.values of the weights be less than or equal to 1 (ie no lev
      # and that each weight is greater than or equal to zero (long only)
    # CVXR::cvxr_norm is a function for getting the sum of the abs.values of a vector
    # CVXR has loads of functions for doing all sorts of operations and aggregations
  constraints <- list(cvxr_norm(weights, 1) <= 1, weights >= 0)  # sum(abs(weights))
  # construct our problem using CVXR::Problem and specifying objective and constraints
  problem <- Problem(objective, constraints)
  # solve
  result <- solve(problem)

  # return the solved values of our CVXR::Variable
  result$getValue(weights)
}
```

Here's our vector of expected returns.

```
# vector of expected returns
r1 <- seq(from = 0.001, by = 0.001, length.out = 6)
r1
```

0.001 · 0.002 · 0.003 · 0.004 · 0.005 · 0.006

Asset 6 has the highest expected return, so we should go all in on this asset (weight should be 1) and assign a zero weight to everything else:

```
# vector of weights we expect the optimiser to find
expected_weights <- matrix(c(0, 0, 0, 0, 0, 1), ncol = 1)  # fully invested in highest expe

weights1 <- maximise_returns(r1)
weights1

weights1 %>%
  as.data.frame() %>%
  ggplot(aes(x = factor(1:6), y = V1)) +
    geom_col() +
    labs(
      x = "asset",
      y = "weight",
      title = "Optimal weights"
    )

# check we got what we expected
all.equal(weights1, expected_weights, tolerance = 1e-9)
```

A matrix: 6 × 1
of type dbl

-1.173742e-15

-3.200916e-16

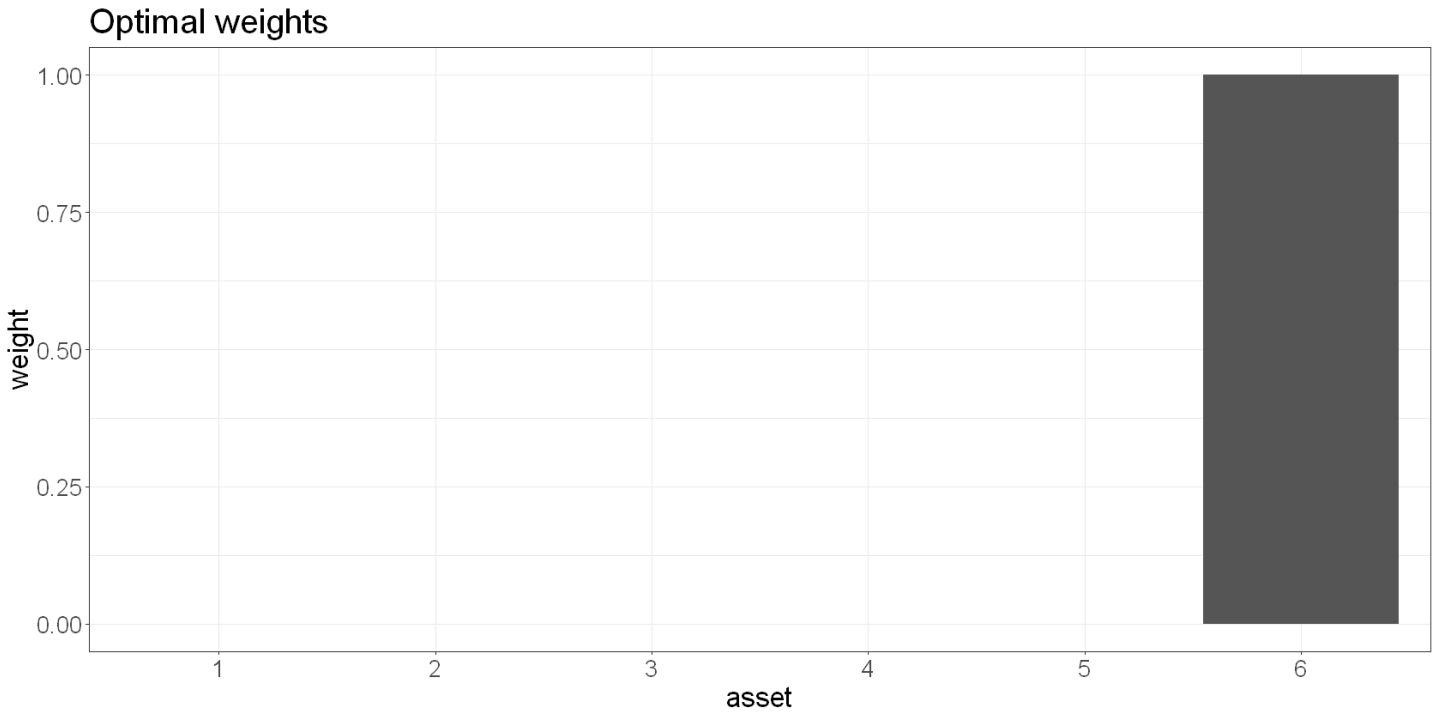
-2.997776e-16

-2.771167e-15

-9.126314e-15

1.000025e+00

'Mean relative difference: 2.470575e-05'



You can see that the optimiser did indeed go all in on asset 6. Also notice that the other weights go extremely close, but not precisely to zero. That's because the optimiser uses a numerical method rather than a precise analytical solution to solve the problem.

Let's verify the long-only constraint. We'll set asset 6's return to be -0.006. We could now maximise our expected portfolio returns by assigning 100% of our position to a short on this asset. But since we have a long-only constraint, this asset should be assigned a zero weight, and asset 5 (expected return 0.005) should get 100% of our weight:

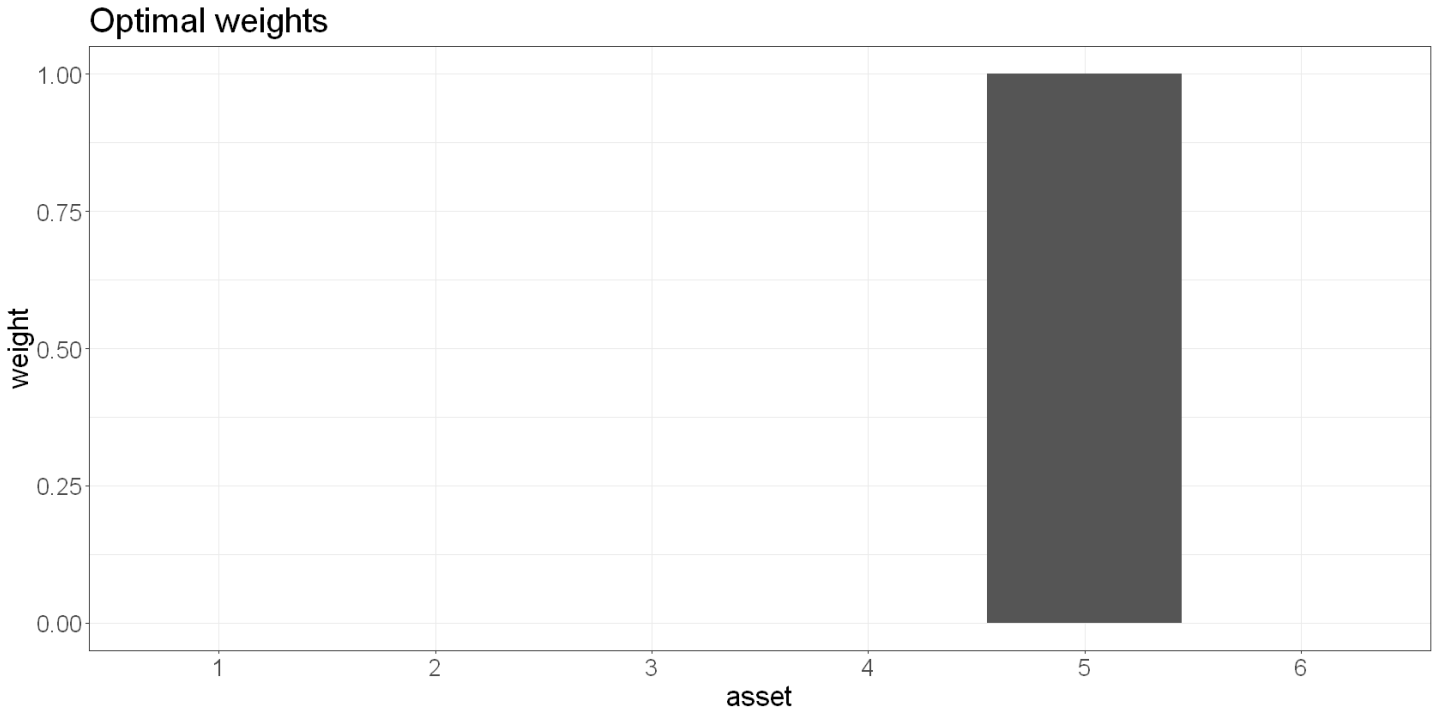
```r
# vector of expected returns
r1a <- r1*c(rep(1, 5), -1)
r1a

# vector of weights we expect the optimiser to find
expected_weights <- matrix(c(0, 0, 0, 0, 1, 0), ncol = 1)  # fully invested in highest expe

weights1a <- maximise_returns(r1a)
weights1a

weights1a %>%
  as.data.frame() %>%
  ggplot(aes(x = factor(1:6), y = V1)) +
    geom_col() +
    labs(
      x = "asset",
      y = "weight",
      title = "Optimal weights"
    )

# check we got what we expected
all.equal(weights1a, expected_weights, tolerance = 1e-9)
```

0.001 · 0.002 · 0.003 · 0.004 · 0.005 · -0.006

A matrix: 6 × 1
of type dbl

1.833225e-16
1.190608e-16
-2.071393e-16
-1.427314e-16
1.000074e+00
-1.191854e-16

'Mean relative difference: 7.402931e-05'



You can see that the optimiser did indeed find an optimal weight of 100% for asset 5 and 0% for everything else.

Maximise expected return given costs and our initial position

The next problem is slightly more complex and starts to reflect what we would think about in the trading problem - maximising our expected return subject to transaction costs, given our existing position.

Imagine you have three assets A, B, and C with the following expected returns:

- A: 0.002
- B: 0.001
- C: 0.001

So you buy asset A in an effort to maximise expected returns.

Now things change and you update your forecasts for your three assets:

- A: 0.002
- B: 0.0025
- c: 0.001

If it didn't cost anything to trade, you would switch to asset B. But say it costs you 5 basis points (0.0005) to trade. In this scenario, your costs wipe out your additional expected returns. Suddenly, there's no benefit to switching, so you stay in asset A.

Our problem is now *"Given these expected returns and these existing positions, and these costs of trading, what are my optimal weights in the next period?*

Here's a function for framing and solving that problem with CVXR, assuming fixed percentage costs (retaining our no-leverage and long-only constraints):

```
maximise_alpha_with_costs <- function(expected_returns, current_weights, costs) {
  # define our weights vector as a CVXR::Variable
  weights <- Variable(rows = length(expected_returns))
  # define an alpha term as the weighted sum of expected returns
    # again, express using linear algebra
  alpha_term <- (t(weights) %*% expected_returns)
  # define a costs term. depends on:
    # cost of trading - needs to be expressed such that it scales with expected returns
      # calculate as elementwise cost * absolute value of weights - current_weights
      # use CVXR::multiply and CVXR::abs
    # absolute distance of current_weights to our weights variable
      # the more our target weights differ from current weights, the more it costs to trade
      # this is a decent representation of fixed percentage costs, but doesn't capture mini
    # sum_entries is a CVXR function for summing the elements of a vector
  costs_term <- sum_entries(multiply(costs, abs(weights - current_weights))) # elementwise
  # now our portfolio expected returns can be modelled as our alpha term less our costs
  # and our objective is to maximise this expression
  objective <- Maximize(alpha_term - costs_term)
  # we'll keep our no leverage constraint - sum of the abs.values of the weights must be le
  # also keep our long only constraint
  constraints <- list(cvxr_norm(weights, 1) <= 1, weights >= 0)
  # specify our problem
  problem <- Problem(objective, constraints)
  # solve it
  result <- solve(problem)

  # return the weights vector that CVXR found
  result$getValue(weights)
}
```

Now we set our expected returns for the next period, our current weights, and our costs.

We'll start off 100% invested in asset 6. But in the next period, we predict asset 5 to return 5 basis points more than asset 6. But we'll set our costs to 5 basis points, and thus expect the optimiser to return our existing positions (ie do no trading in the next period):

```r
# two assets with high expected returns, existing position in one.
# expect that adding transaction costs causes us not to switch out of existing position.
expected_returns <- c(0, 0, 0, 0, 0.0025, 0.002)
w0 <- c(0, 0, 0, 0, 0, 1)
costs <- 5/10000  # costs expressed roughly scaled to expected returns
expected_weights <- matrix(w0, ncol = 1)  # shouldn't switch

weights2 <- maximise_alpha_with_costs(expected_returns, w0, costs)
weights2

weights2 %>%
  as.data.frame() %>%
  ggplot(aes(x = factor(1:6), y = V1)) +
    geom_col() +
    labs(
      x = "asset",
      y = "weight",
      title = "Optimal weights"
    )

all.equal(weights2, expected_weights, tolerance = 1e-9)
```

A matrix: 6 × 1
of type dbl

2.791983e-15

0.000000e+00

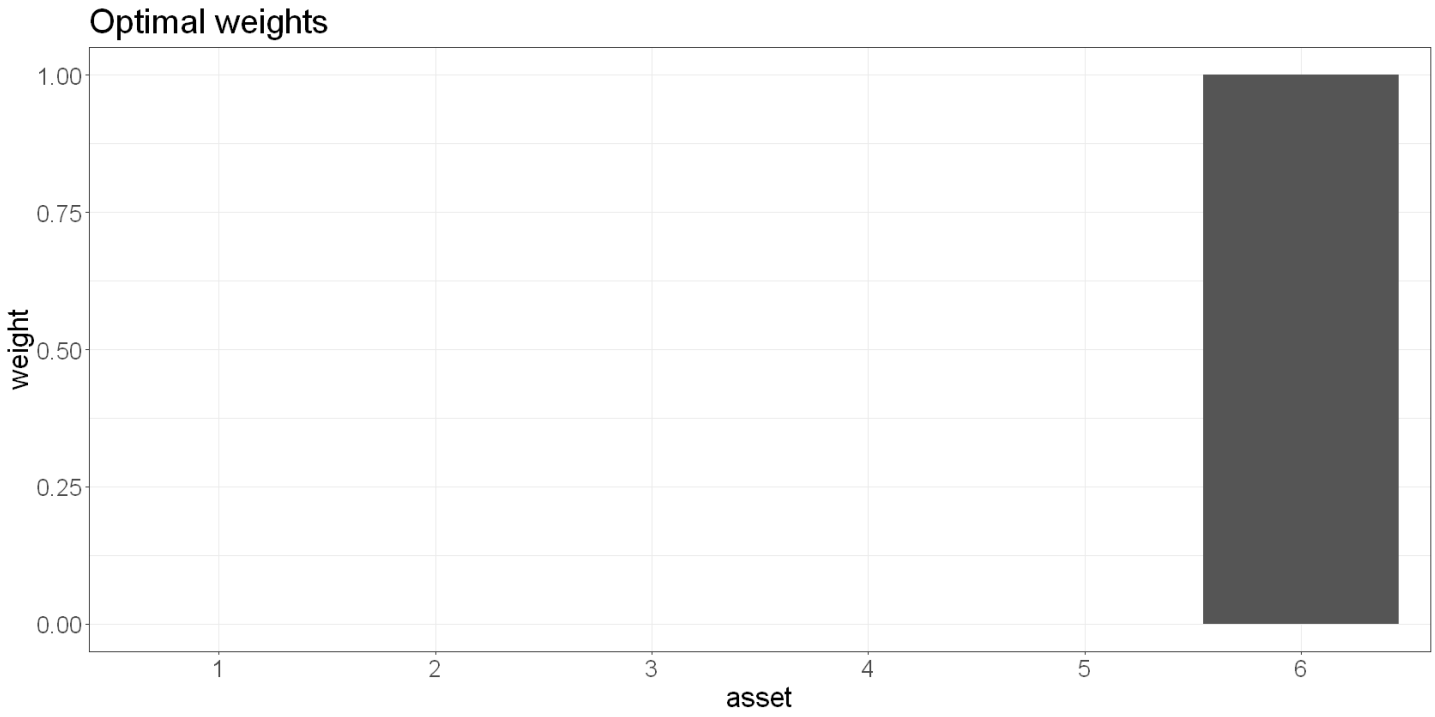
0.000000e+00

0.000000e+00

1.142232e-15

9.999862e-01

'Mean relative difference: 1.375717e-05'



Nice. Again we got the result we expected.

An important point here is the concept of “certain costs versus uncertain expected returns.”

I don’t mean “uncertain” in the probabilistic sense (realising the expected return in the long term despite random variation in the short term). I mean that there’s a chance we overestimate our expected returns - either from biases introduced in the modelling process, alpha decay, increasing market efficiency, or something else.

Since our costs are certain, but our edge is uncertain, we might want to artificially inflate the hurdle we need to overcome to trigger us to trade. We can do this by introducing a coefficient on the costs

term corresponding to our “propensity to trade.” More on this below.

Markowitz (mean-variance) optimisation without costs

Markowitz or mean-variance optimisation incorporates a penalty for risk. Typically we use a covariance matrix to model our risks - we want lower weights on things that tend to move together, because in the extreme it's like having two of the same thing in the portfolio (in this case, risk is narrowly defined as portfolio volatility).

Note that we use our *assumed* or *forecast* covariance matrix going forward - how we estimate this can be a significant source of error. Usually we shrink it towards long-term values or zeros to reflect uncertainty in our estimates.

Under this framework, we're essentially asking the question *"Which portfolio weights maximise my return for a given level of expected portfolio volatility, given these expected returns, and these asset co-movements?"*

We seek to maximise the weighted sum of portfolio returns less a penalty term for risk - which in this case we define as portfolio variance (how much our portfolio returns wiggle around). Portfolio variance is calculated as $w^T \Sigma w$ (the transpose of the weight vector multiplied by the covariance matrix multiplied by the weight vector, where "multiplied by" == the dot product).

We add a coefficient lambda (λ) which represents how much risk we'd like to take on. A higher lambda will put a bigger penalty on the risk term and change the weights accordingly.

So now we seek the weight vector w that maximises:

$$w^T R - \lambda w^T \Sigma w$$

Here's a function for framing and solving this problem with CVXR, retaining our no-leverage and long-only constraints:

```
mvo <- function(expected_returns, covmat, lambda = 1) {
  # define our weights vector as a CVXR::Variable
  weights <- Variable(length(expected_returns))
  # define an alpha term as the weighted sum of expected returns
    # again, express using linear algebra
  alpha_term <- (t(weights) %*% expected_returns)
  # define a risk term as w*Sigma*w
    # quad_form is a CVXR function for doing w*Sigma*w
  risk_term <- quad_form(weights, covmat)
  # define our object
    # maximise our alpha less our risk term multiplied by some factor, lambda
  objective <- Maximize(alpha_term - lambda*risk_term)
  # apply our no leverage and long only constraints
  constraints <- list(cvxr_norm(weights, 1) <= 1, weights >= 0)
  # specify the problem
  problem <- Problem(objective, constraints)
  # solve
  result <- solve(problem)

  # return the values of the variable we solved for
  result$getValue(weights)
}
```

We'll give all six assets the same expected return of 0.01.

And we'll give all our pairs of assets a covariance of zero (meaning that they're uncorrelated with one another), but we'll give assets 5 and 6 a slightly higher covariance so that they noisily move together.

Finally, we'll give all the assets the same variance (the term on the diagonal in the covariance matrix).

Under this scenario, we would expect assets 1-4 to get the same weight. But we would expect assets 5 and 6 to be slightly down-weighted because they're somewhat similar.

```
# 6 assets with equal expected returns
expected_returns <- rep(0.01, 6)

# give all assets except 5 and 6 a covariance of zero
# give assets 5 and 6 a larger covariance
# and set each asset's variance to the same value
covmat <- diag(6)
covmat[5, 6] <- covmat[6, 5] <- 0.5

expected_returns
covmat
```

0.01 · 0.01 · 0.01 · 0.01 · 0.01 · 0.01

A matrix: 6 × 6 of type dbl

1	0	0	0	0.0	0.0
0	1	0	0	0.0	0.0
0	0	1	0	0.0	0.0
0	0	0	1	0.0	0.0
0	0	0	0	1.0	0.5
0	0	0	0	0.5	1.0

```
# we expect to go equal weight into assets 1-4 and have a lower weight on assets 5 and 6 du
weights3 <- mvo(expected_returns, covmat, lambda = 0.01)

weights3

weights3 %>%
  as.data.frame() %>%
  ggplot(aes(x = factor(1:6), y = V1)) +
    geom_col() +
    labs(
      x = "asset",
      y = "weight",
      title = "Optimal weights"
    )
```

A matrix:
6 × 1 of
type dbl

0.1875

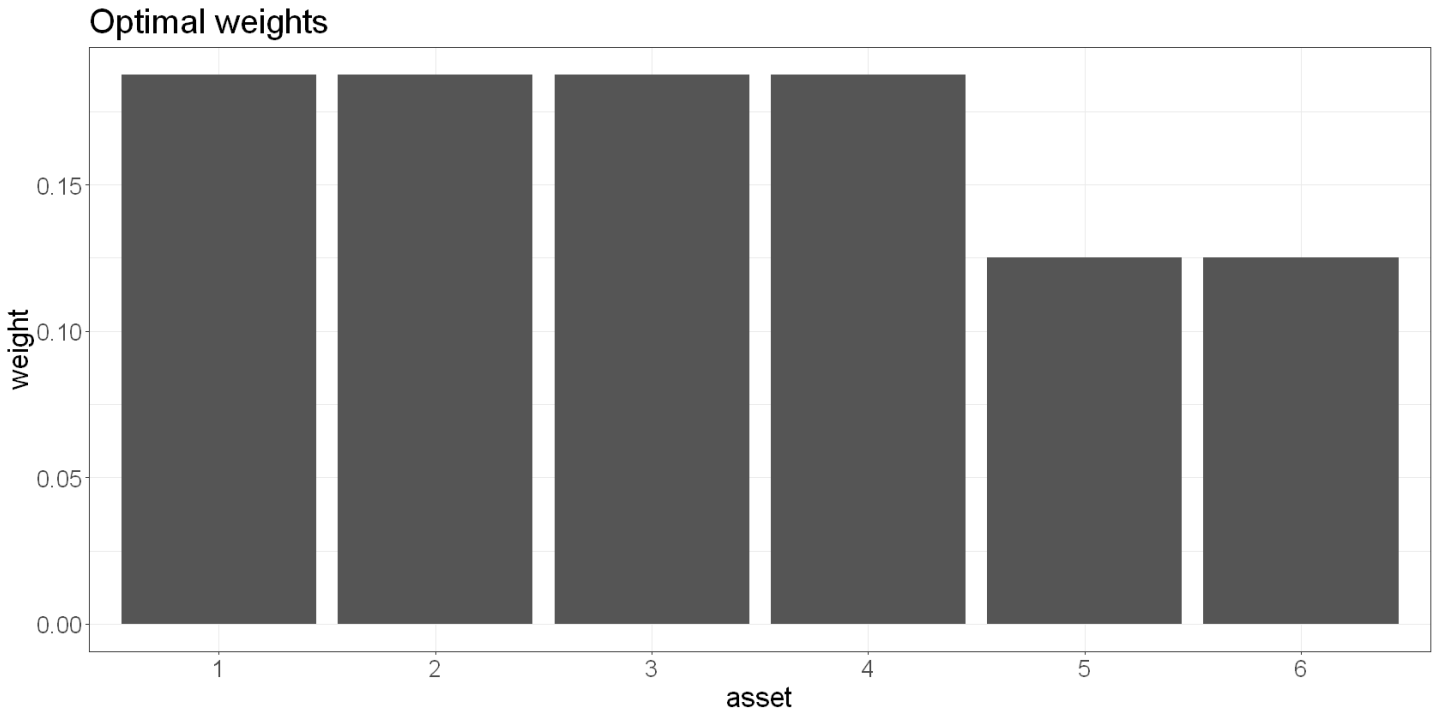
0.1875

0.1875

0.1875

0.1250

0.1250



Perfect. The optimiser correctly diversified the portfolio to minimise portfolio variance by taking into account the co-movement of assets 5 and 6, while maximising expected returns.

It's also illustrative to look at the effect of higher asset-wise variances.

We'll progressively increase the variance of assets 1 through 6 so that asset 1 has the lowest variance and asset 6 has the highest.

Since we're seeking to maximise expected returns and minimise risk, we would expect the optimiser to prefer the assets with the lower volatilities. That is, we'd expect asset 1 to get the highest weight

and asset 6 to get the lowest. Since assets 5 and 6 move noisily together, we'd expect these to be down-weighted more than the others.

```
# 6 assets with equal expected returns
expected_returns <- rep(0.01, 6)

# give all assets except 5 and 6 a covariance of zero
# give assets 5 and 6 a larger covariance
# and set each asset's variance to a larger value
covmat <- diag(6)*(seq(1.1, 1.6, by = 0.1))
covmat[5, 6] <- covmat[6, 5] <- 0.5

expected_returns
covmat
```

$0.01 \cdot 0.01 \cdot 0.01 \cdot 0.01 \cdot 0.01 \cdot 0.01$

A matrix: 6 × 6 of type dbl

1.1	0.0	0.0	0.0	0.0	0.0
0.0	1.2	0.0	0.0	0.0	0.0
0.0	0.0	1.3	0.0	0.0	0.0
0.0	0.0	0.0	1.4	0.0	0.0
0.0	0.0	0.0	0.0	1.5	0.5
0.0	0.0	0.0	0.0	0.5	1.6

```
# we expect to go equal weight into assets 1-4 and have a lower weight on assets 5 and 6 du
weights3 <- mvo(expected_returns, covmat, lambda = 0.01)

weights3

weights3 %>%
  as.data.frame() %>%
  ggplot(aes(x = factor(1:6), y = V1)) +
    geom_col() +
    labs(
      x = "asset",
      y = "weight",
      title = "Optimal weights"
    )
```

A matrix: 6 × 1 of type dbl

0.2163119

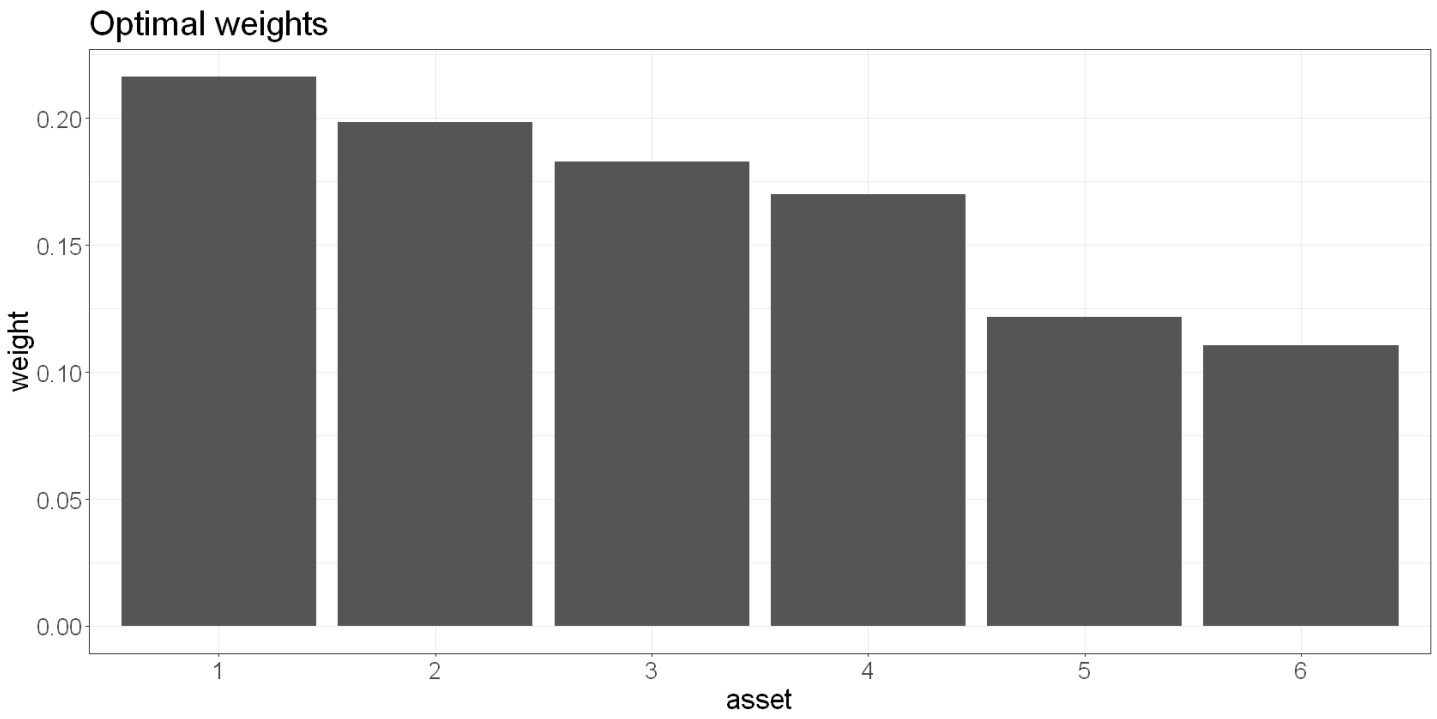
0.1982859

0.1830332

0.1699594

0.1217383

0.1106712



You can see how the optimiser now takes into account the increasing variances by up-weighting the lower-variance assets.

Explore the impact of the lambda parameter

Next, we'll use MVO and explore different values of lambda. This time, we'll include assets with negative expected returns, but force the portfolio to be fully invested and long only. Our covariance matrix will be random (but we need to force symmetry and positive variances on the diagonal to match the real world).

Remember that higher values of lambda will penalise assets that move together more. So:

- higher values of lambda should lead to more diversification due to the higher risk penalty.
- lower values of lambda should lead to more concentration as we penalise risk less and seek the asset with the highest expected returns more aggressively

```
set.seed(503)

# define returns and covariances
num_assets <- 6
expected_returns <- rnorm(num_assets, 0, 0.1)
S <- matrix(rnorm(num_assets^2, 0, 0.1), nrow = num_assets, ncol = num_assets)
S <- t(S) %*% S  # make symmetric and positive variances on diagonal

expected_returns
S
isSymmetric(S)
```

0.0355125069753125 · -0.077889340941631 · -0.0757356956529148 · 0.232980060737877 · 0.0283974925959303 · 0.128102521571192

A matrix: 6 × 6 of type dbl

0.0155231533	0.02012679	-0.0230124341	-0.0001158865	0.006316539	-0.0003914469
0.0201267875	0.06045292	-0.0132481650	-0.0192946840	0.010127719	-0.0185506516
-0.0230124341	-0.01324817	0.0721931391	0.0137559086	-0.019164521	0.0006967321
-0.0001158865	-0.01929468	0.0137559086	0.1004947030	-0.037670426	-0.0224346107
0.0063165389	0.01012772	-0.0191645212	-0.0376704259	0.025109511	0.0065522954
-0.0003914469	-0.01855065	0.0006967321	-0.0224346107	0.006552295	0.0397130317

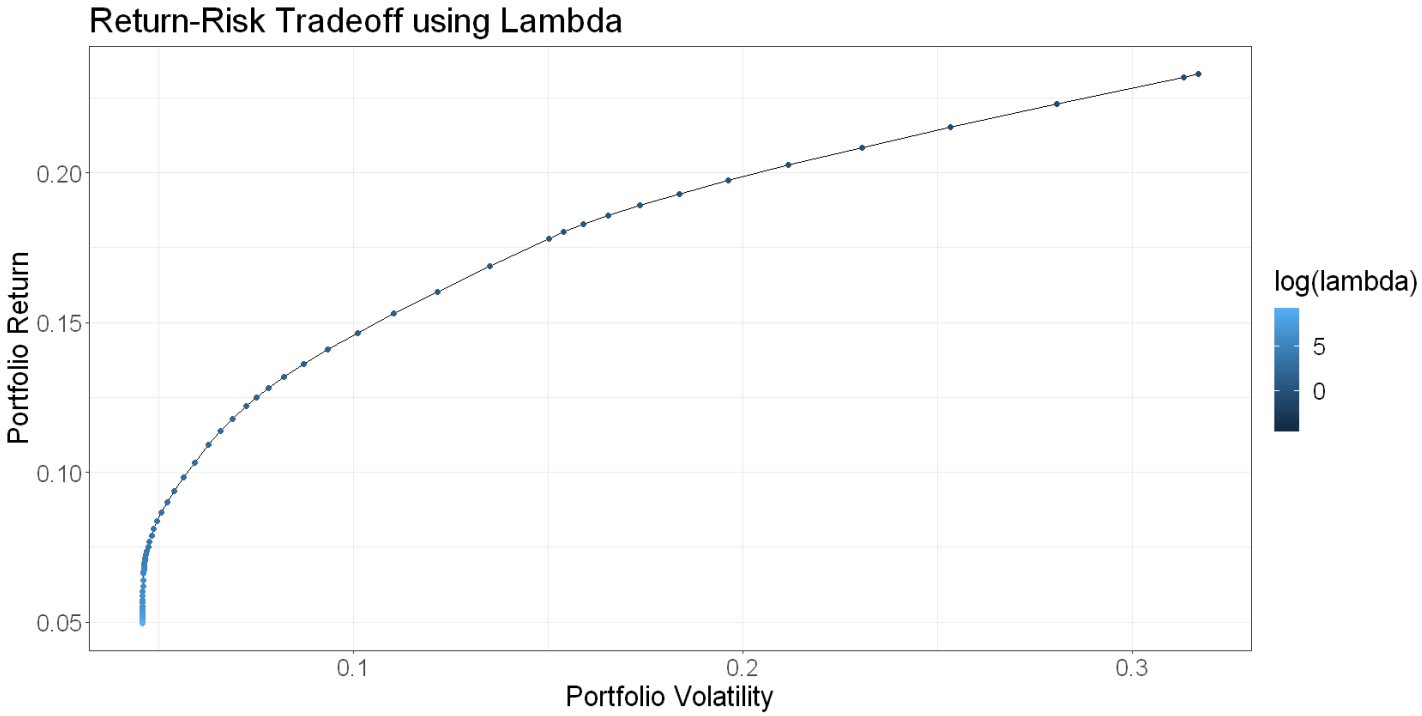
TRUE

We'll loop through different values of lambda, let the optimiser find our optimal weights, and calculate the resulting expected portfolio return and volatility. The end result will be a different set of weights and a different portfolio return and volatility for each value of lamda:

```
# define our optimisation problem inputs
weights <- Variable(num_assets)
portfolio_return <- t(expected_returns) %*% weights
portfolio_vol <- quad_form(weights, S)
constraints <- list(weights >= 0, sum(weights) == 1)  # long only...and force portfolio to

# lambdas
lambdas <- 10^seq(-2, 4, length.out = 100)

# allocate vectors for returns and volatilities
returns <- rep(0, length(lambdas))
vols <- rep(0, length(lambdas))
# allocate matrix for weight vectors
weight_vectors <- matrix(0, nrow = length(lambdas), ncol = num_assets)

# loop through lambdas and solve
for(i in seq_along(lambdas)) {
  lambda <- lambdas[i]
  obj <- Maximize(portfolio_return - lambda*portfolio_vol)
  problem <- Problem(obj, constraints)
  result <- solve(problem)

  # note can't get other variables back if we return `result` from our function, so do outs
  returns[i] <- result$getValue(portfolio_return)
  vols[i] <- result$getValue(sqrt(portfolio_vol))
  weight_vectors[i, ] <- result$getValue(weights)
}
```

Now we can plot the expected portfolio volatility and return for each value of lambda:

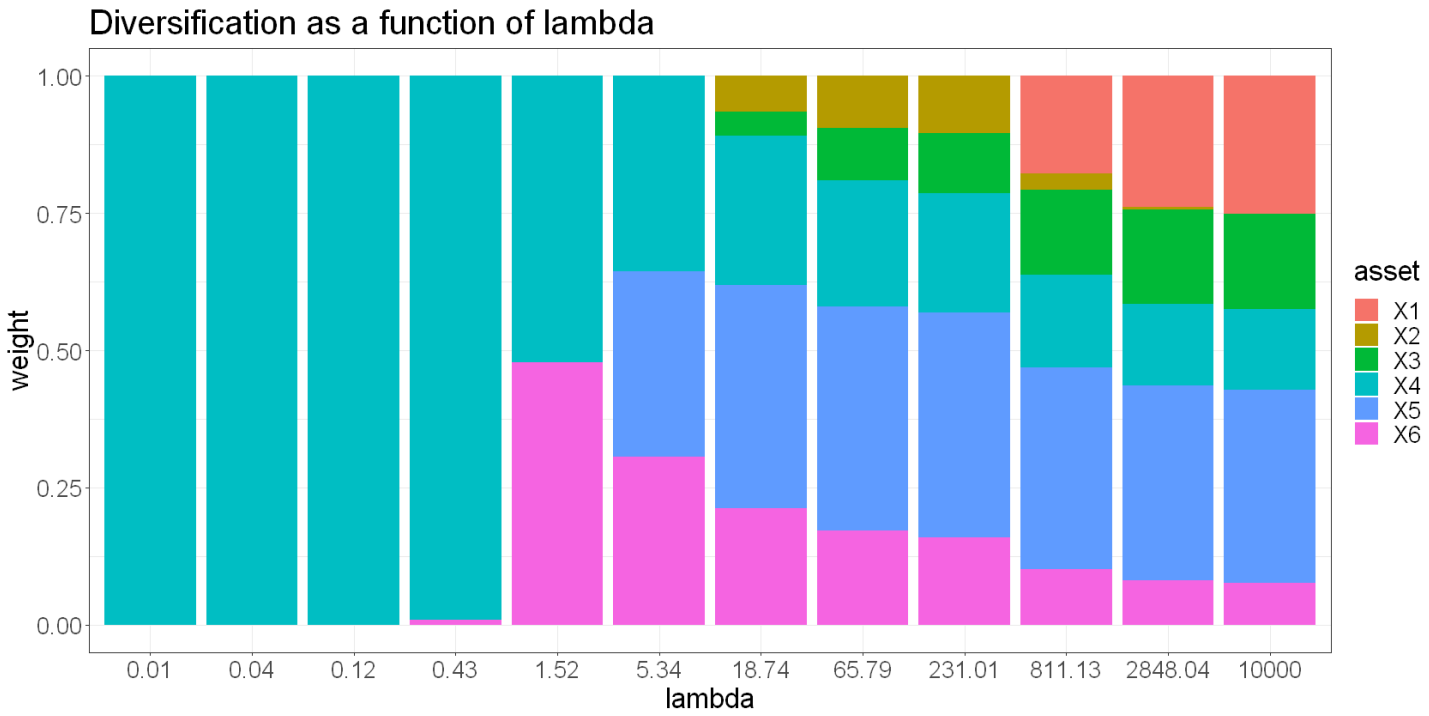
```
data.frame("portfolio_return" = returns, "portfolio_vol" = vols, "lambda" = lambdas) %>%
  ggplot(aes(x = portfolio_vol, y = portfolio_return)) +
  geom_line() +
  geom_point(aes(colour = log(lambda))) +
  labs(
    x = "Portfolio Volatility",
    y = "Portfolio Return",
    title = "Return-Risk Tradeoff using Lambda"
  )
```



You will recognise this as Markowitz's efficient frontier - we have found the weights that maximise return for a given volatility, or minimise volatility for a given return.

Each point on that frontier represents a different set of weights. Let's look at some of those weight vectors for different values of lambda:

```
weight_vecs <- data.frame(weight_vectors, lambdas) %>%
  pivot_longer(-lambdas, names_to = "asset", values_to = "weight") %>%
  rename("lambda" = lambdas)

weight_vecs %>%
  filter(lambda %in% lambdas[seq(from = 1, to = length(lambdas), by = 9)]) %>%
  ggplot(aes(x = factor(round(lambda, 2)), y = weight, fill = asset)) +
    geom_col(position = "stack") +
    labs(
      x = "lambda",
      y = "weight",
      title = "Diversification as a function of lambda"
    )
```



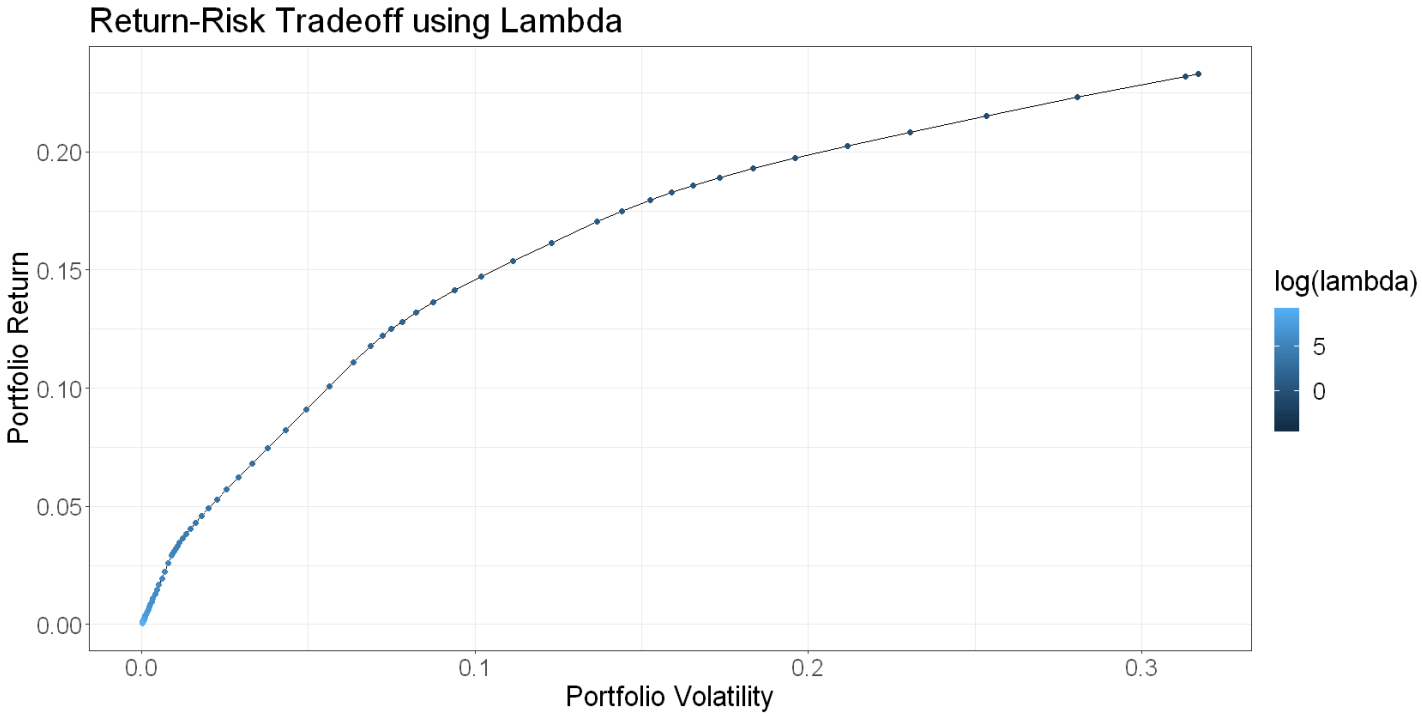
You can see how the lambda parameter impacts diversification:

- Low values seek concentration in the assets with the highest expected returns (asset 4)
- Higher values force diversification into other assets with lower expected returns (we trade expected returns for more diversification and lower volatility)
- There is a limit to diversification and thus a minimum portfolio volatility given our covariance matrix and the fact that we force the portfolio to be fully invested:
    - We see this in the convergence of portfolio weights towards the right of the bar plot above.
    - This corresponds to the left-most part of the efficient frontier where it goes vertical.

How about if we relax the long only and fully invested constraints?

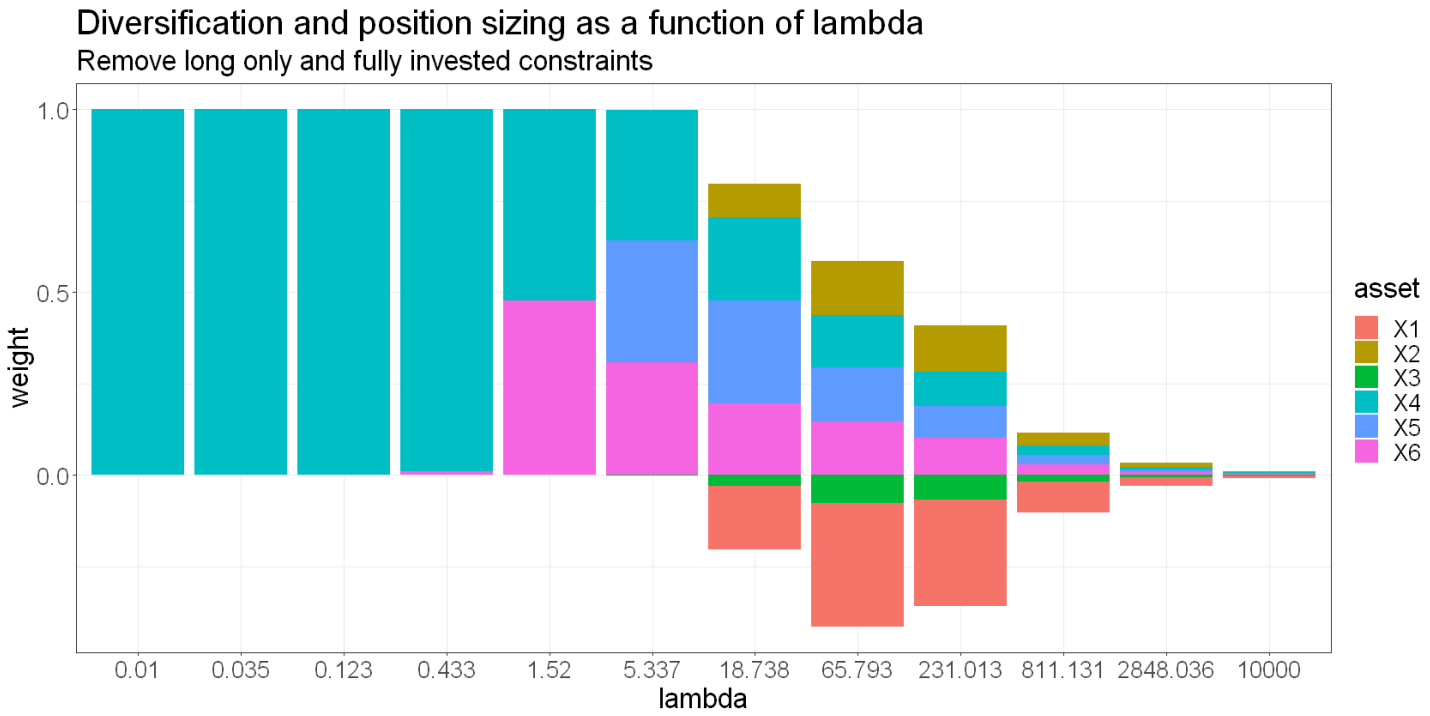
If we repeat the above, but without the long only and fully invested constraints, we should see the lambda parameter force us into more extreme diversification and some short positions. In the extreme, as we increase lambda, the optimiser should prefer smaller target weights as a way to reduce the portfolio volatility beyond what can be achieved through diversification alone:

```
# replace our long only and fully invested constraint with leverage less than 1. Also allow
constraints <- list(cvxr_norm(weights, 1) <= 1)

returns <- rep(0, length(lambdas))
vols <- rep(0, length(lambdas))
weight_vectors <- matrix(0, nrow = length(lambdas), ncol = num_assets)

for(i in seq_along(lambdas)) {
  lambda <- lambdas[i]
  obj <- Maximize(portfolio_return - lambda*portfolio_vol)
  problem <- Problem(obj, constraints)
  result <- solve(problem)

  returns[i] <- result$getValue(portfolio_return)
  vols[i] <- result$getValue(sqrt(portfolio_vol))
  weight_vectors[i, ] <- result$getValue(weights)
}

ggplot(
  data = data.frame("portfolio_return" = returns, "portfolio_vol" = vols, "lambda" = lambda
  aes(x = portfolio_vol, y = portfolio_return)
) +
  geom_line() +
  geom_point(aes(colour = log(lambda))) +
  labs(
    x = "Portfolio Volatility",
    y = "Portfolio Return",
    title = "Return-Risk Tradeoff using Lambda"
  )
```



We can see that under this different set of constraints, portfolio volatility can go all the way to zero. We would assume that that's because the optimiser is no longer forced to find a set of weights that has us fully invested, and that at some point it starts to reduce our position:

```
weight_vecs <- data.frame(weight_vectors, lambdas) %>%
  pivot_longer(-lambdas, names_to = "asset", values_to = "weight") %>%
  rename("lambda" = lambdas)

weight_vecs %>%
  filter(lambda %in% lambdas[seq(from = 1, to = length(lambdas), by = 9)]) %>%
  ggplot(aes(x = factor(round(lambda, 3)), y = weight, fill = asset)) +
    geom_col(position = "stack") +
    labs(
      x = "lambda",
      y = "weight",
      title = "Diversification and position sizing as a function of lambda",
      subtitle = "Remove long only and fully invested constraints"
    )
```



So you can start to see the interplay of the constraints, the alpha term, and the risk term (including the lambda coefficient).

- At low values of lambda, we simply target the asset with the highest expected returns (asset 4) with little to no regard for portfolio volatility.
- At a value of lambda of about 0.45, the diversification provided by a position in asset 6 is worth another reduction in expected returns.
- At a value of about 5, the additional diversification from asset 5 is worth another reduction in expected returns.
- At about 18, the additional diversification provided by a short position in asset 1 is worth another reduction in expected returns.
- At about 60, the optimser starts to reduce the overall position of the portfolio.
- From that point onwards, portfolio positions are reduced further and further, until in the extreme they're essentially zero.

MVO with costs

Our final example adds the cost term to the mean-variance optimisation problem.

We introduce a new parameter, tau, a coefficient on our costs term that controls our "propensity to trade" in a similar way to how lambda controls our risk aversion.

For simplicity, we'll include our no-leverage and long-only constraints.

Our question now becomes *"Given my existing positions, these expected returns, these forecast covariances, these costs to trade, my risk aversion and my propensity to trade, as well as my constraints (long only, no leverage), what is the best set of weights for the next period?"*

We can write this as an optimisation problem:

Find the weights, w that maximise:

$$w^T R - \lambda w^T \Sigma w - \tau c|w - w_0|$$

where w_0 is our initial weights and c is our cost term.

Here's how we frame that problem in CVXR:

```
mvo_with_costs <- function(expected_returns, current_weights, costs, covmat, lambda = 1, ta
  # define our weights vector as a CVXR::Variable
  weights <- Variable(length(expected_returns))
  # define an alpha term as the weighted sum of expected returns
    # again, express using linear algebra
  alpha_term <- (t(weights) %*% expected_returns)
  # define a costs term. depends on:
    # cost of trading - needs to be expressed such that it scales with expected returns
      # calculate as elementwise cost * absolute value of weights - current_weights
      # use CVXR::multiply and CVXR::abs
    # absolute distance of current_weights to our weights variable
      # the more our target weights differ from current weights, the more it costs to trade
      # this is a decent representation of fixed percentage costs, but doesn't capture mini
    # sum_entries is a CVXR function for summing the elements of a vector
  costs_term <- sum_entries(multiply(costs, abs(weights - current_weights))) # elementwise
  # define a risk term as w*Sigma*w
    # quad_form is a CVXR function for doing w*Sigma*w
  risk_term <- quad_form(weights, covmat)
  # define our objective
    # maximise our alpha less our risk term multiplied by some factor, lambda, less our cos
  objective <- Maximize(alpha_term - lambda*risk_term - tau*costs_term)
  # apply our no leverage and long only constraints
  constraints <- list(cvxr_norm(weights, 1) <= 1, weights >= 0)
  # specify the problem
  problem <- Problem(objective, constraints)
  # solve
  result <- solve(problem)

  # return the values of the variable we solved for
  result$getValue(weights)
}
```

We'll use our expected returns and covariance matrix from last time:

```
# use our expected returns and cov mat from last time
expected_returns
S
```

0.0355125069753125 · -0.077889340941631 · -0.0757356956529148 · 0.232980060737877 · 0.0283974925959303 · 0.128102521571192

A matrix: 6 × 6 of type dbl

0.0155231533	0.02012679	-0.0230124341	-0.0001158865	0.006316539	-0.0003914469
0.0201267875	0.06045292	-0.0132481650	-0.0192946840	0.010127719	-0.0185506516
-0.0230124341	-0.01324817	0.0721931391	0.0137559086	-0.019164521	0.0006967321
-0.0001158865	-0.01929468	0.0137559086	0.1004947030	-0.037670426	-0.0224346107
0.0063165389	0.01012772	-0.0191645212	-0.0376704259	0.025109511	0.0065522954
-0.0003914469	-0.01855065	0.0006967321	-0.0224346107	0.006552295	0.0397130317

Given a tau of zero (meaning we disgregard costs altogether) and a lambda of 15, we get the following weights for the next period:

```
# define our existing positions - we're in the asset with the second highest expected retur
w0 <- c(0, 0, 0, 0, 0, 1)
costs <- 5/10000  # costs expressed roughly scaled to expected returns

# set lambda and tau
lambda <- 15
tau <- 0

weights4 <- mvo_with_costs(expected_returns, w0, costs, S, lambda = lambda, tau = tau)
weights4

weights4 %>%
  as.data.frame() %>%
  ggplot(aes(x = factor(1:6), y = V1)) +
    geom_col() +
    labs(
      x = "asset",
      y = "weight",
      title = "Optimal weights"
    )
```

A matrix: 6 × 1
of type dbl

-6.530549e-07

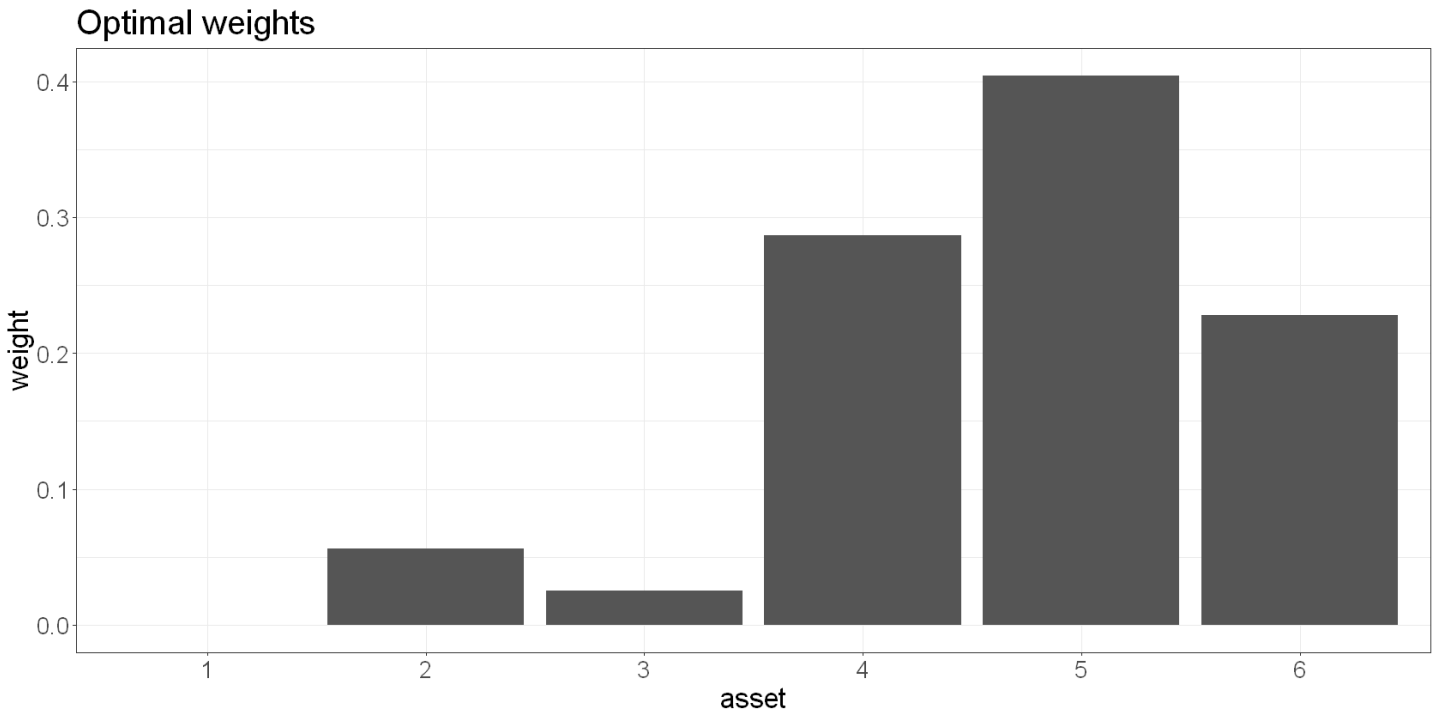
5.589699e-02

2.541301e-02

2.866296e-01

4.041310e-01

2.278091e-01

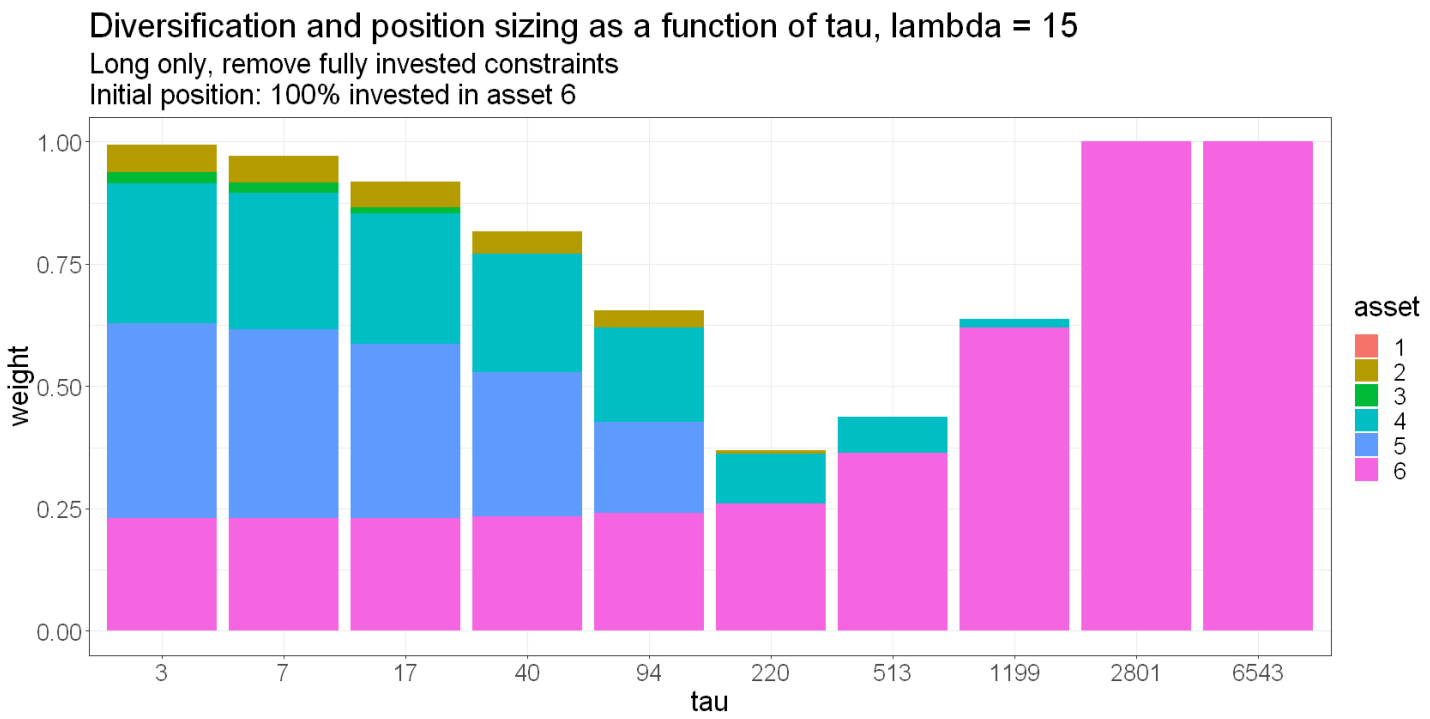


Let’s see what happens as we increase tau. We’ll hold lambda constant at 15.

```
# taus
taus <- 10^seq(0.5, 4, length.out = 20)

# allocate vectors for returns and volatilities
returns <- rep(0, length(lambdas))
vols <- rep(0, length(lambdas))
# allocate matrix for weight vectors
weight_vectors <- matrix(0, nrow = length(taus), ncol = num_assets)

# loop through lambdas and solve
for(i in seq_along(taus)) {
  tau <- taus[i]
  weight_vectors[i, ] <- mvo_with_costs(expected_returns, w0, costs, S, lambda = lambda, ta
}
```

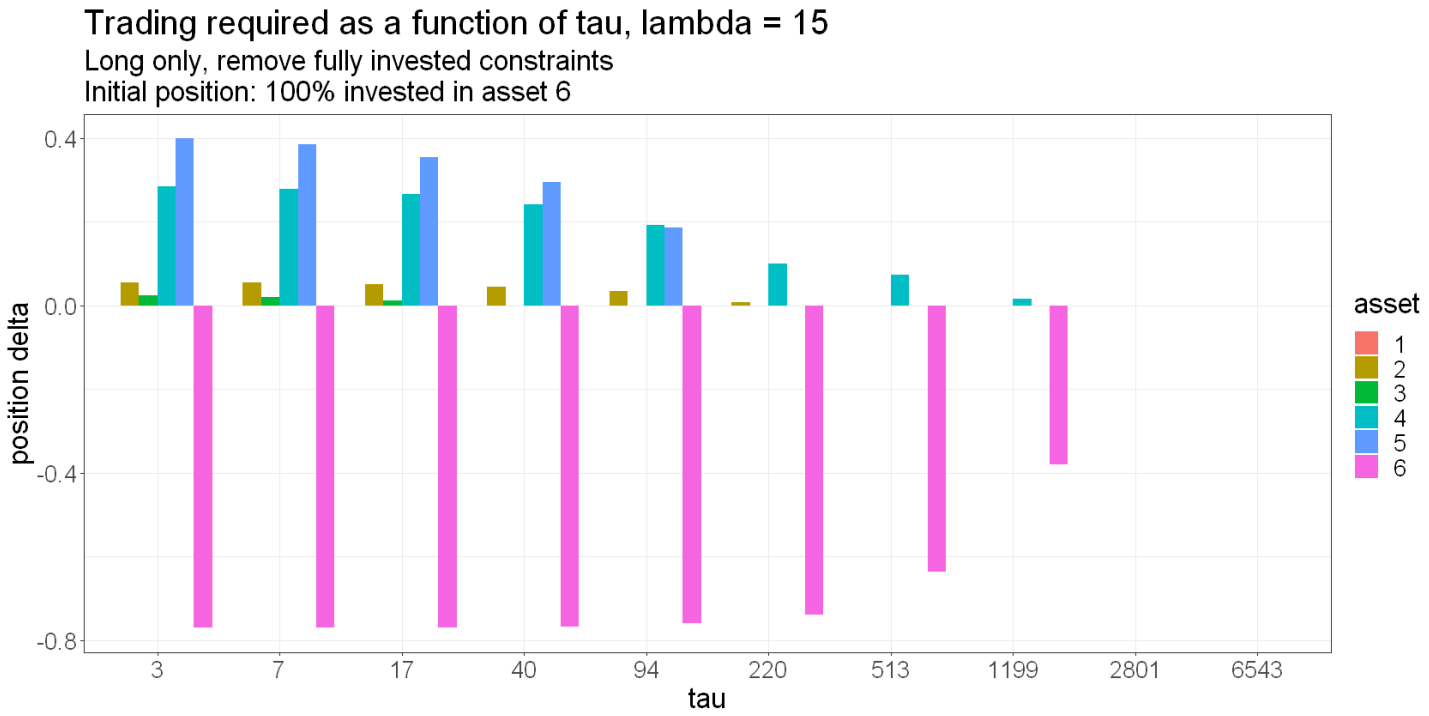
Here's a plot of our next-period weights for different values of tau:

```
weight_vecs <- data.frame(weight_vectors, taus) %>%
  rename_with(~gsub("X", "", .x)) %>%
  pivot_longer(-taus, names_to = "asset", values_to = "weight") %>%
  rename("tau" = taus) %>%
  left_join(
    data.frame(asset = as.character(c(1:6)), starting_weight = w0),
    by = "asset"
  )%>%
  mutate(pos_delta = weight - starting_weight)

weight_vecs %>%
  filter(tau %in% taus[seq(from = 1, to = length(taus), by = 2)]) %>%
  ggplot(aes(x = factor(round(tau, 0)), y = weight, fill = asset)) +
    geom_col(position = "stack") +
    labs(
      x = "tau",
      y = "weight",
      title = glue::glue("Diversification and position sizing as a function of tau, lambda
      subtitle = "Long only, remove fully invested constraints\nInitial position: 100% inve
    )
```



This plot shows the actual trading required for each asset as a proportion of invested capital (negative values imply selling an asset):

```
weight_vecs %>%
filter(tau %in% taus[seq(from = 1, to = length(taus), by = 2)]) %>%
ggplot(aes(x = factor(round(tau, 0)), y = pos_delta, fill = asset)) +
  geom_col(position = "dodge") +
  labs(
    x = "tau",
    y = "position delta",
    title = glue::glue("Trading required as a function of tau, lambda = {lambda}"),
    subtitle = "Long only, remove fully invested constraints\nInitial position: 100% inve
  )
```



You can see that for smaller values of tau, we pay less attention to costs and make more and bigger trades. We sell a big chunk of asset 6 and buy varying amounts of the other assets.

As we increase tau, our target weights start to differ less from our starting weights, until we do no trading at all and our next position is equal to our current position.

You can see how, under this framework, the behaviour of your strategy is controlled by essentially two levers: lambda and tau. You would find appropriate values for these parameters through simulation, similarly to how we found an appropriate value for our no-trade buffer in previous articles. We'll do that in the next article.

Some tips and tricks

Convex optimisation in context

It's important to understand that under this framework, estimating our expected returns and covariances is the whole game. The optimiser will always find the best weights given your expected returns and covariances, but it has no control over the *quality* of those estimates. That's up to you.

Remember, convex optimisation isn't a source of alpha. It's just a tool to help you implement some trading rules to harness your alphas effectively.

CVXR tips and tricks

Here are some general tips and tricks to be aware of when setting up optimisation problems with CVXR:

- <= and >= problems can generally be solved by CVXR
- but > and < problems may not be strictly convex (and hence CVXR won't converge)
- if you get errors, it's often because you didn't specify a linear algebra problem correctly (check you've transposed the correct vector or matrix and used the correct operator)
- look through the lst of CVXR functions - it is incredibly rich
- CVXR will fall over if you covariance matrix isn't symmetrical and positive semi definite, sometimes even if it is out just a touch, so it's wise to put a check on your covariance matrix before proceeding with the optimisation.

Bonus: constrain historical downside risk

I got this idea from Scott Sanderson back in the Quantopian days.

In what we've seen so far, our weights are dependent on expected return and covariance estimates (as well as constraints, costs, etc).

Estimating covariances is, unfortunately, famously difficult. And the weights our optimiser returns will be very sensitive to the covariance values you input (as an exercise, experiment by changing one or more values in the covariance matrixes of previous examples and get a feel for this dependency).

Typically in practice, you'll end up doing some sort of shrinking procedure to reduce your covariance estimates to either zero or their long-run averages.

Another interesting approach that gets around this is to not express a view on covariances at all. Instead, we can model risk as the maximum daily downside that would have resulted from the target weights given the actual historical returns of the assets in our portfolio.

This comes with another set of considerations - for example, it reflects just one path through reality, and we all know that there's no guarantee that the future will be like the past. It's entirely feasible (likely, perhaps) that the future will conspire to produce a set of circumstances where the target weights lead to a greater loss than we saw in the past. But at least it gives us something tangible that we can base our assessment of risk on.

Let me show you how this works.

First, we'll create some historical returns. We'll get 1,000 days of returns for the six assets we're trading:

```
set.seed(503)

num_days <- 1000
num_assets <- 6
historical_returns <- matrix(rnorm(num_days*num_assets, 0, 0.1), nrow = num_days, ncol = nu

head(historical_returns)
```

A matrix: 6 × 6 of type dbl

0.03551251	0.01728147	0.10600270	0.1298158	-0.07601763	0.14176114
-0.07788934	0.04335477	-0.05217292	0.1907053	0.02713085	-0.05079215
-0.07573570	0.02067132	-0.02152929	-0.0940365	-0.25205753	-0.01763523
0.23298006	-0.03187455	-0.12913116	-0.1180375	0.02760712	0.11984626
0.02839749	-0.04619874	-0.11878668	-0.1272778	0.06466424	-0.10790562
0.12810252	0.15281995	0.17616053	0.1354930	0.02578447	-0.06109857

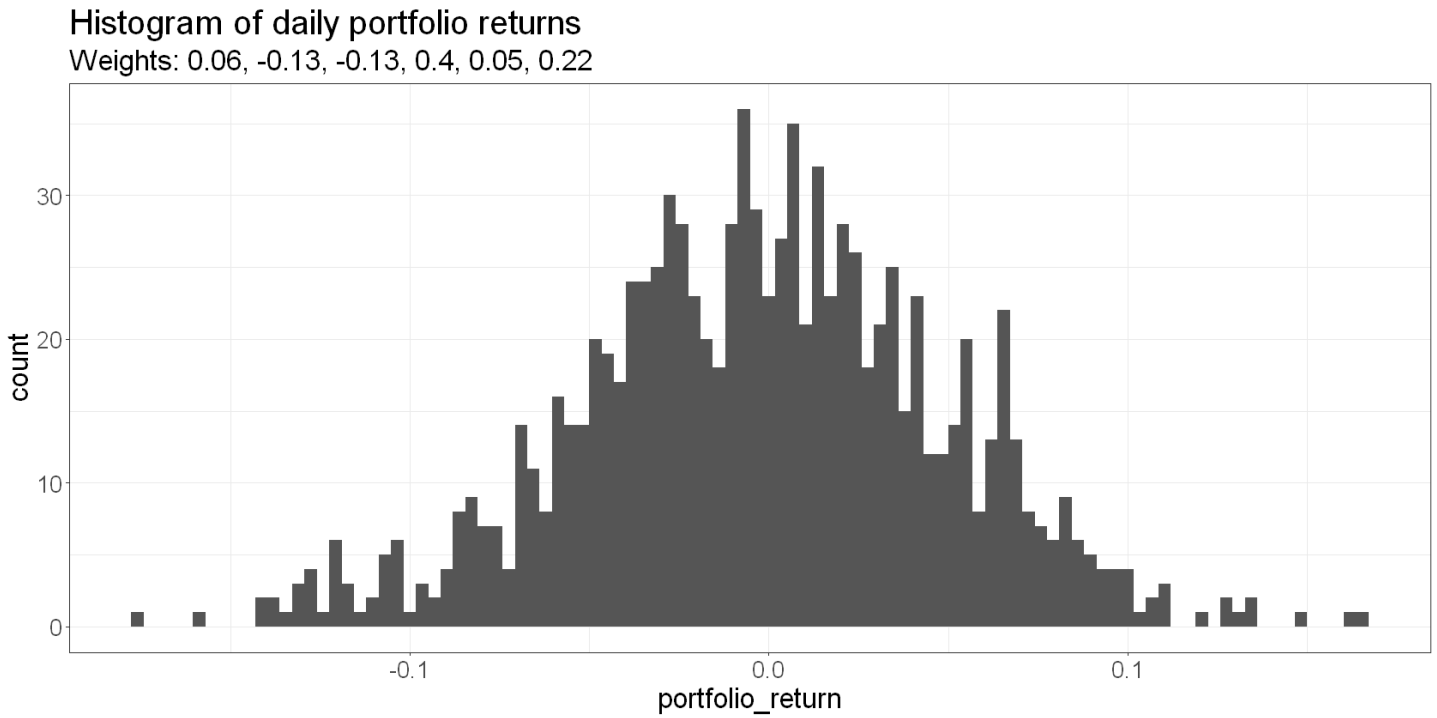
Let's create a random weight vector for the sake of the example:

```
set.seed(503)

num_assets <- 6
w0 <- rnorm(num_assets, 0, 1)
# scale so that their abs.values sum to
w0 <- w0/sum(abs(w0))
w0
```

0.0613747418699724 · -0.134612805512101 · -0.13089075277822 · 0.402649441183583 · 0.0490781678421434 · 0.221394090813981

Now, let's look at the historical daily returns to a portfolio of these six assets at these weights. The dot product of our historical asset-wise returns matrix and our weight vector will give us our daily portfolio returns:

```
historical_port_returns <- historical_returns %*% w0

historical_port_returns %>%
  data.frame(portfolio_return = ., index = c(1:nrow(historical_returns))) %>%
  ggplot(aes(x = portfolio_return)) +
  geom_histogram(bins=100) +
  labs(
    title = "Histogram of daily portfolio returns",
    subtitle = glue::glue("Weights: {glue::glue_collapse(round(w0, 2), sep = ', ')}")
  )
```



You can see that we had several days in our 1,000-day sample in which we lost more than 10%.

Next, let's figure out the 5th percentile of our daily returns. That is, what's the cutoff for our worst 5% of days?

```
# get the bottom
quantile(historical_port_returns, 0.05)
```

5%: -0.0862743958212262

We can use that to firstly extract the bottom 5% of days:

```
worst_days <- historical_port_returns[historical_port_returns <= quantile(historical_port_r
head(worst_days)
```

-0.140243608302904 · -0.0875556659433104 · -0.0898470072466509 · -0.100539995047078 · -0.126077141570876 · -0.0962782588902194

And now we can calculate the mean of our worst days:

```
mean(worst_days)
```

-0.114550160414768

Now let's say we decide to exclude any weights that resulted in a mean worst day loss of 10% or more in the historical data.

That would mean that the weights we chose here would get passed over by the optimiser, because the mean of their worst days was slightly worse than -10%.

And that's the whole idea of this approach in a nutshell:

- Pick a maximum historical average worst day cutoff (say -10%)
- Define your set of "worst days" - here we simply grabbed the bottom 5% of daily returns
- Define a constraint such that the optimiser excludes weights whose average worst day in the past exceeded your cutoff value

A logical question would be *"Why complicate things by using the average of your worst days? Why not simply pick a single maximum daily loss and use that instead?"*

That would introduce a lot of sensitivity and randomness. For example, an otherwise optimal set of weights might have resulted in a single outlying bad return that would exclude it from the solution space. You might not want to exclude a set of weights due to a single bad day - you're essentially letting randomness drive the results to a very large extent.

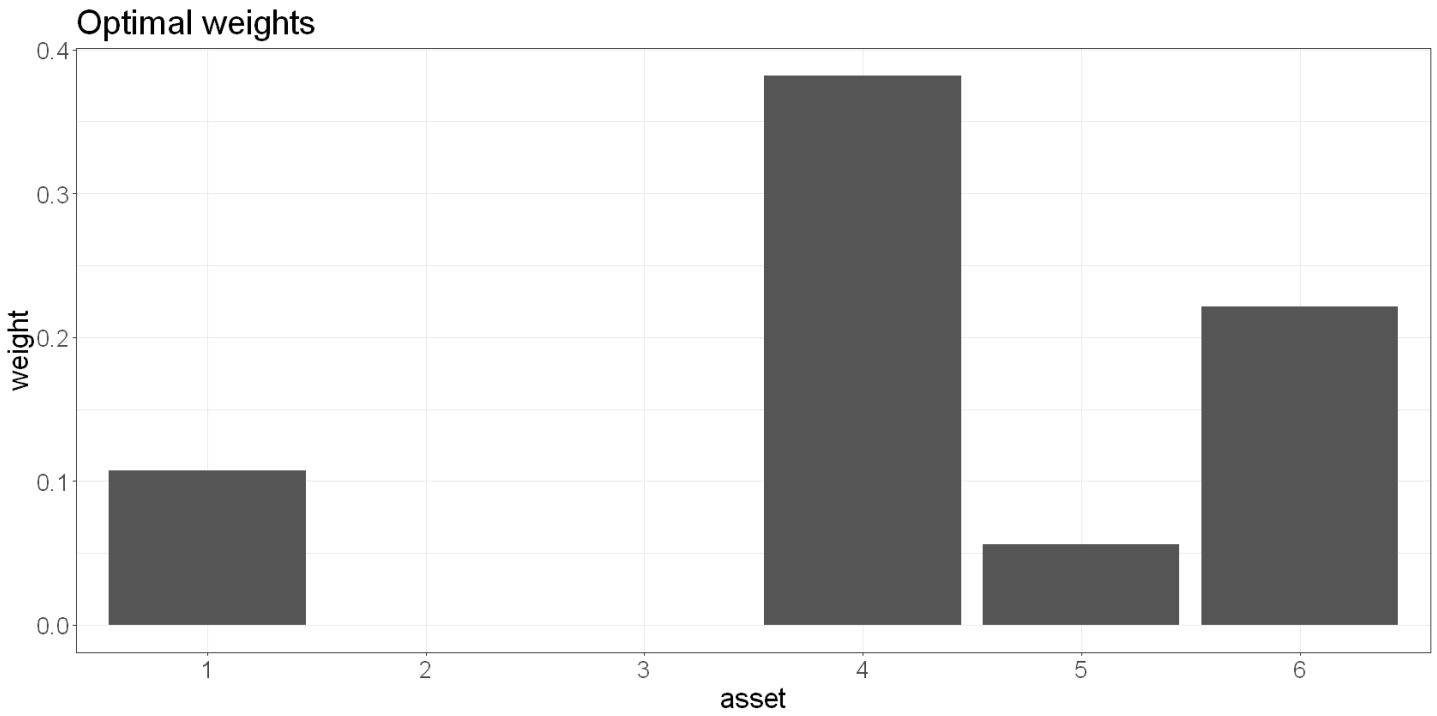
Constraining our cutoff to a mean of the bottom $x\%$ of bad days somewhat reduces this randomness.

Now let's implement this approach using CVXR:

```
constrain_downside <- function(expected_returns, current_weights, historical_returns, max_l
  # define our weights vector as a CVXR::Variable
  weights <- Variable(length(expected_returns))
  # define an alpha term as the weighted sum of expected returns
    # again, express using linear algebra
  alpha_term <- (t(weights) %*% expected_returns)
  # define a costs term. depends on:
    # cost of trading - needs to be expressed such that it scales with expected returns
      # calculate as elementwise cost * absolute value of weights - current_weights
      # use CVXR::multiply and CVXR::abs
    # absolute distance of current_weights to our weights variable
      # the more our target weights differ from current weights, the more it costs to trade
      # this is a decent representation of fixed percentage costs, but doesn't capture mini
    # sum_entries is a CVXR function for summing the elements of a vector
  costs_term <- sum_entries(multiply(costs, abs(weights - current_weights))) # elementwise
  # define our objective
    # maximise our alpha less our costs term multiplied by tau
  objective <- Maximize(alpha_term)  #  - tau*costs_term
  # define a risk constraint as the average of our bottom 5% of bad days
    # you might want to pick of different value for k, or pass it as a parameter to the fun
    k = round(nrow(historical_returns) * 0.05, 0)
    historical_port_returns = historical_returns %*% weights
    # sum_smallest sums the smallest k values
    ave_bad_day = CVXR::sum_smallest(historical_port_returns, k)/k
  # apply our no leverage and long only constraints
  constraints <- list(cvxr_norm(weights, 1) <= 1, weights >= 0, ave_bad_day >= max_loss)
  # specify the problem
  problem <- Problem(objective, constraints)
  # solve
  result <- solve(problem, solver = "ECOS")

  # return the values of the variable we solved for
  result$getValue(weights)
}
```

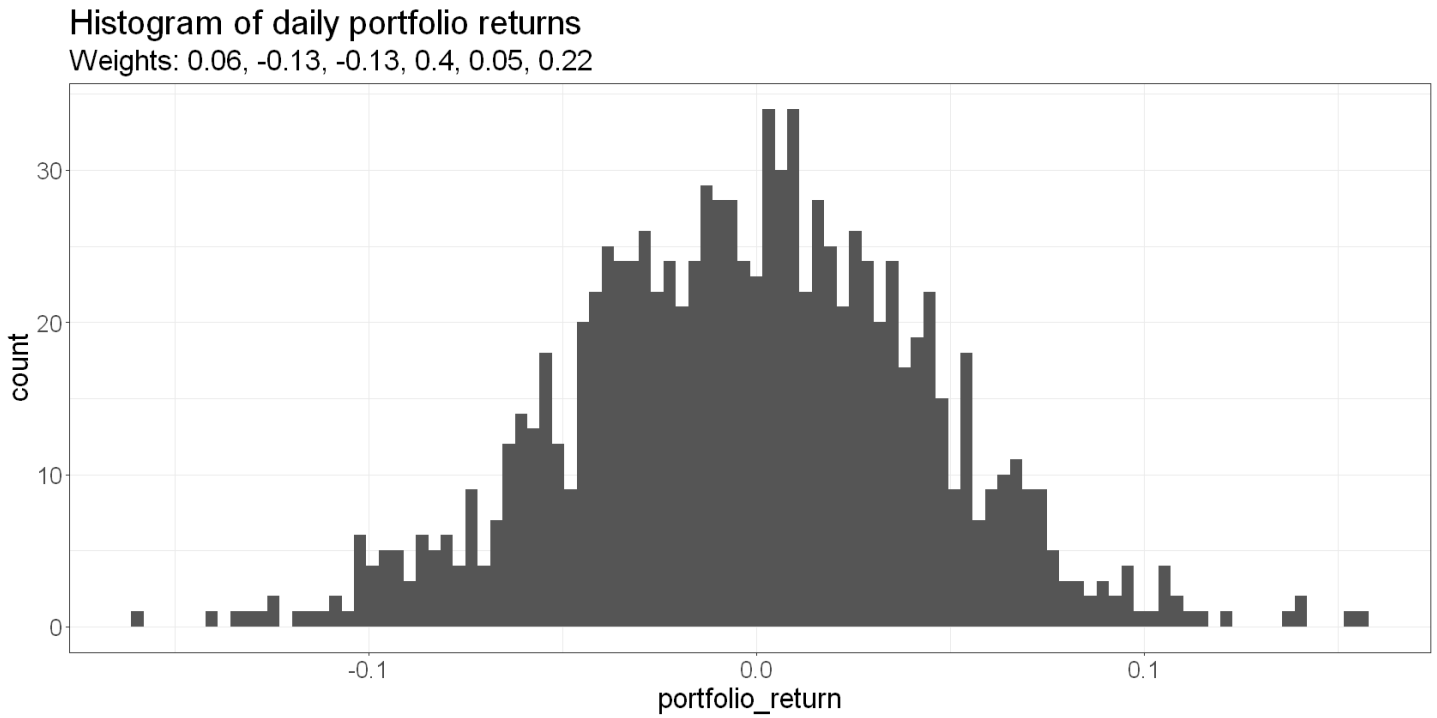
We'll set our maximum average worst day to -10% and apply a tau of 20 to our costs parameter:

```
w <- constrain_downside(expected_returns = expected_returns, current_weights = w0, historic

# plot optimal weights
w %>%
  as.data.frame() %>%
  ggplot(aes(x = factor(1:num_assets), y = V1)) +
    geom_col() +
    labs(
      x = "asset",
      y = "weight",
      title = "Optimal weights"
    )
```



Let's have a look at the historical returns given these weights:

```
# historical daily portfolio returns at these weights
historical_port_returns <- historical_returns %*% w

historical_port_returns %>%
  data.frame(portfolio_return = ., index = c(1:nrow(historical_returns))) %>%
  ggplot(aes(x = portfolio_return)) +
  geom_histogram(bins=100) +
  labs(
    title = "Histogram of daily portfolio returns",
    subtitle = glue::glue("Weights: {glue::glue_collapse(round(w0, 2), sep = ', ')}")
  )
```




```
# what's the mean of our bottom 5% of bad days?
worst_days <- historical_port_returns[historical_port_returns <= quantile(historical_port_r
mean(worst_days)
```

-0.0999999999993967

You can see that the average of the worst days given these weights did indeed come in just north of our cutoff of -10%.

Conclusion

The purpose of this chapter was to build your intuition for convex optimisation. Specifcally, how will certain constraints and objectives influence the optimiser's decisions?

We also saw several examples of constructing different optimisation problems using the CVXR library, which provides a high-level convex optimisation modelling language with user-friendly syntax.

In particular, we saw that we can construct the trading problem as a mean-variance optimisation that includes costs. Under this framework, we can essentially control the strategy's behaviour using two parameters: our risk version (lambda) and our propensity to trade (tau). In the next article, we'll see how we can use simulation to pick appropriate values for these parameters.

Given the difficulties around covariance estimates and the sensitivity to small changes in these values, we also saw an approach that defines risk in terms of historical losses rather than covariances and portfolio volatility. The main drawback with this approach is that the future will likely not look exactly like the past. But maybe it will look enough like the past that this approach is feasible.

Finally, we underline the point that convex optimisation is not a source of alpha itself. It's simply a tool to help you navigate the various operational trade-offs involved with harnessing your edges, and is entirely dependent on the quality of your forecasts.

Navigating Tradeoffs with Convex Optimisation

So far in this book, we have:

- Introduced a general approach for doing stat arb
- Brainstormed some ideas for crypto stat arb alphas
- Explored how we might quantify and combine those alphas
- Introduced the no-trade buffer: a heuristic approach to navigating the tradeoff between uncertain alpha and certain costs
- Described how to model features as expected returns - a prerequisite for using an optimisation-based approach
- Provided a ton of examples of different optimisation problems and how they work in CVXR

In the previous article, we built some intuition for what an optimiser does under the hood. In this article, we'll use the optimiser to navigate the tradeoffs between costs, alpha and risk. Essentially we're swapping out the simpler heuristic approach from earlier in the series for an optimisation-based approach.

There are advantages and disadvantages to doing this:

- The heuristic approach is simple and intuitive and leads to a predictable, mechanical set of rules for managing your positions. The optimiser is a bit more black box.
- The optimisation approach accommodates real-world constraints directly.
- It also enables incorporation of risk models such as covariance estimates, Value-at-Risk, etc. In the heuristic approach, you'd have to incorporate this upstream of your rebalancing somehow.
- The optimisation-based approach is very flexible and scalable: you can add new signals or risk estimates without re-fitting anything.

You certainly don't *need* to add the complexity of an optimisation-based approach. It's not a pre-requisite to making money. But if you have the resources to implement it (time, skills, etc) it can boost performance and allow you to scale fairly quickly by adding new signals or risk models as you develop them.

In this final chapter, I'll show you how to simulate trading with convex optimisation.

First, load our libraries and the dataset we've been using throughout the series:

```
# session  options
options(repr.plot.width = 14, repr.plot.height=7, warn = -1)

library(tidyverse)
library(tidyfit)
library(glue)
library(CVXR)
library(tibbletime)
library(roll)
library(patchwork)
pacman::p_load_current_gh("Robot-Wealth/rsims", dependencies = TRUE)

# chart options
theme_set(theme_bw())
theme_update(text = element_text(size = 20))

# data
perps <- read_csv("https://github.com/Robot-Wealth/r-quant-recipes/raw/master/quantifying-c
head(perps)
```

```
Rows: 187251 Columns: 11
── Column specification ────────────────────────────────────────────────────────
Delimiter: ","
chr  (1): ticker
dbl  (9): open, high, low, close, dollar_volume, num_trades, taker_buy_volum...
date (1): date

ℹ Use `spec()` to retrieve the full column specification for this data.
ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

A tibble: 6 × 11

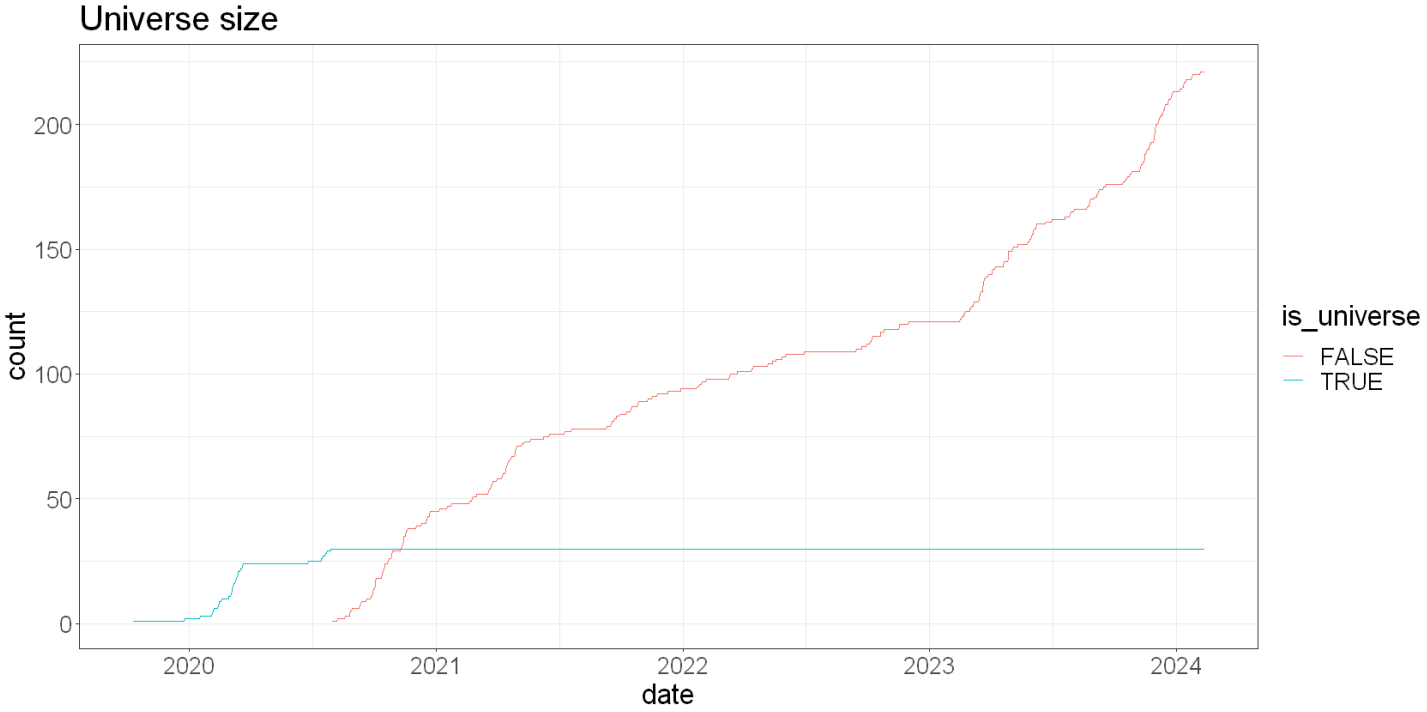
ticker	date	open	high	low	close	dollar_volume	num_trades	ta
<chr>	<date>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	
BTCUSDT	2019-09-11	10172.13	10293.11	9884.31	9991.84	85955369	10928	
BTCUSDT	2019-09-12	9992.18	10365.15	9934.11	10326.58	157223498	19384	
BTCUSDT	2019-09-13	10327.25	10450.13	10239.42	10296.57	189055129	25370	
BTCUSDT	2019-09-14	10294.81	10396.40	10153.51	10358.00	206031349	31494	
BTCUSDT	2019-09-15	10355.61	10419.97	10024.81	10306.37	211326874	27512	
BTCUSDT	2019-09-16	10306.79	10353.81	10115.00	10120.07	208211376	29030	

For the purposes of this example, we'll create the same crypto universe that we used last time - the top 30 Binance perpetual futures contracts by trailing 30-day dollar-volume, with stables and wrapped tokens removed.

We'll also calculate returns at this step for later use.

```r
# get same universe as before - top 30 by rolling 30-day dollar volume, no stables

# remove stablecoins
# list of stablecoins from defi llama
url <- "https://stablecoins.llama.fi/stablecoins?includePrices=true"
response <- httr::GET(url)

stables <- response %>%
  httr::content(as = "text", encoding = "UTF-8") %>%
  jsonlite::fromJSON(flatten = TRUE) %>%
  pluck("peggedAssets") %>%
  pull(symbol)

# sort(stables)

perps <- perps %>%
  filter(!ticker %in% glue::glue("{stables}USDT"))
```

```
# get the top 30 by trailing 30-day volume
# exclude from universe until we have enough data to do a covariance estimate (see later)
trading_universe_size <- 30
ew_cov_init_wdw <- 30

universe <- perps %>%
  group_by(ticker) %>%
  # also calculate returns for later
  mutate(
    total_return_simple = funding_rate + (close - lag(close, 1))/lag(close, 1),
    total_return_log = log(1 + total_return_simple),
    total_fwd_return_simple = dplyr::lead(funding_rate, 1) + (dplyr::lead(close, 1) - close
    total_fwd_return_log = log(1 + total_fwd_return_simple),
    trail_volume = roll_mean(dollar_volume, 30)
  ) %>%
  mutate(days_since_listing = row_number()) %>%   # the number of days we have data for eac
  na.omit() %>%
  ungroup() %>%
  # exclude from volume ranking - won't matter if we use a rolling volume calculated with m
  mutate(to_exclude = days_since_listing < ew_cov_init_wdw) %>%
  group_by(date) %>%
  mutate(
    volume_rank = if_else(to_exclude, NA_integer_, row_number(-trail_volume)),
    is_universe = volume_rank <= trading_universe_size,
  )

universe %>%
  group_by(date, is_universe) %>%
  summarize(count = n(), .groups = "drop") %>%
  ggplot(aes(x=date, y=count, color = is_universe)) +
  geom_line() +
  labs(
    title = 'Universe size'
)
```


Universe size
count
200
150
100
50
0
2020
2021
2022
2023
2024
date
is_universe
FALSE
TRUE

Calculate features

Next, calculate our features as before. We have:

- Short-term (10-day) cross-sectional momentum (bucketed into deciles by date)
- Short-term (1-day) cross-sectional carry (also bucketed into deciles by date)
- A breakout feature defined as the number of days since the 20-day high which we use as a time-series return predictor.

```r
# calculate features as before
rolling_days_since_high_20 <- purrr::possibly(
  tibbletime::rollify(
    function(x) {
      idx_of_high <- which.max(x)
      days_since_high <- length(x) - idx_of_high
      days_since_high
    },
    window = 20, na_value = NA),
  otherwise = NA
)

features <- universe %>%
  group_by(ticker) %>%
  arrange(date) %>%
  mutate(
    breakout = 9.5 - rolling_days_since_high_20(close),  # puts this feature on a scale -9.
    momo = close - lag(close, 10)/close,
    carry = funding_rate
  ) %>%
  ungroup() %>%
  na.omit()

# create a model df on our universe with momo and carry features scaled into deciles
model_df <- features %>%
  filter(is_universe) %>%
  group_by(date) %>%
  mutate(
    carry_decile = ntile(carry, 10),
    momo_decile = ntile(momo, 10),
    # also calculate demeaned return for everything in our universe each day for later
    demeaned_return = total_return_simple - mean(total_return_simple, na.rm = TRUE),
    demeaned_fwd_return = total_fwd_return_simple - mean(total_fwd_return_simple, na.rm = T
  ) %>%
  ungroup()

  # start simulation from date we first have n tickers in the universe
start_date <- features %>%
  group_by(date, is_universe) %>%
  summarize(count = n(), .groups = "drop") %>%
  filter(count >= trading_universe_size) %>%
  head(1) %>%
  pull(date)
```

Model expected returns

We'll model our features as expected returns. See here for a detailed exploration of this topic.

```
# use a 90-day window and refit every 10 days
is_days <- 90
step_size <- trading_universe_size*10

# rolling model for cross-sectional features
roll_xs_coeffs_df <- model_df %>%
  filter(date >= start_date) %>%
  regress(
    demeaned_fwd_return ~ carry_decile + momo_decile,
    m("lm", vcov. = "HAC"),
    .cv = "sliding_index",
    .cv_args = list(lookback = days(is_days), step = step_size, index = "date"),
    .force_cv = TRUE,
    .return_slices = TRUE
  )

# rolling model for time series features
breakout_cutoff <- 5.5  # below this level, we set our expected return to zero
roll_ts_coeffs_df <- model_df %>%
  filter(date >= start_date) %>%
  # setting regression weights to zero when breakout < breakout_cutoff will give these data
  mutate(regression_weights = case_when(breakout < breakout_cutoff ~ 0, TRUE ~ 1)) %>%
  regress(
    total_fwd_return_simple ~ breakout,
    m("lm", vcov. = "HAC"),
    .weights = "regression_weights",
    .cv = "sliding_index",
    .cv_args = list(lookback = days(is_days), step = step_size, index = "date"),
    .force_cv = TRUE,
    .return_slices = TRUE
  )
```

This results in a nested dataframe that contains the model objects and various metadata:

```
roll_xs_coeffs_df %>% head
roll_ts_coeffs_df %>% select(-settings) %>% head
```

A tidyfit.models: 6 × 7

model	estimator_fct	size (MB)	grid_id	model_object	settings	slice_id
<chr>	<chr>	<dbl>	<chr>	<list>	<list>	<chr>
lm	stats::lm	1.360960	#0010000	<environment: 0x000002c9993fbe90>	HAC	2021-02-11
lm	stats::lm	1.367016	#0010000	<environment: 0x000002c9994915b8>	HAC	2021-02-21
lm	stats::lm	1.380032	#0010000	<environment: 0x000002c9997f4918>	HAC	2021-03-03
lm	stats::lm	1.384864	#0010000	<environment: 0x000002c999c1dc98>	HAC	2021-03-13
lm	stats::lm	1.384808	#0010000	<environment: 0x000002c999f2ada0>	HAC	2021-03-23
lm	stats::lm	1.384944	#0010000	<environment: 0x000002c98f421ae8>	HAC	2021-04-02

A tidyfit.models: 6 × 6

model	estimator_fct	size (MB)	grid_id	model_object	slice_id
<chr>	<chr>	<dbl>	<chr>	<list>	<chr>
lm	stats::lm	1.245552	#0010000	<environment: 0x000002c99633fe90>	2021-02-11
lm	stats::lm	1.257864	#0010000	<environment: 0x000002c995893588>	2021-02-21
lm	stats::lm	1.254896	#0010000	<environment: 0x000002c989f4bd30>	2021-03-03
lm	stats::lm	1.255688	#0010000	<environment: 0x000002c9891edc98>	2021-03-13
lm	stats::lm	1.260624	#0010000	<environment: 0x000002c988f475b8>	2021-03-23
lm	stats::lm	1.257784	#0010000	<environment: 0x000002c987e70f28>	2021-04-02

`slice_id` is the date the model goes out of sample - so we'll need to make sure that we align our model coefficients to avoid using them on the data they were fitted on.

This requires a little data wrangling:

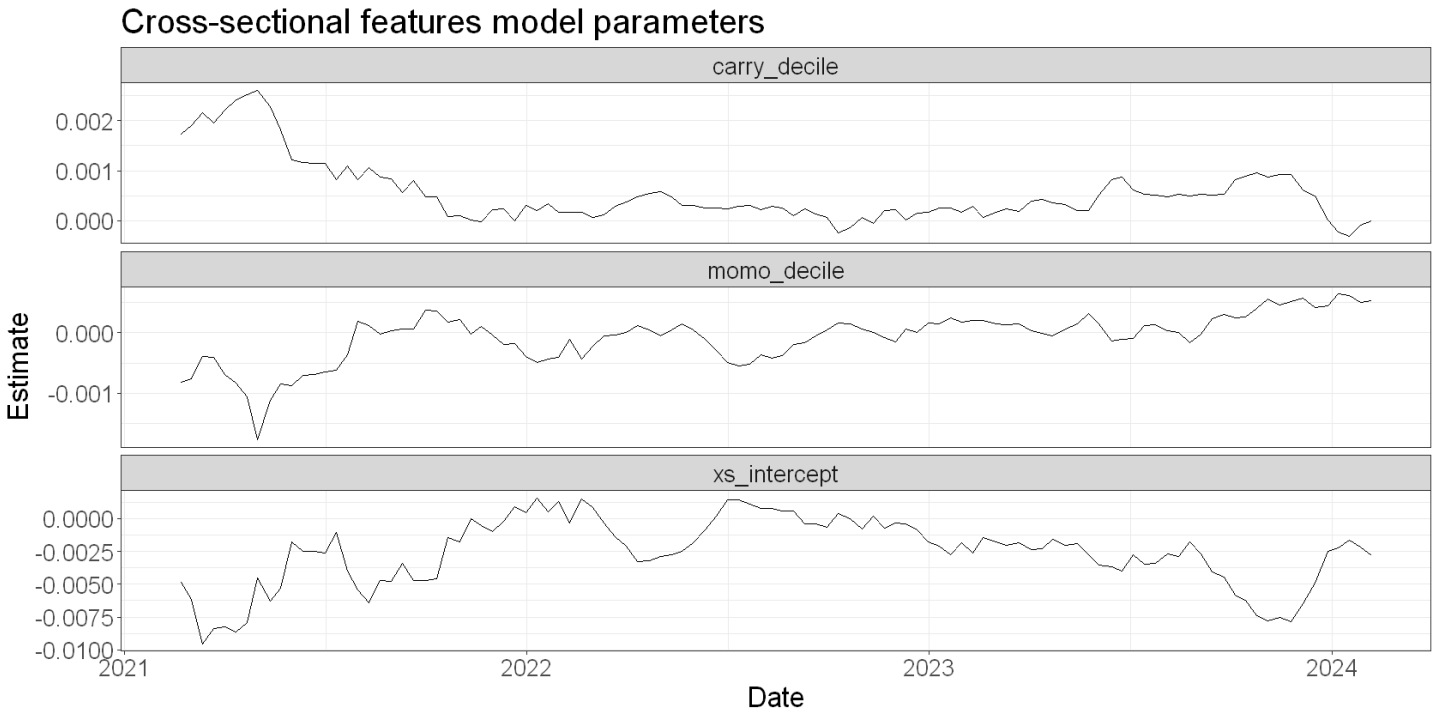
```
# for this to work, need to install.packages("sandwich", "lmtest")
xs_coefs <- roll_xs_coeffs_df %>%
  coef()

xs_coefs_df <- xs_coefs %>%
  ungroup() %>%
  select(term, estimate, slice_id) %>%
  pivot_wider(id_cols = slice_id, names_from = term, values_from = estimate) %>%
  mutate(slice_id = as_date(slice_id)) %>%
  # need to lag slice id to make it oos
  # slice_id_oos is the date we start using the parameters
  mutate(slice_id_oos = lead(slice_id)) %>%
  rename("xs_intercept" = `(Intercept)`)

ts_coefs <- roll_ts_coeffs_df %>%
  coef()

ts_coefs_df <- ts_coefs %>%
  ungroup() %>%
  select(term, estimate, slice_id) %>%
  pivot_wider(id_cols = slice_id, names_from = term, values_from = estimate) %>%
  mutate(slice_id = as_date(slice_id)) %>%
  # need to lag slice id to make it oos
  # slice_id_oos is the date we start using the parameters
  mutate(slice_id_oos = lead(slice_id)) %>%
  rename("ts_intercept" = `(Intercept)`)

xs_coefs_df %>% head
# ts_coefs_df %>% head
```

A tibble: 6 × 5

slice_id	xs_intercept	carry_decile	momo_decile	slice_id_oos
<date>	<dbl>	<dbl>	<dbl>	<date>
2021-02-11	-0.004872208	0.001732322	-0.0008239738	2021-02-21
2021-02-21	-0.006129608	0.001902553	-0.0007640190	2021-03-03
2021-03-03	-0.009538180	0.002148154	-0.0003897220	2021-03-13
2021-03-13	-0.008374302	0.001953655	-0.0004141515	2021-03-23
2021-03-23	-0.008275046	0.002211137	-0.0006898811	2021-04-02
2021-04-02	-0.008613423	0.002413307	-0.0008298451	2021-04-12

Here's a plot of our cross-sectional features' regression coefficients through time:

```r
# plot cross-sectional estimates
xs_coefs_df %>%
  select(-slice_id) %>%
  pivot_longer(cols = -slice_id_oos, names_to = "coefficient", values_to = "estimate") %>%
  ggplot(aes(x = slice_id_oos, y = estimate)) +
  geom_line() +
  facet_wrap(~coefficient, ncol = 1, scales = "free_y") +
  labs(
    title = "Cross-sectional features model parameters",
    x = "Date",
    y = "Estimate"
  )

# plot time-series estimates
# ts_coefs_df %>%
#   select(-slice_id) %>%
#   pivot_longer(cols = -slice_id_oos, names_to = "coefficient", values_to = "estimate") %>
#   ggplot(aes(x = slice_id_oos, y = estimate)) +
#   geom_line() +
#   facet_wrap(~coefficient, ncol = 1, scales = "free_y") +
#   labs(
#     title = "Time-series features model parameters",
#     x = "Date",
#     y = "Estimate"
#   )
```



The estimates for the coefficients for our carry and momentum features change over time to reflect the changing relationship with forward returns.

In particular, notice how the momentum coefficient flipped sign a few times, but especially from mid-2022, which is in line with our understanding of how the feature evolved.

Plot frictionless returns to our features

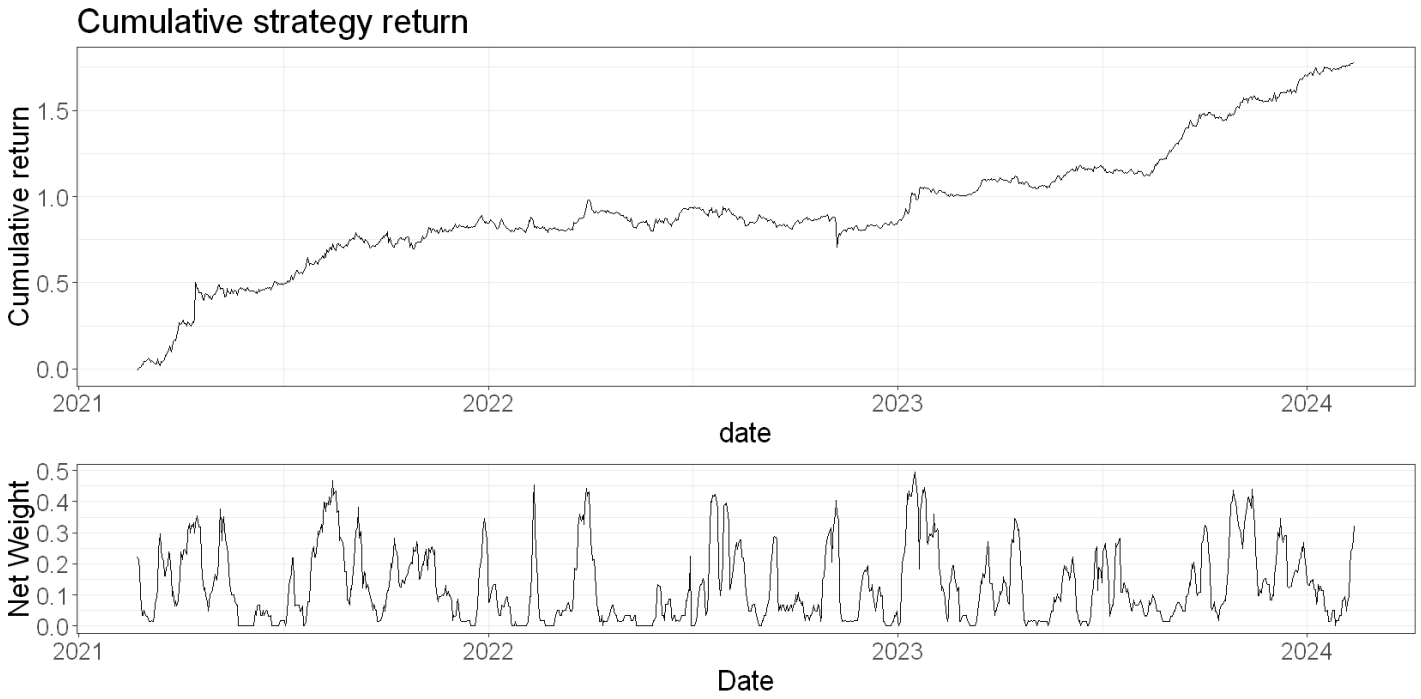
Now we can plot a time series of returns to a frictionless trading strategy based on these expected return estimates. This isn't a backtest - it makes no attempt to address real-world issues such as costs and turnover. It simply plots the returns to our predictions of expected returns over time.

I won't actually use the linear model of the breakout feature - instead I'll just set its expected return to 0.002 when it's greater than 5 and 0 otherwise.

I'll calculate target positions proportional to their cross-sectional return estimates. I'll then let the breakout feature tilt the portfolio net long, but I'll constrain the maximum delta that this feature can add to a position.

```r
# join and fill using slice_id to designate when the model goes oos
exp_return_df <- model_df %>%
  left_join(
    xs_coefs_df %>% left_join(ts_coefs_df, by = c("slice_id", "slice_id_oos")),
    by = join_by(closest(date > slice_id_oos)), suffix = c("_factor", "_coef")
  ) %>%
  na.omit() %>%
  # forecast cross-sectional expected return as
  mutate(expected_xs_return = carry_decile_factor*carry_decile_coef + momo_decile_factor*mo
  # mean expected xs return each day is zero
  # let total expected return be xs return + ts return - allows time series expected return
  mutate(expected_ts_return = case_when(breakout_factor >= 5.5 ~ 0.002, TRUE ~ 0)) %>%
  ungroup()

# long-short the xs expected return
# layer ts expected return on top
# position by expected return

# 1 in the numerator lets it get max 100% long due to breakout
max_ts_pos <- 0.5/trading_universe_size

strategy_df <- exp_return_df %>%
  filter(date >= start_date) %>%
  group_by(date) %>%
  mutate(xs_position = expected_xs_return - mean(expected_xs_return, na.rm = TRUE)) %>%
  # scale positions so that leverage is 1
  group_by(date) %>%
  mutate(xs_position = if_else(xs_position == 0, 0, xs_position/sum(abs(xs_position)))) %>%
  # layer ts expected return prediction
  ungroup() %>%
  mutate(ts_position = sign(expected_ts_return)) %>%
  # constrain maximum delta added by time series prediction
  mutate(ts_position = if_else(ts_position >= 0, pmin(ts_position, max_ts_pos), pmax(ts_pos
  mutate(position = xs_position + ts_position) %>%
  # strategy return
  mutate(strat_return = position*total_fwd_return_simple) %>%
  # scale back to leverage 1
  group_by(date) %>%
  mutate(position = if_else(position == 0, 0, position/sum(abs(position))))

returns_plot <- strategy_df %>%
  group_by(date) %>%
  summarise(total_ret = sum(strat_return)) %>%
  ggplot(aes(x = date, y = cumsum(log(1+total_ret)))) +
  geom_line() +
  labs(
    title = "Cumulative strategy return",
    y = "Cumulative return"
  )

weights_plot <- strategy_df %>%
  summarise(net_pos = sum(position)) %>%
  ggplot(aes(x = date, y = net_pos)) +
```

```
  geom_line() +
  labs(
    x = "Date",
    y = "Net Weight"
  )

returns_plot / weights_plot + plot_layout(heights = c(2,1))
```



Setting up the optimisation framework

There are a few things we need for our optimisation framework:

- A dataframe of prices and expected returns for all the tickers that were ever in the universe
- A risk model - we'll use a shrunk exponentially weighted covariance matrix
- We'll need a covariance matrix for each day of our simulation, so we'll compute these prior to running the simulation and store them in a list
- We'll also need a function for doing the optimisation at each time step and returning the weights for the next period
- Finally, we'll need to figure out appropriate values for lambda and tau (the parameters that control our risk and propensity to trade). We'll do this by testing out a few values on a sample of our data. We'll do this using parallel processing to speed things up a bit.

Then, we can do our actual simulation.

Prices and expected returns dataframes

First, we make a dataframe of prices and expected returns for all the tickers that were ever in the tradeable universe:

```
# tickers that were ever in the universe
universe_tickers <- features %>%
  filter(is_universe) %>%
  pull(ticker) %>%
  unique()

# universe_tickers
```

```
strategy_df <- exp_return_df %>%
  filter(date >= start_date) %>%
  group_by(date) %>%
  mutate(expected_xs_return = expected_xs_return - mean(expected_xs_return, na.rm = TRUE)) %>%
  ungroup() %>%
  mutate(expected_return = expected_xs_return + expected_ts_return) %>%
  select(date, ticker, expected_return) %>%
  # join back onto df of prices for all tickers that were ever in the universe
  # so that we have prices before and after a ticker comes into or out of the universe
  # for backtesting purposes
  right_join(
    features %>%
      filter(ticker %in% universe_tickers) %>%
      select(date, ticker, close, total_fwd_return_simple, funding_rate),
    by = c("date", "ticker")
  ) %>%
  arrange(date, ticker) %>%
  filter(date >= start_date)
```

```
# get same length exp returns vector for each date
exp_returns_wide <- strategy_df %>%
  select(ticker, date, expected_return) %>%
  # will give NA where a ticker didn't exist
  pivot_wider(id_cols = date, names_from = ticker, values_from = expected_return)
```

Risk model

Next, we need a risk model for each day in our simulation.

We'll use an exponentially weighted covariance estimate - this puts more weight on recent data and less on the past and tends to remove the big jumps you get using a fixed window approach. See here for more details on this approach.

We'll also shrink our covariance matrix:

- Off-diagonal elements (covariances) will be shrunk towards zero to reflect our uncertainty.
- On-diagonal elements (variances) will be shrunk towards the average variance.

First, make our exponentially weighted covariance estimates:

```r
# EWMA covariance estimate
# note definition of lambda in line with Risk Metrics
# ie higher values of lambda put less weight on the most recent returns and more weight on
ewma_cov <- function(x, y, lambda, initialisation_wdw = 100) {
  # check that x and y are the same length and greater than initialisation_wdw
  stopifnot("x and y must be of equal length" = length(x) == length(y))
  if(length(x) <= initialisation_wdw) {
    ewma_cov <- rep(NA, length(x))
    return(ewma_cov)
  }

  # create initialisation window and estimation window
  init_x = x[1:initialisation_wdw]
  init_y = y[1:initialisation_wdw]

  num_obs <- length(x)

  # initial covariance and mean return estimates
  old_cov <- cov(init_x, init_y)
  old_x <- mean(init_x)
  old_y <- mean(init_y)

  # preallocate output vector
  ewma_cov <- vector(mode = "numeric", length = num_obs)

  # pad with NA for initialisation window
  ewma_cov[1:initialisation_wdw] <- NA

  # covariance estimate
  for(i in c((initialisation_wdw+1):num_obs)) {
    ewma_cov[i] <- lambda*old_cov + (1 - lambda)*(old_x * old_y)
    old_cov <- ewma_cov[i]
    old_x <- x[i]
    old_y <- y[i]
  }
  ewma_cov
}
```

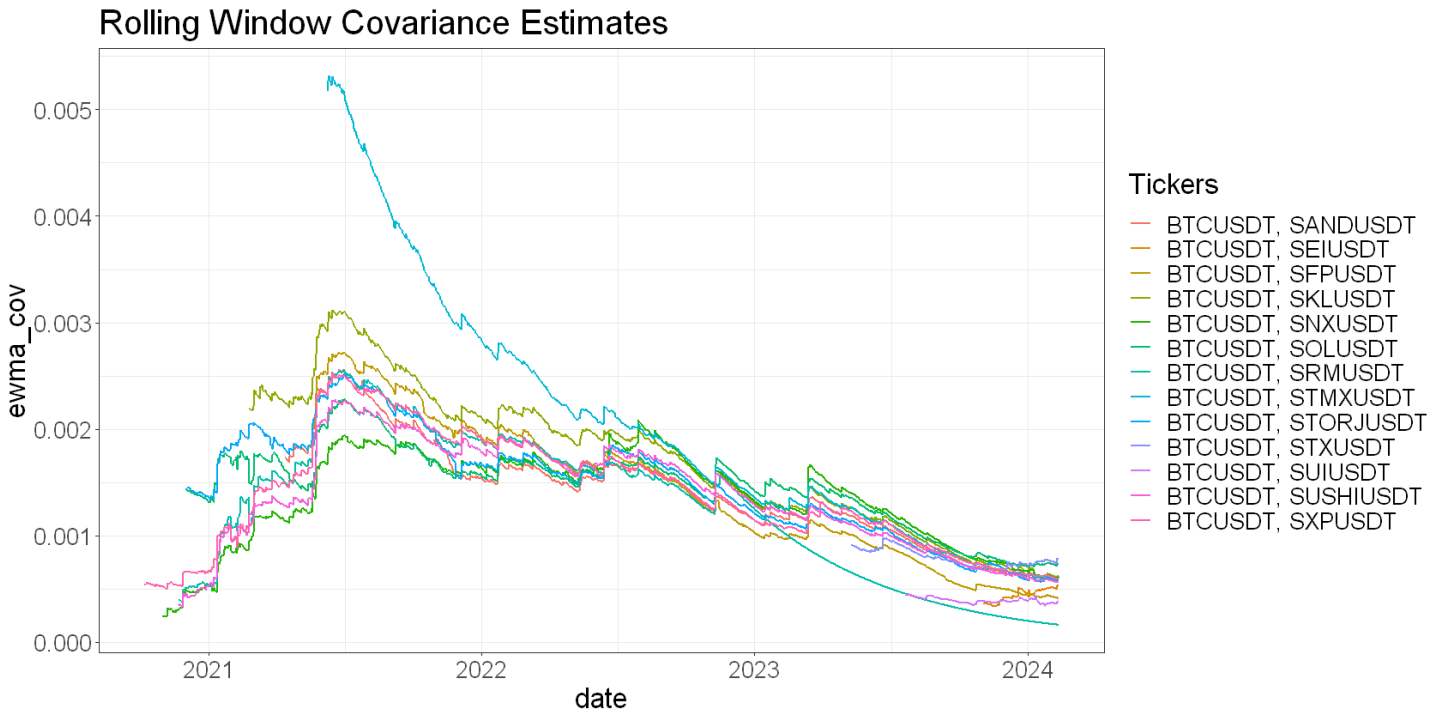
```
cov_lambda <- 0.995
wdw <- ew_cov_init_wdw

returns <- features %>%
  filter(ticker %in% universe_tickers) %>%
  select(ticker, date, total_return_simple)

# long dataframe of pairwise EW covariances
ewma_covs <- returns %>%
  full_join(returns, by = "date") %>%
  na.omit() %>%
  ungroup() %>%
  # get all combinations (tickers) and remove duplicate combos (eg BTC-ETH, ETH-BTC)
  mutate(tickers = ifelse(ticker.x < ticker.y, glue("{ticker.x}, {ticker.y}"), glue("{ticke
  distinct(date, tickers, .keep_all = TRUE) %>%
  # calculate rolling pairwise ewma correlations
  group_by(tickers) %>%
  arrange(date, .by_group = TRUE) %>%
  mutate(ewma_cov = ewma_cov(total_return_simple.x, total_return_simple.y, lambda = cov_lam
  select(date, tickers, ewma_cov, ticker.x, ticker.y) %>%
  na.omit() %>%
  ungroup()

tail(ewma_covs)
```

A tibble: 6 × 5

date	tickers	ewma_cov	ticker.x	ticker.y
<date>	<chr>	<dbl>	<chr>	<chr>
2024-02-07	ZRXUSDT, ZRXUSDT	0.003337849	ZRXUSDT	ZRXUSDT
2024-02-08	ZRXUSDT, ZRXUSDT	0.003321164	ZRXUSDT	ZRXUSDT
2024-02-09	ZRXUSDT, ZRXUSDT	0.003306101	ZRXUSDT	ZRXUSDT
2024-02-10	ZRXUSDT, ZRXUSDT	0.003290074	ZRXUSDT	ZRXUSDT
2024-02-11	ZRXUSDT, ZRXUSDT	0.003274115	ZRXUSDT	ZRXUSDT
2024-02-12	ZRXUSDT, ZRXUSDT	0.003259633	ZRXUSDT	ZRXUSDT

Here's a plot of some covariance estimates for BTC and various coins:

```
# plot pairwise rolling covariances
ewma_covs %>%
  filter(str_starts(tickers, "BTCUSDT, S")) %>%
  ggplot(aes(x = date, y = ewma_cov, colour = tickers)) +
    geom_line(size=0.8) +
    labs(
      title = "Rolling Window Covariance Estimates",
      colour = "Tickers"
    )
```



We'll also need some functions for extracting today's covariance matrix from our long dataframe of pairwise covariances, making sure it's symmetrical and positive semi definite, and shrinking its values.

I've also included a function `extend_covmat` that extends today's covariance matrix to include all the tickers that were ever in the universe. That's just to make the optimisation loop a little simpler - we can use consistent expected returns vectors and covariance matrixes at each iteration. For tickers not in the tradeable universe on a particular day, we set their:

- expected return to NA
- pairwise covariances to zero
- variances to the average variance of tradeable assets

We also explicitly constrain their weights to be zero in the optimisation.

```r
# functions for wrangling covariance matrix
library(Matrix)
library(matrixcalc)

get_today_covmat <- function(ewma_covs_long, today_date) {
    today_cov_df <- ewma_covs_long %>%
        ungroup() %>%
        filter(date == today_date) %>%
        select(ticker.x, ticker.y, ewma_cov) %>%
        pivot_wider(names_from = ticker.y, values_from = ewma_cov) %>%  # , names_prefix =
        column_to_rownames(var = "ticker.x")

    # check rownames and colnames match
    stopifnot("error making today's cov matrix" = all.equal(rownames(today_cov_df), colname
    today_covmat <- as.matrix(today_cov_df)
    today_covmat[lower.tri(today_covmat)] <- t(today_covmat)[lower.tri(today_covmat)]

    # return nearest positive semi-definite matrix
    today_covmat <- nearPD(today_covmat)
    return(today_covmat$mat)
}

#' shrinks covs towards zero, reflecting uncertainty in cov estimates. may increases varian
#' @param shrinkage_intensity: a value between 0 and 1. 0 means no shrinkage (use the origi
shrink_covmat <- function(covmat, shrinkage_intensity = 0.9) {
    # average variance from the diagonal
    average_variance <- mean(diag(covmat))
    # shrinkage target matrix (identity matrix scaled by average variance)
    target_matrix <- diag(average_variance, nrow = ncol(covmat))
    # shrunk covariance matrix
    covmat_shrunk <- (1 - shrinkage_intensity) * covmat + shrinkage_intensity * target_matr

    return(covmat_shrunk)
}

#' extend covmat to cover all tickers in the tradeable universe
#' put average variance on the diagonal, zeroes elsewhere
#' intent is that optimisation will explicitly constrain weights for assets not yet in the
extend_covmat <- function(covmat, all_tickers) {
    current_tickers <- colnames(covmat)

    full_covmat <- matrix(0, nrow = length(all_tickers), ncol = length(all_tickers),
         dimnames = list(all_tickers, all_tickers))

    current_tickers_ordered <- intersect(all_tickers, rownames(covmat))

    # ensure covmats are ordered correctly
    covmat_reordered <- covmat[current_tickers_ordered, current_tickers_ordered] %>%
      # ensure is Matrix type for compatibility
      as.matrix()
    full_covmat <- full_covmat[all_tickers, all_tickers]

    # replace in full_covmat
```

```
    full_covmat[current_tickers_ordered, current_tickers_ordered] <- covmat_reordered

    # replace zeros on diagonal with average variance
    zero_diag_indices <- which(diag(full_covmat) == 0)
    average_variance <- mean(diag(covmat))
    diag(full_covmat)[zero_diag_indices] <- average_variance

    return(full_covmat)
}

wrangle_covmat <- function(ewma_covs_long, date, tickers, shrinkage_intensity = 0.9) {
    today_covmat <- get_today_covmat(ewma_covs_long, date) %>%
      shrink_covmat(shrinkage_intensity = shrinkage_intensity) %>%
      extend_covmat(all_tickers = tickers)

    # check
    if(is.positive.semi.definite(today_covmat, tol=1e-8)){
        if(isSymmetric(today_covmat)) {
            glue("covmat crated for {date} not symmetric")
        }
    } else {
        glue("covmat crated for {date} not PSD")
    }

    return(today_covmat)
}
```

```
Attaching package: 'Matrix'


The following objects are masked from 'package:tidyr':

    expand, pack, unpack



Attaching package: 'matrixcalc'


The following object is masked from 'package:CVXR':

    vec
```

Next we precompute our covariance estimates for each day in our simulation and store them in a list. This just saves a bit of time in the optimisation loop.

```r
# precompute covariances matrixes
simulation_start_date <- start_date + days(100)  # allow for first in-sample period - other
exp_returns_sim_df <- exp_returns_wide %>% filter(date >= simulation_start_date)

dates <- exp_returns_sim_df$date
covmat_list <- map(dates, ~wrangle_covmat(ewma_covs, .x, tickers = universe_tickers, shrink
```

Define optimisation function

Next we define a function for finding our optimal weights at each time step. I'll use the mean-variance optimisation with costs. We explore this in more detail in this article.

Notice that we constrain the portfolio to be unleveraged (max total weight less than or equal to 1), but we don't put a constraint on how *net* long or short the portfolio can be. That means that in the extreme, we could end up being fully long or fully short. Below, I'll show you an example of constraining the net position as well.

We also pass `na_idxs` to tell the optimiser which assets in our expected returns vector are not in the tradeable universe and should therefore get a weight of zero.

```
# define our mvo function
mvo_with_costs <- function(expected_returns, current_weights, na_idxs = c(), costs, covmat,
  # define our weights vector as a CVXR::Variable
  weights <- Variable(length(expected_returns))
  # define an alpha term as the weighted sum of expected returns
    # again, express using linear algebra
  alpha_term <- (t(weights) %*% expected_returns)
  # define a costs term. depends on:
    # cost of trading - needs to be expressed such that it scales with expected returns
      # calculate as elementwise cost * absolute value of weights - current_weights
      # use CVXR::multiply and CVXR::abs
    # absolute distance of current_weights to our weights variable
      # the more our target weights differ from current weights, the more it costs to trade
      # this is a decent representation of fixed percentage costs, but doesn't capture mini
    # sum_entries is a CVXR function for summing the elements of a vector
  costs_term <- sum_entries(multiply(costs, abs(weights - current_weights))) # elementwise
  # define a risk term as w*Sigma*w
    # quad_form is a CVXR function for doing w*Sigma*w
  risk_term <- quad_form(weights, covmat)
  # define our objective
    # maximise our alpha less our risk term multiplied by some factor, lambda, less our cos
  objective <- Maximize(alpha_term - lambda*risk_term - tau*costs_term)
  # apply our no leverage constraint
  constraints <- list(cvxr_norm(weights, 1) <= 1, weights[na_idxs] == 0)
  # specify the problem
  problem <- Problem(objective, constraints)
  # solve
  result <- solve(problem)

  # return the values of the variable we solved for
  result$getValue(weights)
}
```

Explore lambda-tau parameter space

Next we need to figure out some reasonable values for lambda (our risk aversion) and tau (our propensity to trade).

If you need a refresher on lambda and tau, see the previous article on building intuition for optimisation. The TLDR is:

- Higher values of lambda will put more weight on the risk term and lead to higher diversification and eventually to reduced position sizes.
- Higher values of tau will reduce our trading by requiring a greater expected return hurdle.

We probably don't need to run the simulation over our entire history in order to find reasonable values of lambda and tau. So we'll just use 500 days to figure out our parameter values, then do a full simulation on that single parameter set.

We'll loop over four values of lambda and five values of tau, for a total of 20 simulations over 500 days each. And we'll do these in parallel to speed things up.

Here's how to set up a parallel backend and prepare each worker environment:

```
library(doParallel)
library(foreach)

# register parallel backend
num_cores <- detectCores() - 1  # leave one core free
cl <- makeCluster(num_cores)
registerDoParallel(cl)

# use clusterEvalQ to define functions and objects available to each worker
clusterExport(
  cl,
  varlist = c("get_today_covmat", "shrink_covmat", "extend_covmat", "wrangle_covmat", "mvo_
)

# load libraries in each worker
clusterEvalQ(cl, {
  library(CVXR); library(Matrix); library(matrixcalc); library(dplyr); library(tidyr); libr
})
```

```
Loading required package: foreach


Attaching package: 'foreach'


The following objects are masked from 'package:purrr':

    accumulate, when


Loading required package: iterators

Loading required package: parallel
```

1. 'glue' · 'tidyr' · 'dplyr' · 'matrixcalc' · 'Matrix' · 'CVXR' · 'stats' · 'graphics' · 'grDevices' · 'utils' · 'datasets' · 'methods' · 'base'
2. 'glue' · 'tidyr' · 'dplyr' · 'matrixcalc' · 'Matrix' · 'CVXR' · 'stats' · 'graphics' · 'grDevices' · 'utils' · 'datasets' · 'methods' · 'base'
3. 'glue' · 'tidyr' · 'dplyr' · 'matrixcalc' · 'Matrix' · 'CVXR' · 'stats' · 'graphics' · 'grDevices' · 'utils' · 'datasets' · 'methods' · 'base'
4. 'glue' · 'tidyr' · 'dplyr' · 'matrixcalc' · 'Matrix' · 'CVXR' · 'stats' · 'graphics' · 'grDevices' · 'utils' · 'datasets' · 'methods' · 'base'
5. 'glue' · 'tidyr' · 'dplyr' · 'matrixcalc' · 'Matrix' · 'CVXR' · 'stats' · 'graphics' · 'grDevices' · 'utils' · 'datasets' · 'methods' · 'base'
6. 'glue' · 'tidyr' · 'dplyr' · 'matrixcalc' · 'Matrix' · 'CVXR' · 'stats' · 'graphics' · 'grDevices' · 'utils' · 'datasets' · 'methods' · 'base'
7. 'glue' · 'tidyr' · 'dplyr' · 'matrixcalc' · 'Matrix' · 'CVXR' · 'stats' · 'graphics' · 'grDevices' · 'utils' · 'datasets' · 'methods' · 'base'

Next, define our parameters for the simulations:

```
num_days <- nrow(exp_returns_wide)
lambdas <- c(0.1, 0.3, 1, 3)
taus <- c(0.003, 0.1, 0.3, 1, 3)
date_idxs_to_do <- c(1:500)
costs <- 0.15/100
```

Finally, we can do our 500-day simulations for each combination of lambda and tau in parallel. This still takes a while to complete, so we'll save the results to disk:

```r
# custom combine function for flattening results of foreach to one big list
# ensures that each result from the inner loop is appended to the main list, rather than ne
# results in a flat structure where each element is a list corresponding to an iteration ov
# requres .init arg of foreach to be list() - initialises an empty list for storing the res
flatten_lists <- function(x, y) {
  c(x, y)
}

results <- foreach(lambda = lambdas, .combine = 'flatten_lists', .init = list()) %:%   #
  foreach(tau = taus, .combine = 'flatten_lists', .init = list()) %dopar% {

   weights <- list()
   errors <- list()
   w0 <- rep(0, length(universe_tickers))
   for( i in date_idxs_to_do ) {
      # today's date
      d = exp_returns_sim_df$date[i]

      # cov estimate
      today_covmat <- covmat_list[[i]]

      # check cov matrix and exp returns vector are ticker-aligned
      # all.equal(exp_returns_sim_df %>% filter(date == d) %>% pivot_longer(-date, names_to

      # get row of expected returns as a vector in the same order as the columns of the cov
      # contains NA at this point
      exp_rets <- exp_returns_sim_df %>% filter(date == d) %>% pivot_longer(-date, names_to

      # get na indexes
      # we'll explicitly constrain these to get zero weight
      na_idxs <- which(is.na(exp_rets))
      # convert NA to zero
      exp_rets[is.na(exp_rets)] <- 0

      # initialise w in case we get a solver error (can retain w0 in these cases)
      w <- w0

      # solve for next period's weights
      tryCatch({
         w <- mvo_with_costs(expected_returns = exp_rets, current_weights = w0, na_idxs = n
      }, error = function(e) {
            # in case of solver error, log and retain existing weights
            errors[[i]] <- e
         }
      )

      # store weights
      # w is a one-column matrix
      weights[[i]] <- w
      w0 <- w
    }
    # return a list with weights, parameters, and any errors
    list(list(weights = weights, lambda = lambda, tau = tau, errors = errors))
```

```
}

stopCluster(cl)

# save to disk
saveRDS(results, file = "lambda_tau_search_results_1_500.rds")
```

```
# read results from disk
# results <- readRDS("lambda_tau_search_results_1_500.rds")
```

Wrangle simulation results

Our simulation results are a list of lists - one sublist for each lambda-tau combination. Each sublist contains 500 weights vectors (one for each day in the simulation), the values of lambda and tau, and a list of solver errors, if any occurred.

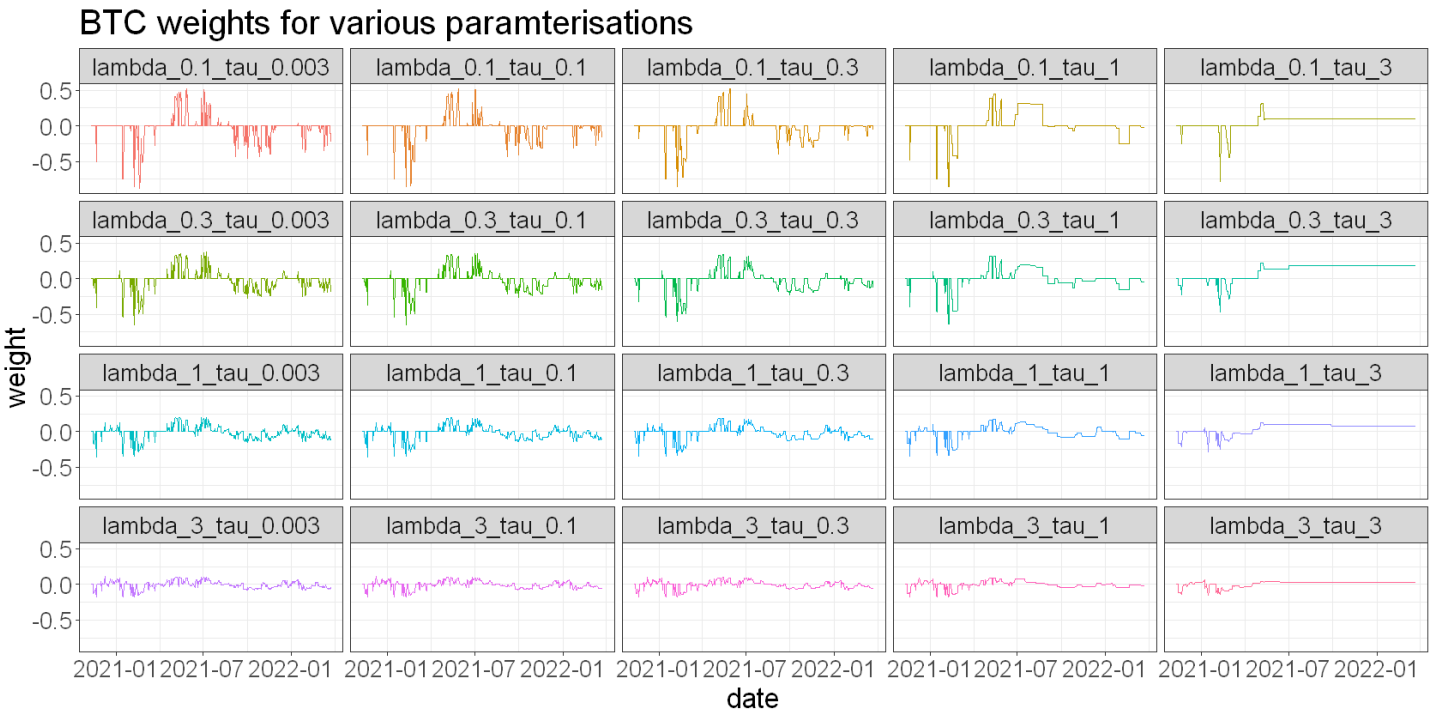
We can make one big dataframe of weights for each lambda-tau combination using the awesome `purrr` tools:

```
# function for converting a day's weights vector to a dataframe
weights_mat_to_df <- function(x, d, lambda, tau, tickers) {
  x %>%
    as_tibble(.name_repair = ~paste0("lambda_",lambda,"_tau_",tau)) %>%
    mutate(date = d) %>%
    mutate(ticker = tickers)
}

# make one dataframe of weights for each lambda-tau combination
weights_dfs <- list()
for(i in seq_along(results)) {
  this_entry <- results[[i]]
  these_weights <- this_entry$weights
  this_lambda <- this_entry$lambda
  this_tau <- this_entry$tau

  weights_df <- purrr::map2(
    these_weights,
    exp_returns_wide$date[date_idxs_to_do],
    ~weights_mat_to_df(.x, .y, lambda = this_lambda, tau = this_tau, tickers = universe_tic
  ) %>%
    bind_rows()

  weights_dfs[[i]] <- weights_df
}

# combine dataframes for each lmbda-tau combination into one big dataframe
all_weights <- weights_dfs %>% purrr::reduce(left_join, by = c("date", "ticker")) %>%
  relocate(c(date, ticker))

head(all_weights)
```

date	ticker	lambda_0.1_tau_0.003	lambda_0.1_tau_0.1	lambda_0.1_tau_0.3	lambda
<date>	<chr>	<dbl>	<dbl>	<dbl>	
2020-11-10	BTCUSDT	0	0	0	
2020-11-10	ETHUSDT	0	0	0	
2020-11-10	BCHUSDT	0	0	0	
2020-11-10	XRPUSDT	0	0	0	
2020-11-10	EOSUSDT	0	0	0	
2020-11-10	LTCUSDT	0	0	0	

Let's look at how our different parameterisations impact the weights for BTC:

```
all_weights %>%
  pivot_longer(cols = c(-date, -ticker), names_to = "parameterisation", values_to = "weight
  filter(ticker == "BTCUSDT") %>%
  ggplot(aes(x = date, y = weight, colour = parameterisation)) +
  geom_line() +
  facet_wrap(~parameterisation) +
  theme(legend.position = "none") +
  labs(
    title = "BTC weights for various paramterisations"
  )
```

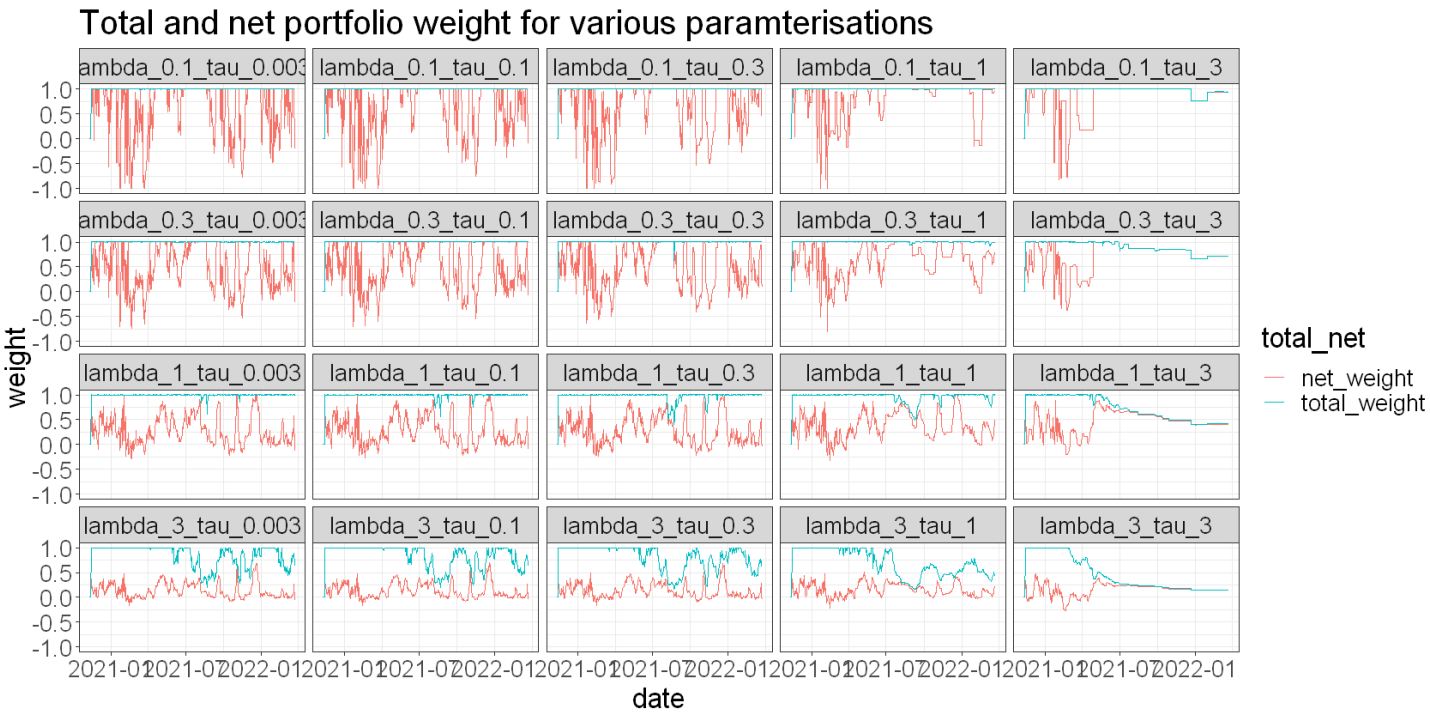


You can see that as lambda increases, the weight we put on BTC decreases - as we increase lambda, the optimiser would start by assigning some of BTC's weight to other tickers in order to increase diversification, and as we increase lambda further, the total portfolio exposure would decrease, further reducing the weight on BTC.

And you can see that as tau increases, there are fewer changes in those weights because we do less trading. In the extreme, we hold our existing position for far longer than we would like to.

Let's look at how our total and net portfolio weight changes:

```
all_weights %>%
  pivot_longer(cols = c(-date, -ticker), names_to = "parameterisation", values_to = "weight
  group_by(parameterisation, date) %>%
  summarise(total_weight = sum(abs(weight)), net_weight = (sum(weight)), .groups = "drop")
  pivot_longer(cols = c(total_weight, net_weight), names_to = "total_net", values_to = "wei
  ggplot(aes(x = date, y = weight, colour = total_net)) +
  geom_line() +
  facet_wrap(~parameterisation) +
  labs(
    title = "Total and net portfolio weight for various paramterisations"
  )
```



You can see that for lower values of lambda, the portfolio is mostly fully invested (blue lines). As we increase lambda, we have periods where we reduce our total portfolio exposure. This becomes more extreme as we increase lambda.

Also notice that for lower values of lambda, we can have more extreme net long or short positions (red lines). As we increase lambda, our net position tends not to fluctuate as much. We also tend not be net short as often, reflecting the long-only nature of the breakout feature, and the fact that one or two large negative expected returns will have less impact as lambda increases.

Simulate our parameterisations

Next, we'll use `rsims` to run an accurate simulation for each of our parameterisations. See this article for more information and examples of using `rsims`.

We'll assume costs of 0.15% of traded value.

```
backtest_df <- all_weights %>%
  pivot_longer(cols = c(-date, -ticker), names_to = "parameterisation", values_to = "weight
  left_join(strategy_df, by = c("ticker", "date"))

head(backtest_df)
```

A tibble: 6 × 8

date	ticker	parameterisation	weight	expected_return	close	total_fwd_retu
<date>	<chr>	<chr>	<dbl>	<dbl>	<dbl>	
2020-11-10	BTCUSDT	lambda_0.1_tau_0.003	0	NA	15167.88	0
2020-11-10	BTCUSDT	lambda_0.1_tau_0.1	0	NA	15167.88	0
2020-11-10	BTCUSDT	lambda_0.1_tau_0.3	0	NA	15167.88	0
2020-11-10	BTCUSDT	lambda_0.1_tau_1	0	NA	15167.88	0
2020-11-10	BTCUSDT	lambda_0.1_tau_3	0	NA	15167.88	0
2020-11-10	BTCUSDT	lambda_0.3_tau_0.003	0	NA	15167.88	0

```
params <- unique(backtest_df$parameterisation)
params
```

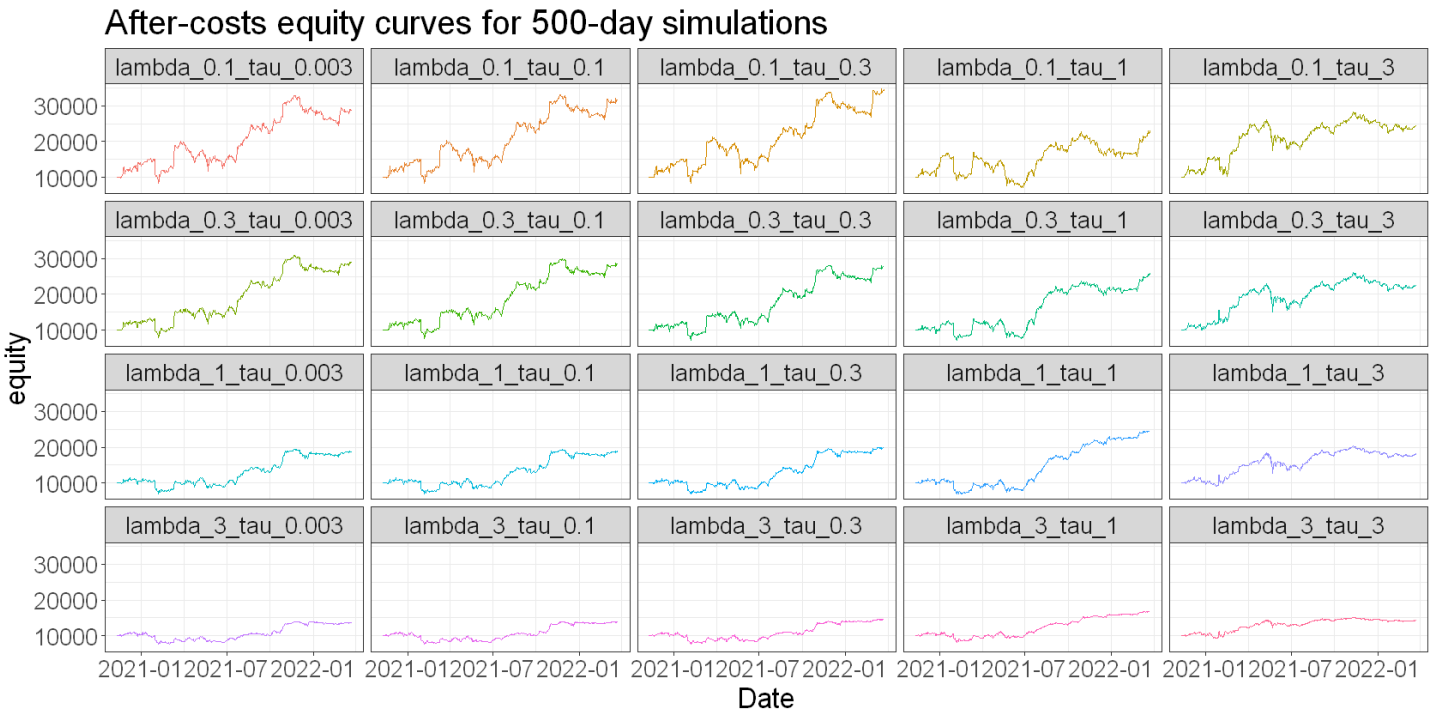
'lambda_0.1_tau_0.003' · 'lambda_0.1_tau_0.1' · 'lambda_0.1_tau_0.3' · 'lambda_0.1_tau_1' · 'lambda_0.1_tau_3' · 'lambda_0.3_tau_0.003' · 'lambda_0.3_tau_0.1' · 'lambda_0.3_tau_0.3' · 'lambda_0.3_tau_1' · 'lambda_0.3_tau_3' · 'lambda_1_tau_0.003' · 'lambda_1_tau_0.1' · 'lambda_1_tau_0.3' · 'lambda_1_tau_1' · 'lambda_1_tau_3' · 'lambda_3_tau_0.003' · 'lambda_3_tau_0.1' · 'lambda_3_tau_0.3' · 'lambda_3_tau_1' · 'lambda_3_tau_3'

```r
# simulation for each parameterisation

# no trade buffer (trades controlled by tau parameter and are embedded in weights)
equity_curves <- vector("list", length = length(params))
results_dfs <- vector("list", length = length(params))
i <- 1
for( p in params ) {
    # get weights as a wide matrix
    # note that date column will get converted to unix timestamp
    backtest_weights <- backtest_df %>%
        dplyr::filter(parameterisation == p) %>%
        pivot_wider(id_cols = date, names_from = ticker, values_from = c(close, weight)) %>
        select(date, starts_with("weight")) %>%
        mutate(date = as.numeric(date)) %>%
        data.matrix()

    # NA weights should be zero
    backtest_weights[is.na(backtest_weights)] <- 0

    # get prices as a wide matrix
    # note that date column will get converted to unix timestamp
    backtest_prices <- backtest_df %>%
        dplyr::filter(parameterisation == p) %>%
        pivot_wider(id_cols = date, names_from = ticker, values_from = c(close, weight)) %>
        select(date, starts_with("close_")) %>%
        mutate(date = as.numeric(date)) %>%
        data.matrix()

    # get funding as a wide matrix
    # note that date column will get converted to unix timestamp
    backtest_funding <- backtest_df %>%
        dplyr::filter(parameterisation == p) %>%
        pivot_wider(id_cols = date, names_from = ticker, values_from = c(close, funding_rat
        select(date, starts_with("funding_rate_")) %>%
        mutate(date = as.numeric(date)) %>%
        data.matrix()

    # simulation
    results_df <- fixed_commission_backtest_with_funding(
        prices = backtest_prices,
        target_weights = backtest_weights,
        funding_rates = backtest_funding,
        trade_buffer = 0.,
        initial_cash = 10000,
        margin = 0.05,
        commission_pct = 0.0015,
        capitalise_profits = FALSE
        ) %>%
        mutate(ticker = str_remove(ticker, "close_")) %>%
        # remove coins we don't trade from results
        drop_na(Value)

    margin <- results_df %>%
```

```
        group_by(Date) %>%
        summarise(Margin = sum(Margin, na.rm = TRUE))

    cash_balance <- results_df %>%
        filter(ticker == "Cash") %>%
        select(Date, Value) %>%
        rename("Cash" = Value)

    equity <- cash_balance %>%
        left_join(margin, by = "Date") %>%
        mutate(Equity = Cash + Margin) %>%
        select(Date, Equity) %>%
        rename(!!p := Equity)

    equity_curves[[i]] <- equity
    results_dfs[[i]] <- results_df

    i <- i + 1
}
```

Plot equity curves for each 500-day simulation:

```
equity_curves %>%
  purrr::discard(is.null) %>%
  reduce(left_join, by = "Date") %>%
  pivot_longer(-Date, names_to = "param", values_to = "equity") %>%
  ggplot(aes(x = Date, y = equity, colour = param)) +
  geom_line() +
  facet_wrap(~param) +
  theme(legend.position = "none") +
  labs(
    title = "After-costs equity curves for 500-day simulations"
  )
```



Eyeballing equity curves is illustrative, but let's also quantify these results. For each 500-day simulation, we'll calculate:

- Sharpe ratio
- Total return
- Average daily turnover (as percentage of allocated cash)

```r
initial_cash <- 10000
i <- 1
dfs <- list()
for (r in results_dfs) {
  # get lambda and tau
  these_params <- str_extract_all(params[i],"[0-9]+\\.*[0-9]*", simplify = TRUE)
  this_lambda <- these_params[1, 1]
  this_tau <- these_params[1, 2]

  # average daily turnover
  average_turnover <- r %>%
    filter(ticker != "Cash") %>%
    group_by(Date) %>%
    summarise(turnover = 100*sum(abs(TradeValue))/initial_cash, .groups = "drop") %>%
    summarise(ave_turnover = mean(turnover)) %>%
    pull(ave_turnover)

  # total return
  equity <- equity_curves[[i]][, 2, drop = TRUE]

  fin_eq <- equity %>%
    tail(1)

  init_eq <- equity%>%
    head(1)

  total_return <- (fin_eq/init_eq - 1) * 100
  days <- length(equity)
  ann_return <- total_return * 365/days

  # sharpe
  returns <- na.omit(equity/lag(equity) - 1)
  sharpe <- sqrt(365)*mean(returns)/sd(returns)

  # store results
  df <- data.frame(
    lambda = this_lambda,
    tau = this_tau,
    sharpe = sharpe,
    total_return = total_return,
    average_turnover = average_turnover
  )
  dfs[[i]] <- df

  i <- i + 1
}

metrics <- dfs %>% bind_rows()
metrics %>% arrange(desc(sharpe))
```

A data.frame: 20 × 5

lambda <chr>	tau <chr>	sharpe <dbl>	total_return <dbl>	average_turnover <dbl>
0.3	0.003	1.5979996	187.91581	86.563401
1	1	1.5843898	147.20999	26.719370
0.1	0.3	1.5632515	243.76855	62.853329
0.3	0.1	1.5582835	184.85528	74.244957
0.1	0.1	1.5049803	215.82454	82.633311
0.3	0.3	1.4733322	177.88543	57.872224
3	1	1.4360708	68.91465	23.088435
0.1	0.003	1.4106634	187.10321	98.365393
0.3	1	1.3676426	157.92362	28.740003
0.1	3	1.3467732	146.92856	13.373507
0.3	3	1.3300004	127.71079	11.963360
1	0.3	1.2548320	98.57962	48.687742
1	0.003	1.1851221	87.42170	71.599827
1	0.1	1.1803581	87.72210	62.037604
0.1	1	1.1530450	128.67032	30.273378
1	3	1.1099375	81.99863	11.013410
3	3	1.0846910	43.49599	8.641305
3	0.3	1.0200999	45.43914	38.044012
3	0.1	0.8968249	38.13561	46.235756
3	0.003	0.8670867	36.31248	51.907027

These metrics give you an idea of the tradeoffs:

- The parameters that resulted in our highest Sharpe turn over more than 85% of the portfolio daily. This would be hard to manage if you were trading by hand.
- You could achieve a slightly lower Sharpe but turn over much less, for example with lambda = 1, tau = 1. This would be operationally simpler.

Bear in mind though that these results only represent the first 500-days of the simulation, and there's no guarantee that they would be reflective of the results over the entire history. But they're a good starting point for thinking about the tradeoffs.

For now, let's pick lambda = 1 and tau = 1 and simulate over the full sample:

```r
lambda <- 1
tau <- 1
costs <- 0.15/100

weights <- list()
errors <- list()
w0 <- rep(0, length(universe_tickers))
for( i in c(1:length(exp_returns_sim_df$date)) ) {
    # today's date
    d = exp_returns_sim_df$date[i]

    # cov estimate
    today_covmat <- covmat_list[[i]]

    # check cov matrix and exp returns vector are ticker-aligned
    # all.equal(exp_returns_sim_df %>% filter(date == d) %>% pivot_longer(-date, names_to =

    # get row of expected returns as a vector in the same order as the columns of the covar
    # contains NA at this point
    exp_rets <- exp_returns_sim_df %>% filter(date == d) %>% pivot_longer(-date, names_to =
    # get na indexes
    # we'll explicitly constrain these to get zero weight
    na_idxs <- which(is.na(exp_rets))
    # convert NA to zero
    exp_rets[is.na(exp_rets)] <- 0
    # initialise w in case we get a solver error (can retain w0 in these cases)
    w <- w0
    tryCatch({
        w <- mvo_with_costs(expected_returns = exp_rets, current_weights = w0, na_idxs = na
    }, error = function(e) {
        # in case of solver error, log and retain existing weights
        errors[[i]] <- e
        }
    )

    # w is a one-column matrix
    weights[[i]] <- w
    w0 <- w
}

results <- list(weights = weights, lambda = lambda, tau = tau, errors = errors)

saveRDS(results, file = "lambda_1_tau_1.rds")
```

```
# wrangle weights into a dataframe
weights_df <- purrr::map2(
  weights,
  exp_returns_sim_df$date,
  ~weights_mat_to_df(.x, .y, lambda = lambda, tau = tau, tickers = universe_tickers)
) %>%
  bind_rows() %>%
  rename("weight" = lambda_1_tau_1)

head(weights_df)
```

A tibble: 6 × 3

weight	date	ticker
<dbl>	<date>	<chr>
0	2021-02-18	BTCUSDT
0	2021-02-18	ETHUSDT
0	2021-02-18	BCHUSDT
0	2021-02-18	XRPUSDT
0	2021-02-18	EOSUSDT
0	2021-02-18	LTCUSDT

```
# join weights onto prices
backtest_df <- weights_df %>%
  left_join(strategy_df, by = c("ticker", "date"))

head(backtest_df)
```

A tibble: 6 × 7

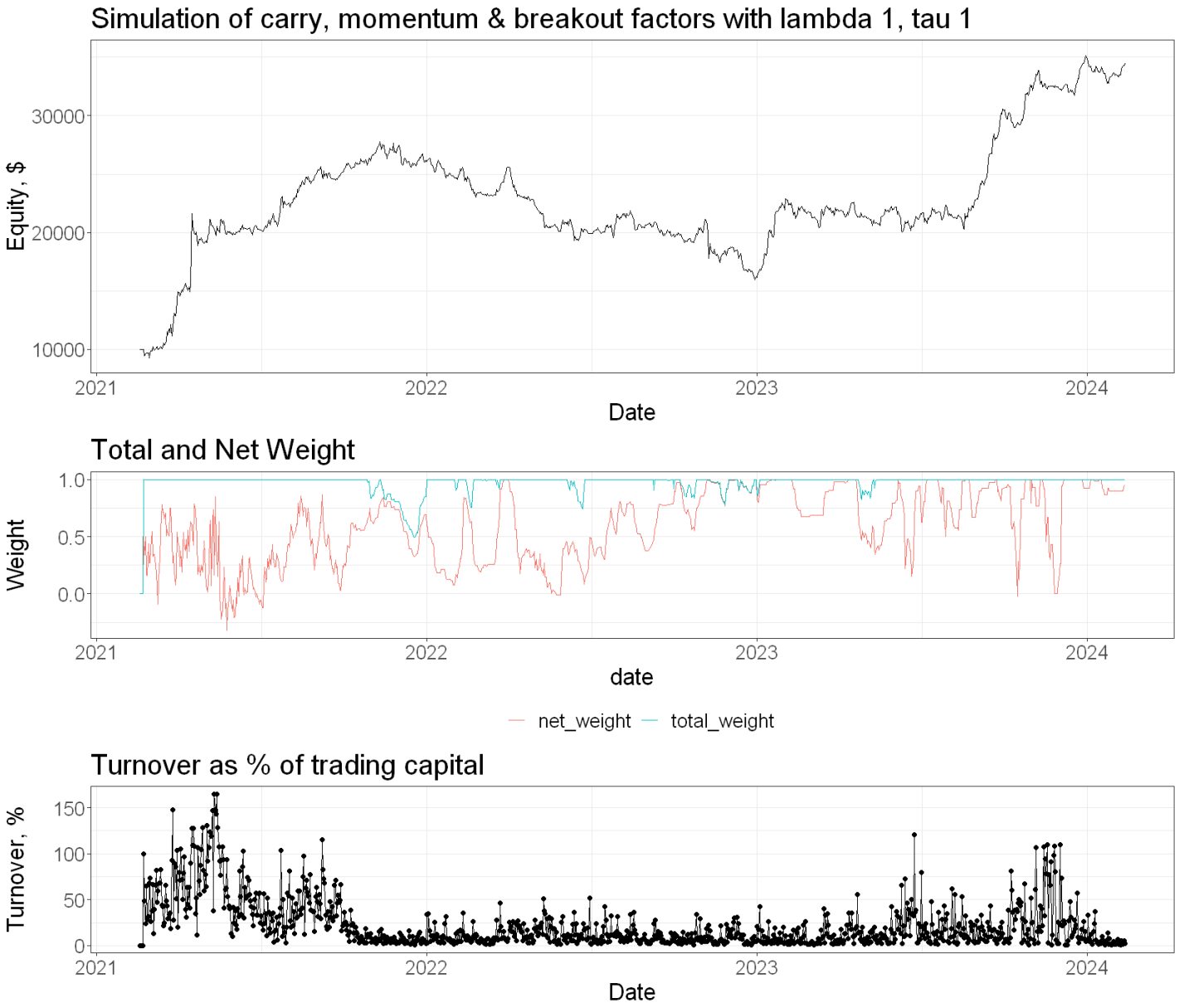
weight	date	ticker	expected_return	close	total_fwd_return_simple	funding
<dbl>	<date>	<chr>	<dbl>	<dbl>	<dbl>	<
0	2021-02-18	BTCUSDT	NA	51663.6400	0.041106415	-0.004
0	2021-02-18	ETHUSDT	NA	1910.9900	0.007813203	-0.005
0	2021-02-18	BCHUSDT	NA	702.5500	0.015704896	-0.007
0	2021-02-18	XRPUSDT	NA	0.5302	0.030592698	-0.006
0	2021-02-18	EOSUSDT	NA	4.8000	0.077784750	-0.005
0	2021-02-18	LTCUSDT	NA	225.8000	0.020896880	-0.006

```r
# simulate

# get weights as a wide matrix
backtest_weights <- backtest_df %>%
  pivot_wider(id_cols = date, names_from = ticker, values_from = c(close, weight)) %>%  # p
  select(date, starts_with("weight")) %>%
  mutate(date = as.numeric(date)) %>%
  data.matrix()

# NA weights should be zero
backtest_weights[is.na(backtest_weights)] <- 0

# get prices as a wide matrix
backtest_prices <- backtest_df %>%
  pivot_wider(id_cols = date, names_from = ticker, values_from = c(close, weight)) %>%  # p
  select(date, starts_with("close_")) %>%
  mutate(date = as.numeric(date)) %>%
  data.matrix()

# get funding as a wide matrix
backtest_funding <- backtest_df %>%
  pivot_wider(id_cols = date, names_from = ticker, values_from = c(close, funding_rate)) %>
  select(date, starts_with("funding_rate_")) %>%
  mutate(date = as.numeric(date)) %>%
  data.matrix()

# simulation
results_df <- fixed_commission_backtest_with_funding(
  prices = backtest_prices,
  target_weights = backtest_weights,
  funding_rates = backtest_funding,
  trade_buffer = 0.,
  initial_cash = 10000,
  margin = 0.05,
  commission_pct = 0.0015,
  capitalise_profits = FALSE
) %>%
mutate(ticker = str_remove(ticker, "close_")) %>%
# remove coins we don't trade from results
drop_na(Value)

margin <- results_df %>%
  group_by(Date) %>%
  summarise(Margin = sum(Margin, na.rm = TRUE))

cash_balance <- results_df %>%
  filter(ticker == "Cash") %>%
  select(Date, Value) %>%
  rename("Cash" = Value)

equity <- cash_balance %>%
  left_join(margin, by = "Date") %>%
```

```
  mutate(Equity = Cash + Margin) %>%
  select(Date, Equity)
```

Here are some plots showing the after-cost performance:

```
# plot results
equity_plot <- equity %>%
  ggplot(aes(x = Date, y = Equity)) +
  geom_line() +
  labs(
    title = "Simulation of carry, momentum & breakout factors with lambda 1, tau 1",
    y = "Equity, $"
  )

weights_plot <- backtest_df %>%
  group_by(date) %>%
  summarise(
    total_weight = sum(abs(weight)),
    net_weight = sum(weight)
  ) %>%
  pivot_longer(-date, names_to = "type", values_to = "weight") %>%
  ggplot(aes(x = date, y = weight, colour = type)) +
  geom_line() +
  labs(
    title = "Total and Net Weight",
    y = "Weight",
    colour = ""
  ) +
  theme(legend.position = "bottom")

turnover_plot <- results_df  %>%
  filter(ticker != "Cash") %>%
  group_by(Date) %>%
  summarise(Turnover = 100*sum(abs(TradeValue))/initial_cash) %>%
  ggplot(aes(x = Date, y = Turnover)) +
    geom_line() +
    geom_point() +
    labs(
      title = "Turnover as % of trading capital",
      y = "Turnover, %"
    )

options(repr.plot.width = 14, repr.plot.height=12)
equity_plot / weights_plot / turnover_plot + plot_layout(heights = c(2, 1, 1))
options(repr.plot.width = 14, repr.plot.height=7)
```



Notice that we get quite long from time to time. And sometimes we're not quite fully invested. Maybe that's what you want, maybe not.

Let's repeat the process, but this time constrain the portfolio to a maximum net weight of +/- 0.5:

```r
# define our mvo function
mvo_with_costs_constrained <- function(expected_returns, current_weights, na_idxs = c(), co
  # define our weights vector as a CVXR::Variable
  weights <- Variable(length(expected_returns))
  # define an alpha term as the weighted sum of expected returns
    # again, express using linear algebra
  alpha_term <- (t(weights) %*% expected_returns)
  # define a costs term. depends on:
    # cost of trading - needs to be expressed such that it scales with expected returns
      # calculate as elementwise cost * absolute value of weights - current_weights
      # use CVXR::multiply and CVXR::abs
    # absolute distance of current_weights to our weights variable
      # the more our target weights differ from current weights, the more it costs to trade
      # this is a decent representation of fixed percentage costs, but doesn't capture mini
    # sum_entries is a CVXR function for summing the elements of a vector
  costs_term <- sum_entries(multiply(costs, abs(weights - current_weights))) # elementwise
  # define a risk term as w*Sigma*w
    # quad_form is a CVXR function for doing w*Sigma*w
  risk_term <- quad_form(weights, covmat)
  # define our objective
    # maximise our alpha less our risk term multiplied by some factor, lambda, less our cos
  objective <- Maximize(alpha_term - lambda*risk_term - tau*costs_term)
  # constrain to a maximum net weight +/- 0.5, maximum leverage 1
  constraints <- list(cvxr_norm(weights, 1) <= 1, abs(sum_entries(weights)) <= 0.5, weights
  # specify the problem
  problem <- Problem(objective, constraints)
  # solve
  result <- solve(problem)

  # return the values of the variable we solved for
  result$getValue(weights)
}
```

```r
lambda <- 1
tau <- 1
weights <- list()
errors <- list()
w0 <- rep(0, length(universe_tickers))
for( i in c(1:length(exp_returns_sim_df$date)) ) {
    # today's date
    d = exp_returns_sim_df$date[i]

    # cov estimate
    today_covmat <- covmat_list[[i]]

    # check cov matrix and exp returns vector are ticker-aligned
    # all.equal(exp_returns_sim_df %>% filter(date == d) %>% pivot_longer(-date, names_to =

    # get row of expected returns as a vector in the same order as the columns of the covar
    # contains NA at this point
    exp_rets <- exp_returns_sim_df %>% filter(date == d) %>% pivot_longer(-date, names_to =
    # get na indexes
    # we'll explicitly constrain these to get zero weight
    na_idxs <- which(is.na(exp_rets))
    # convert NA to zero
    exp_rets[is.na(exp_rets)] <- 0
    # initialise w in case we get a solver error (can retain w0 in these cases)
    w <- w0
    tryCatch({
        w <- mvo_with_costs_constrained(expected_returns = exp_rets, current_weights = w0,
    }, error = function(e) {
        # in case of solver error, log and retain existing weights
        errors[[i]] <- e
        }
    )

    # w is a one-column matrix
    weights[[i]] <- w
    w0 <- w
}

saveRDS(results, file = "lambda_1_tau_1_net_weight_constrained.rds")
```

```
# wrangle weights into a dataframe
weights_df <- purrr::map2(
  weights,
  exp_returns_sim_df$date,
  ~weights_mat_to_df(.x, .y, lambda = lambda, tau = tau, tickers = universe_tickers)
) %>%
  bind_rows() %>%
  rename("weight" = lambda_1_tau_1)

# join weights onto prices
backtest_df <- weights_df %>%
  left_join(strategy_df, by = c("ticker", "date"))

head(backtest_df)
```

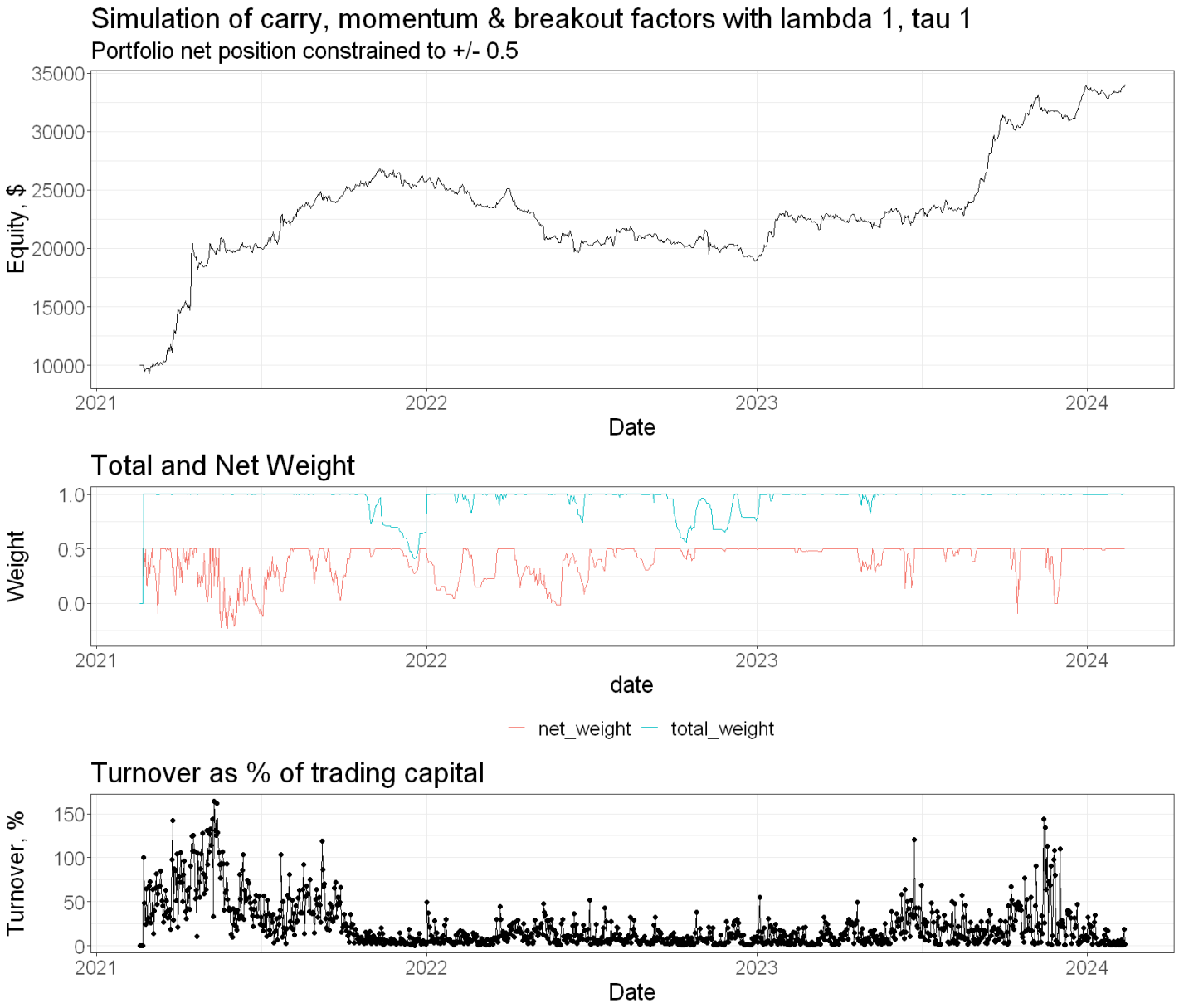
A tibble: 6 × 7

weight	date	ticker	expected_return	close	total_fwd_return_simple	funding
<dbl>	<date>	<chr>	<dbl>	<dbl>	<dbl>	<
0	2021-02-18	BTCUSDT	NA	51663.6400	0.041106415	-0.004
0	2021-02-18	ETHUSDT	NA	1910.9900	0.007813203	-0.005
0	2021-02-18	BCHUSDT	NA	702.5500	0.015704896	-0.007
0	2021-02-18	XRPUSDT	NA	0.5302	0.030592698	-0.006
0	2021-02-18	EOSUSDT	NA	4.8000	0.077784750	-0.005
0	2021-02-18	LTCUSDT	NA	225.8000	0.020896880	-0.006

```r
# simulate

# get weights as a wide matrix
backtest_weights <- backtest_df %>%
  pivot_wider(id_cols = date, names_from = ticker, values_from = c(close, weight)) %>%  # p
  select(date, starts_with("weight")) %>%
  mutate(date = as.numeric(date)) %>%
  data.matrix()

# NA weights should be zero
backtest_weights[is.na(backtest_weights)] <- 0

# get prices as a wide matrix
backtest_prices <- backtest_df %>%
  pivot_wider(id_cols = date, names_from = ticker, values_from = c(close, weight)) %>%  # p
  select(date, starts_with("close_")) %>%
  mutate(date = as.numeric(date)) %>%
  data.matrix()

# get funding as a wide matrix
backtest_funding <- backtest_df %>%
  pivot_wider(id_cols = date, names_from = ticker, values_from = c(close, funding_rate)) %>
  select(date, starts_with("funding_rate_")) %>%
  mutate(date = as.numeric(date)) %>%
  data.matrix()

# simulation
results_df <- fixed_commission_backtest_with_funding(
  prices = backtest_prices,
  target_weights = backtest_weights,
  funding_rates = backtest_funding,
  trade_buffer = 0.,
  initial_cash = 10000,
  margin = 0.05,
  commission_pct = 0.0015,
  capitalise_profits = FALSE
) %>%
mutate(ticker = str_remove(ticker, "close_")) %>%
# remove coins we don't trade from results
drop_na(Value)

margin <- results_df %>%
  group_by(Date) %>%
  summarise(Margin = sum(Margin, na.rm = TRUE))

cash_balance <- results_df %>%
  filter(ticker == "Cash") %>%
  select(Date, Value) %>%
  rename("Cash" = Value)

equity <- cash_balance %>%
  left_join(margin, by = "Date") %>%
```

```
  mutate(Equity = Cash + Margin) %>%
  select(Date, Equity)
```

```
# plot results
equity_plot <- equity %>%
  ggplot(aes(x = Date, y = Equity)) +
  geom_line() +
  labs(
    title = "Simulation of carry, momentum & breakout factors with lambda 1, tau 1",
    subtitle = "Portfolio net position constrained to +/- 0.5",
    y = "Equity, $"
  )

weights_plot <- backtest_df %>%
  group_by(date) %>%
  summarise(
    total_weight = sum(abs(weight)),
    net_weight = sum(weight)
  ) %>%
  pivot_longer(-date, names_to = "type", values_to = "weight") %>%
  ggplot(aes(x = date, y = weight, colour = type)) +
  geom_line() +
  labs(
    title = "Total and Net Weight",
    y = "Weight",
    colour = ""
  ) +
  theme(legend.position = "bottom")

turnover_plot <- results_df  %>%
  filter(ticker != "Cash") %>%
  group_by(Date) %>%
  summarise(Turnover = 100*sum(abs(TradeValue))/initial_cash) %>%
  ggplot(aes(x = Date, y = Turnover)) +
    geom_line() +
    geom_point() +
    labs(
      title = "Turnover as % of trading capital",
      y = "Turnover, %"
    )

options(repr.plot.width = 14, repr.plot.height=12)
equity_plot / weights_plot / turnover_plot + plot_layout(heights = c(2, 1, 1))
options(repr.plot.width = 14, repr.plot.height=7)
```



You can see that now the portfolio never gets more than 50% net long. You can adjust this constraint further, and add any of your own, if you like.

Conclusions

In this chapter, we demonstrated how to frame the trading problem - that is, how to navigate the tradeoffs between edge, costs, and constraints - as a mathematical optimisation problem, and how to simulate trading the output of the optimisation.

We showed how you can control the optimiser's decisions with the lambda (risk aversion) and tau (propensity to trade) parameters. We saw how different parameter values affect the optimiser's trading decisions.

It's worth reiterating that optimisation doesn't represent alpha. Your alpha is embedded in your forecasts of expected returns. Optimisation is a tool for navigating the tradeoffs involved with harnessing your expected returns, but it doesn't provide any alpha itself. This implies that the real work is in finding, quantifying and understanding your signals.

If using a risk model in your optimisation, like we did here with our covariance matrixes, it's important to note that the optimiser can be very sensitive to your covariance estimates. It's reasonable to shrink these estimates as a reflection of the inherent uncertainty.

How does the optimisation approach stack up against the heuristic no-trade buffer that we explored earlier? The heuristic is simpler to reason about and implement. But it requires a bit more fudging upstream, particularly if you wanted to incorporate a risk model.

The optimisation approach is fiddly to set up at the outset, but extremely scalable once set up. It requires some skill in modelling expected returns and covariances.

Conclusions

This book attempts to capture what I know about general statistical arbitrage trading. It includes things that I observed in my professional trading career, tips and tricks that I got from other traders, as well as things that I learned as a solo trader.

The book covered quite a lot of territory. I wanted to cover idea generation and research, as well as practical implementation considerations.

In particular, I:

- Talked about some real world considerations for pairs trading
- Introduced a general approach for doing stat arb
- Brainstormed some ideas for crypto stat arb alphas
- Explored how we might quantify and combine those alphas
- Introduced the no-trade buffer: a heuristic approach to navigating the tradeoff between uncertain alpha and certain costs
- Described how to model features as expected returns - a prerequisite for using an optimisation-based approach
- Provided a ton of examples of different optimisation problems and how they work in CVXR
- Showed you how you might simulate trading with convex optimisation and explore the lambda-tau parameter space for the classic mean-variance with costs framework

For many independent traders, the simple heuristic approach to managing turnover with a no-trade buffer parameter will be more than sufficient. In fact, given its simplicity and intuitiveness, you could argue that it is indeed the optimal approach for the time- and resource-poor independent trader.

On the other hand, if you have the time, headspace, and resources to accommodate something more sophisticated, the convex optimisation framework might be more suitable.

Both techniques have their advantages and disadvantages. In practice, I've rarely seen the convex optimisation approach implemented outside of a professional setting. It's a hard thing to set up as an independent trader, but it can certainly be done. You just have to figure out if its worth the additional overhead.

On a practical level, as an independent trader, if you were very focused on a particular statistical arbitrage basket trade and weren't trading much else, then I'd consider the convex optimisation framework. On the other hand, if you're a bit more scrappy and agile, moving into and out of trades or asset classes as opportunities come and go, then I'd almost certainly keep things as simple as possible. You don't want to invest a ton of time and effort into something, not to mention accrue a bunch of technical debt, unless you're confident you're going to be using it for some time. Especially when a simpler approach will get you 80+% of the way there.

Final word

I hope this work was useful for you and it contained some insights that help you with your trading. If it was, come and check out robotwealth.com where we help independent traders understand the fundamentals of getting an edge in the markets, as well as share our research, data and trades with the RW Pro community.