Backtesting with strand

Jeff Enos and David Kane, Strand Technologies

Introduction

Note: this document assumes familiarity with the notion of investment strategy backtesting. For an introduction, see the article *Backtests* in R-News Volume 7/1.

Evaluating an investment strategy is a multi-step process. Often the first step is to scrutinize a strategy's underlying signal, or alpha, by running a top-bottom quartile spread analysis using a tool like the R package `backtest`. A quartile analysis gives a good idea as to whether the signal is predictive of future returns. However, the analysis ignores many real-world aspects of strategy implementation and trading. For example, in a spread analysis it is assumed that we can trade immediately, regardless of actual liquidity. As a result, it is difficult to learn from a spread analysis how the performance of a strategy degrades as investment capital and portfolio size increases. In a sophisticated backtest more of an attempt is made to mimic how the strategy would be implemented in practice. This includes using optimization-based portfolio construction and trade selection, and making conservative assumptions about what trades could actually have been made in the market.

The `strand` package provides a framework for running this more realistic type of backtest. Once a strategy is defined in terms of its alpha, risk constraints, and position and turnover limits, the system simulates how the strategy would be operated day-by-day, including daily order generation and realistic trade filling.

The purpose of this vignette is to describe how to set up and run a `strand` simulation.

System overview

The `strand` system is meant to mimic a daily professional-level portfolio management process. The process involves the following steps:

- Prepare input data.
- Specify the strategy:
  - Define a **universe** of stocks in which to invest.
  - Its level of **capital** to which we want to trade.
  - The **alpha** to which we want to maximize exposure.
  - Any **exposure constraints** we want to impose, e.g., that the portfolio's exposure to any one sector must be within the range +/- 5%, or that the portfolio's exposure to a numeric factor like beta must fall within the range +/- 1%.
  - Any **liquidity constraints** on trading individual stocks, e.g., that we don't want to submit an order for a stock that is greater than 5% of the volume we expect to see for that stock in the market.
  - Any **turnover constraint** on total trading, such as setting a maximum of $2mm of trading on a single day.
- Operate the strategy by doing the following each day:
  - Process corporate actions and adjust start-of-day share values.
  - Calculate the size of the portfolio's start-of-day positions,
  - Based on these starting positions, create a list of orders by solving a constrained optimization problem where we maximize the alpha exposure of the portfolio subject to any constraints we have imposed (such as constraints on liquidity and exposure).
  - Submit these orders to the market with a limit on *participation*, i.e., how much of the day's trading volume we want to represent.
  - At the end of the day process fills and compute end-of-day position share and market values.

– Calculate performance statistics.

Running a backtest in `strand` follows the same process. First, data is prepared and organized for input into the system. Second, the strategy is defined in a configuration file. Finally, we run the simulation and analyze results. In the next few sections we'll cover each of these major steps in turn.

Configuration

All setup and configuration in `strand` is accomplished by filling in a yaml-format configuration file. The backtest in this vignette, for example, will use the file `vignettes/sample.yaml`. The configuration file contains two major sections. The `strategies` section contains entries for the strategy's alpha, any portfolio construction constraints, position size constraints, etc. (Note that it is possible to operate multiple strategies in the same simulation, but this vignette will only cover a simulation with one strategy.) The `simulator` section contains the location of input files, input file column mappings, and simulator options. There are also several top-level settings that control aspects of the simulation.

Preparing input data

Three types of input data are needed to run a `strand` simulation:

1. A security reference, or listing, of the securities in the backtesting universe.
2. Daily alpha and factor inputs for each stock in the universe.
3. Daily market data for each stock in the universe.

`strand` reads each type of input data from files stored in the language-independent binary `feather` format. In this vignette we'll be using a set of (fake) sample data available in `strand`'s GitHub repository. We assume that sample_data.tar.gz has been extracted in the `vignettes` directory of the package.

Security reference

The security reference specifies static information about each security in the backtest. Security reference data must include at least the following columns:

- `id` containing a unique security identifier.
- `symbol` for reporting.
- One column for each category used in an exposure constraint.

Below is a listing of a few rows and columns from the `secref.feather` file included in the sample dataset:

```
read_feather("sample_data/secref.feather") %>%
  select("id", "symbol", "category_1")
#> # A tibble: 500 x 3
#>   id    symbol category_1
#>   <chr> <chr>  <chr>
#> 1 101   DW     A
#> 2 102   DX     B
#> 3 103   DY     E
#> 4 104   DZ     B
#> # ... with 496 more rows
```

Shown are three columns, all of class `character`: `id`, `symbol`, and one category column `category_1`. To configure the location of the security reference input file, set the value of `/simulator/secref_data`:

```
simulator:
  secref_data:
```

```
    type: file
    filename: sample_data/secref.feather
```

The `type` field specifies that secref information should come from a file and not be passed to the simulator as a constructor parameter. The `filename` field gives the location of the file.

Alpha and factor inputs

Alpha and factor inputs are used in the daily portfolio construction process. Each data file for alpha and factor inputs must contain at least the following columns:

- `id`: the security identifier. The identifier must appear in the security reference discussed above.
- `average_volume`: an estimate of the amount of volume, in shares, that we expect to see for one day, on average. This measure of volume is used in the portfolio optimization step to ensure that our order sizes are in line with the amount we expect to be able to trade in the market. It is also used in the optimization's liquidity constraint on position size.
- One column for each alpha variable used in the simulation.
- One column for each numeric factor variable used in an exposure constraint.

The `/simulator/input_data/directory` and `/simulator/input_data/prefix` configuration options specify where the system should find these inputs. Below is the pertinent section from `sample.yaml`:

```
simulator:
  input_data:
    file:
    directory: sample_data/inputs
    prefix: inputs
```

This entry indicates that inputs files can be found in `sample_data/inputs`. By convention the data for each day has its own file with file name `prefix_YYYYmmdd.feather`. Therefore the file `inputs_20190715.feather` contains the alpha and risk values that will be used for trading on 2019-07-15.

> *Note regarding date semantics for inputs files:* The data in `inputs_20190715.feather` is used for constructing the portfolio and generating orders for trading on 2019-07-15. As a result, it is assumed that this data is known *before* trading begins on 2019-07-15. Suppose that in production, for example, trades are generated in the morning before trading begins. In our simulation, data in `inputs_20190715.feather` would contain values calculated using data through the close of trading on the previous day, 2019-07-12. How the strategy operates and any expectations around the timeliness of data delivery should dictate the contents of the inputs file for a given date.

Below are the key columns from one of the inputs files included in the sample data set:

```
read_feather("sample_data/inputs/inputs_20190715.feather") %>%
  select("id", "average_volume", "alpha_1", "factor_1",  "factor_2")
#> # A tibble: 500 x 5
#>   id    average_volume alpha_1 factor_1 factor_2
#>   <chr>          <dbl>   <dbl>    <dbl>    <dbl>
#> 1 101           945025  1.51     -0.407    0.316
#> 2 102           310188 -0.0936   -1.14     0.549
#> 3 103           649715 -0.774     0.608    1.66
#> 4 104           966942 -0.342    -0.989   -0.216
#> # ... with 496 more rows
```

The first and second columns are the `id` and `average_volume` measures described above. The column `alpha_1` is a numeric column that serves as the signal for the backtest run in this vignette. Columns `factor_1` and `factor_2` are numeric values that are used in factor constraints.

Market data

Market data is used to value positions, calculate portfolio performance, and simulate trade fills in the backtest. In the current version of the package, all prices and market values are assumed to be in a single reference currency. Each data file for market data must contain at least the following columns:

- `id`: the security identifier. The identifier must appear in the security reference discussed above.
- `close_price`: the unadjusted closing price for the security
- `prior_close_price`: the unadjusted closing price for the security on the previous day
- `adjustment_ratio`: ratio indicating any change in the adjustment basis. The ratio is the number of old shares over the number of new shares. So in the case of a 2:1 split, the adjustment ratio is 0.5.
- `volume`: the number of total shares of the security traded in the market.
- `dividend`: any cash dividend issued for the security.
- `distribution`: any other cash or equivalent per-share distribution, such as a spin-off, that does not affect the company's shares outstanding.

Below is the `/simulator/pricing_data` section of the `sample.yaml` configuration file:

```
simulator:
  pricing_data:
    type: file
    directory: sample_data/pricing
    prefix: pricing
    columns:
      close_price: price_unadj
      prior_close_price: prior_close_unadj
      adjustment_ratio: adjustment_ratio
      volume: volume
      dividend: dividend_unadj
      distribution: distribution_unadj
```

The `/simulator/pricing_data/directory` value specifies the file system location for the market data files. The value of `/simulator/input_data/prefix` indicates the prefix for each file name. So in our example, the market data for 2019-07-15 will be found in `sample_data/pricing/pricing_20190715.feather`.

The `/simulator/pricing_data/columns` section allows us to map columns in our data file to columns the backtester expects to find in the data. For example, the entry `columns/close_price=price_unadj` indicates that there is a column `price_unadj` in the data file that should be treated by the system as the required `close_price` column.

A few rows of the market data file in the sample data are shown below:

```
read_feather("sample_data/pricing/pricing_20190715.feather") %>%
  select("id", "price_unadj", "prior_close_unadj", "adjustment_ratio",
         "volume", "dividend_unadj", "distribution_unadj")
#> # A tibble: 500 x 7
#>   id    price_unadj prior_close_unadj adjustment_ratio volume
#>   <chr>       <dbl>             <dbl>            <dbl>  <dbl>
#> 1 101         23.5              23.5                 1 945013
#> 2 102         10.9              10.8                 1 310193
#> 3 103         38.8              38.9                 1 649691
#> 4 104          6.67              6.64                1 966922
#>   dividend_unadj distribution_unadj
#>            <dbl>              <dbl>
#> 1             NA                 NA
#> 2             NA                 NA
#> 3             NA                 NA
```

```
#> 4              NA                  NA
#> # ... with 496 more rows
```

As per the configuration file, the `price_unadj`, `prior_close_unadj`, `dividend_unadj`, and `distribution_unadj` columns are treated by the simulator as the `close_price`, `prior_close_price`, `dividend`, and `distrubtion` columns, respectively. There are no dividends or distributions for the securities shown, so these values are `NA`. Neither are there any splits or other changes to the securities' adjustment basis, so all `adjustment_ratio` values are 1.

> *Note regarding date semantics for market data files:* The data in `pricing_20190715.feather` is data *as-of* 2019-07-15. Therefore, some of the data could not have been known until the end of that day. For example, the `close_price` and `volume` data items are measured at the end of the trading day on 2019-07-15. Note that these semantics are in contrast with the inputs data file date semantics, where all data is assumed to have been known *before* trading beings on the date indicated in the file name.

Strategy specification

All strategy settings, including input signal and exposure constraints, are specified in the yaml configuration file. Below are the key strategy settings from the `sample.yaml` file:

```
strategies:
  strategy_1:
    in_var: alpha_1
    strategy_capital: 1e6
    ideal_long_weight: 1
    ideal_short_weight: 1
    position_limit_pct_lmv: 0.5
    position_limit_pct_smv: 0.5
    position_limit_pct_adv: 30
    trading_limit_pct_adv: 5
    constraints:
      factor_1:
        type: factor
        in_var: factor_1
        upper_bound: 0.01
        lower_bound: -0.01
      factor_2:
        type: factor
        in_var: factor_2
        upper_bound: 0.01
        lower_bound: -0.01
      category_1:
        type: category
        in_var: category_1
        upper_bound: 0.02
        lower_bound: -0.02
turnover_limit: 25000
target_weight_policy: half-way
```

In this section we'll be discussing each of the entries listed above. Note that we are only working with a single strategy in this vignette (although it is possible to use `strand` to run multiple strategies in a single simulation). This strategy is called `strategy_1`. The settings specific to `strategy_1` all fall under the `/strategies/strategy_1` entry above.

Strategy alpha

We set the input alpha for `strategy_1` by setting the `in_var` parameter:

```
strategies:
  strategy_1:
    in_var: alpha_1
```

The above indicates that in the objective function for our optimization we are maximizing the exposure of our portfolio to `alpha_1`. As we saw in the previous section, `alpha_1` is one of the columns in our inputs data.

Market value constraints

The market value of the portfolio is controlled by the following section of the configuration file:

```
strategies:
  strategy_1:
    strategy_capital: 1e6
    ideal_long_weight: 1
    ideal_short_weight: 1
```

The `strategy_capital` setting controls the amount of capital allocated to the strategy. The `ideal_long_weight` and `ideal_short_weight` parameters control the leverage for the long and short sides, respectively. The target long market value is the product of the `strategy_capital` and `ideal_long_weight` parameters, while the target short market value is -1 times the product of the `strategy_capital` and `ideal_short_weight` parameters. In our example the ideal long and short leverage values are both 1 with a strategy capital of $1mm. This means our target long market value is $1mm and target short market value is -$1mm.

Position size constraints

Position sizes are controlled by the parameters `position_limit_pct_lmv` and `position_limit_pct_smv`:

```
strategies:
  strategy_1:
    position_limit_pct_lmv: 0.5
    position_limit_pct_smv: 0.5
```

These parameters express limits on the size of positions as a percentage of the target long and short market values for the portfolio (calculated in the previous example). In our example both are set to 0.5, which means that the maximum size for a long position is 0.5% times $1mm = $5,000, and the maximum size for a short position is 0.5% times -$1mm = -$5,000.

Liquidity constraints

There are two ways in which the `average_volume` measure in our inputs data is used to impose liquidity constraints on our portfolio construction process.

First, the `position_limit_pct_adv` parameter is used to impose a position size constraint in addition to the constraints discussed in the previous section. This parameter limits the position size, in absolute value, to a percentage of the `average_volume` measure in the current day's inputs data file. In `sample.yaml` we have:

```
strategies:
  strategy_1:
    position_limit_pct_adv: 30
```

which means a position in our simulation can be no greater than 30% of our `average_volume` value. For example, suppose the price of security ABC is $100 and its average volume measure is 100,000 shares. This means the the liquidity constraint imposes a limit on the size of long positions in ABC of $3mm and a limit on the size of short positions of -$3mm.

Second, the `trading_limit_pct_adv` parameter limits the size of the order that can be generated for a stock. The idea is to size orders in line with the amount of shares that are expected to trade in the market and any limitation we are planning on imposing on participation. It's difficult to control exposures effectively if we don't do this! In the example configuration file, we have:

```
strategies:
  strategy_1:
    trading_limit_pct_adv: 5
```

which means that we can buy at most 5% or sell at most 5% of a security's `average_volume` measure on a given day. Continuing our example above, if our measurement on a given day is that ABC is trading on average 100,000 shares per day, we can buy at most 5,000 shares or sell at most 5,000 shares on that day.

Factor constraints

Factor constraints limit the amount of exposure we can have in our optimization to a given numeric value. That is, we impose an upper and/or lower bound on the product of our position weights and the numeric value. In the context of exposure constraints, a *position weight* means the signed market value of a position divided by the strategy's `strategy_capital` value.

In this vignette's example we impose factor constraints on `factor_1` and `factor_2`:

```
strategies:
  strategy_1:
    constraints:
      factor_1:
        type: factor
        in_var: factor_1
        upper_bound: 0.01
        lower_bound: -0.01
      factor_2:
        type: factor
        in_var: factor_2
        upper_bound: 0.01
        lower_bound: -0.01
```

Recall that both `factor_1` and `factor_2` must be columns present in the daily inputs data. Each constraint is configured in a separate entry in the `constraints` section that contains the following key/value pairs:

- `type`: in this case `factor` because we are constraining exposure to a numeric value.
- `in_var`: the name of the column in the inputs data file that contains the numeric value.
- `upper_bound`: the upper bound on exposure to `in_var`.
- `lower_bound`: the lower bound on exposure to `in_var`.

In our example we are limiting exposure to both `factor_1` and `factor_2` to be within +/-1%.

Category exposure constraints

Category exposure constraints are similar to factor exposure constraints. A category constraint imposes a limit on the exposure (i.e., the sum of the position weights) within each level of a category. In our example we have a single constraint on `category_1`:

```
strategies:
  strategy_1:
    constraints:
      category_1:
        type: category
        in_var: category_1
        upper_bound: 0.02
        lower_bound: -0.02
```

Here, `category_1` must be a column that appears in the security reference. As with factor constraints, the category constraint is defined in its own entry in the `constraints` section of the configuration file for `strategy_1` with the following key/value pairs:

- `type`: in this case `category` because we are constraining exposure to the levels of a category.
- `in_var`: the name of the column in the security reference that contains the category level for each security.
- `upper_bound`: the upper bound on exposure for each level of `in_var`.
- `lower_bound`: the lower bound on exposure for each level of `in_var`.

There are 6 levels in `category_1` in our security reference:

```
security_reference <- read_feather("sample_data/secref.feather")
table(security_reference[["category_1"]])
#>
#>   A   B   C   D   E   F
#>  86  83 106  92  93  40
```

The constraint above indicates that in our optimization we may have no more than +/-2% of exposure in any one of these levels.

Turnover limit

The top-level configuration entry `turnover_limit: 25000` imposes a fixed turnover constraint on the optimization. This constraint means that, if the reference currency is USD, the most that can be traded on a single day is $25,000.

Note that the system will be allowed to trade more than the turnover limit specified above if the portfolio is significantly over- or under-invested. For example, if the target gross market value of the portfolio (T) is $1mm, the current gross market value (C) of the portfolio is $1.2mm, and the turnover limit (tl) is $25,000, the effective turnover limit will be $\texttt{max}(tl, |T - C|) = \$200{,}000$.

Target weight policy

The `target_weight_policy` top-level configuration item controls how aggresively the ideal long and short weights are targeted during portfolio optimization. Currently there are two allowable values: `full` and `half-way`. When set to `full`, the ideal weights are targeted during optimization. When set to `half-way`, the optimization uses the midpoint between the current and ideal weight as the target weight. For example, suppose the current portfolio is empty, the ideal long weight is 1, and `target_weight_policy` is set to `half-way`. In this case a target long weight of 0.5 will be used during optimization.

Constraint loosening

There may be cases where, given the constraints imposed on the portfolio construction process, no solution can be found. In this scenario, the violating constraints are *loosened* in an attempt to find a solution. This loosening applies to factor and category exposure constraints.

For example, suppose the current exposure to level `A` of `category_1` is 3%, and we have set a 2% exposure limit on exposure to `A`. If no solution is found, the optimization is re-run with a limit that is 50% as strict. That is, we set the limit to the midpoint between the current exposure and the constraint limit. In our example, the midpoint between the current exposure and the limit is 2.5%. If no solution is found with the new limit of 2.5%, the constraint is loosened again by 50%, moving the limit to 2.75%. If no solution is found after this second round of loosening, the constraint limit is set to the current exposure, in this case 3%, to ensure that the constraint can be satisfied.

Simulator settings

In the previous section we discussed how to define the strategy that we want to backtest, in terms of the input signal and portfolio construction constraints. In this section we will cover how to configure different aspects of the backtest that are not related to portfolio construction.

Date range

The top-level items `from` and `to` control the starting and ending dates for the backtest:

```
from: 2019-01-02
to: 2019-12-31
```

In this vignette we will run a simulation over 1 year of daily data, beginning on January 2, 2019 and ending on December 31, 2019.

Solver

The top-level parameter `solver` controls which linear optimization toolkit is used to solve the portfolio construction constrained optimization problem:

```
solver: glpk
```

Currently there are two options: `glpk` to use the GNU Linear Programming Kit, and `symphony` to use the COIN-OR SYMPHONY solver.

Participation rate limit

The simulator setting `fill_rate_pct_vol` controls what percentage of the observed volume for a security on a given day can be used to fill our order. We call this the *fill rate* or *participation rate* for the backtest. In `sample.yaml` we set this value to 4%:

```
simulator:
  fill_rate_pct_vol: 4
```

As an example, suppose on 2019-07-15 that the optimization process for `strategy_1` generates an order to buy 5,000 shares of security ABC. But suppose ABC only trades 100,000 shares on that day. Because we have set a 4% limit on participation, `strategy_1` will only be able to buy 4,000 shares, despite generating an order to buy 5,000.

Transaction costs

The value of the parameter `transaction_cost_pct` determines the fixed transaction costs, as a percentage of traded notional, for the strategy. In `sample.yaml`, we have:

```
simulator:
  transaction_cost_pct: 0.1
```

which means that we are charging a transaction cost penalty of 10bps of traded notional. For example, if we trade $10,000 worth of stock ABC on 2019-07-15 we incur a transaction cost of $10 for that day.

Financing costs

Setting `financing_cost_pct` controls the financing rate used for the backtest. Financing for day t is applied to the starting notional of the position and ignores any trading on day t. Financing is calculated using a standard 360-day-count methodology and is triple-charged on Mondays. The financing charge is set to 1% for the vignette's backtest:

```
simulator:
  financing_cost_pct: 1
```

For example, if we have a position in ABC valued at $100,000 at the beginning of Monday, July 15, 2019, we would apply a financing charge of $3 \times \frac{0.01 \times \$100{,}000}{360} = \$8.33$ for the position on that day.

Running the simulation

At this point we have covered all of the setup required to run the backtest. We have prepared our data, including the security master and daily inputs and market data. We have filled in the configuration file to specify the strategy and control different aspects of the simulator. We are ready to run the backtest.

The `strand` package is implemented using the `R6` OOP system. To run the backtest, we create a `Simulation` object by passing the path to the yaml configuration file to the constructor. Then we call the method `run()`:

```
sim <- Simulation$new("sample.yaml")
res <- sim$run()
```

Viewing summary statistics

When the backtest is finished, we can call methods to summarize and plot the results. For example, the `overallStatsDf()` method returns a data frame of key statistics:

```
res$overallStatsDf()
#>                          Item  Gross       Net
#> 1                   Total P&L -8,773   -35,912
#> 2       Total Return on GMV (%)  -0.4      -1.9
#> 3  Annualized Return on GMV (%)  -0.4      -1.8
#> 4            Annualized Vol (%)   0.6       0.6
#> 5             Annualized Sharpe -0.71     -2.97
#> 6                       Avg GMV       1,992,131
#> 7                       Avg NMV             -94
#> 8                     Avg Count             387
#> 9           Avg Daily Turnover          27,092
#> 10     Holding Period (months)             7.0
```

This display shows that, gross of transaction and financing costs, the strategy had a return of -0.4% on gross market value (GMV) and had an annualized Sharpe ratio of -0.71. Net of costs, the strategy's return was -1.9% with a Sharpe of -2.97. The average gross market value of the portfolio (GMV) was 1,992,131, while the average net market value (NMV) was -94, as expected given that the strategy's target market values are $1mm long, -$1mm short. On average there were 387 positions in the portfolio. The average daily turnover (gross market value of trading) was 27,092, implying a holding period of 7.0 months.

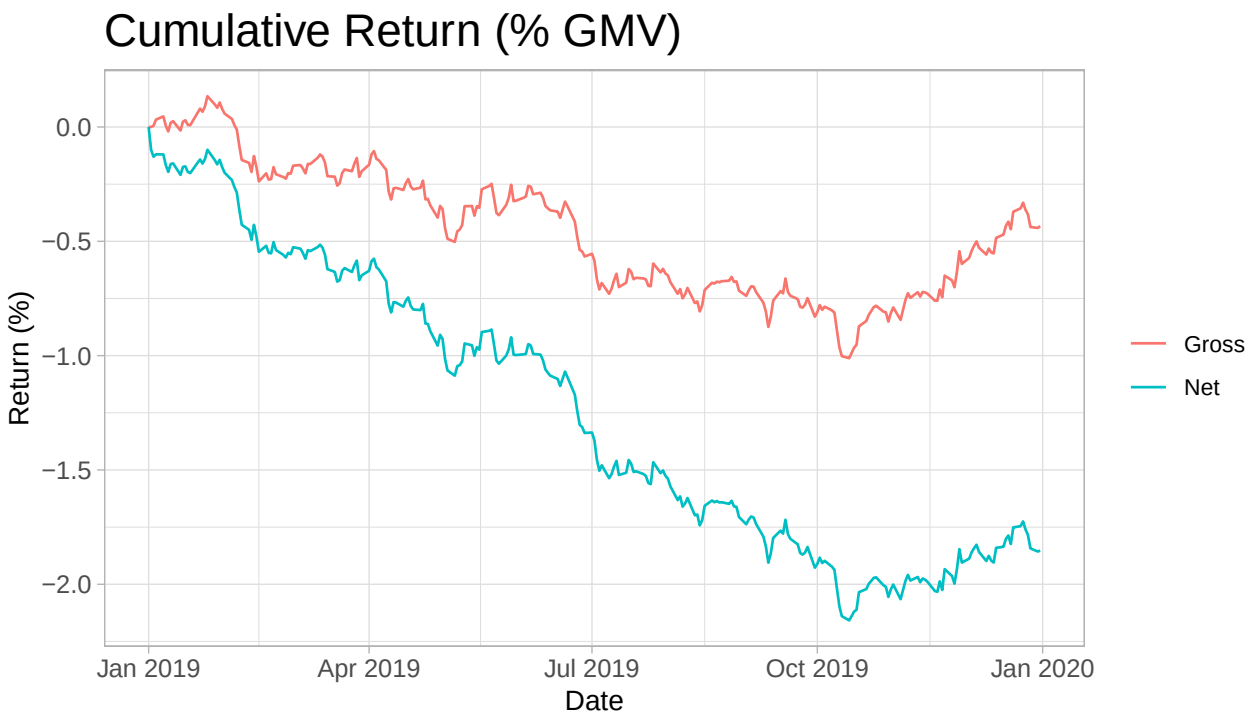
Plotting results

There are several methods that can be used to visualize backtest results.

Portfolio returns

The `plotPerformance()` method plots gross and net return on GMV over time:

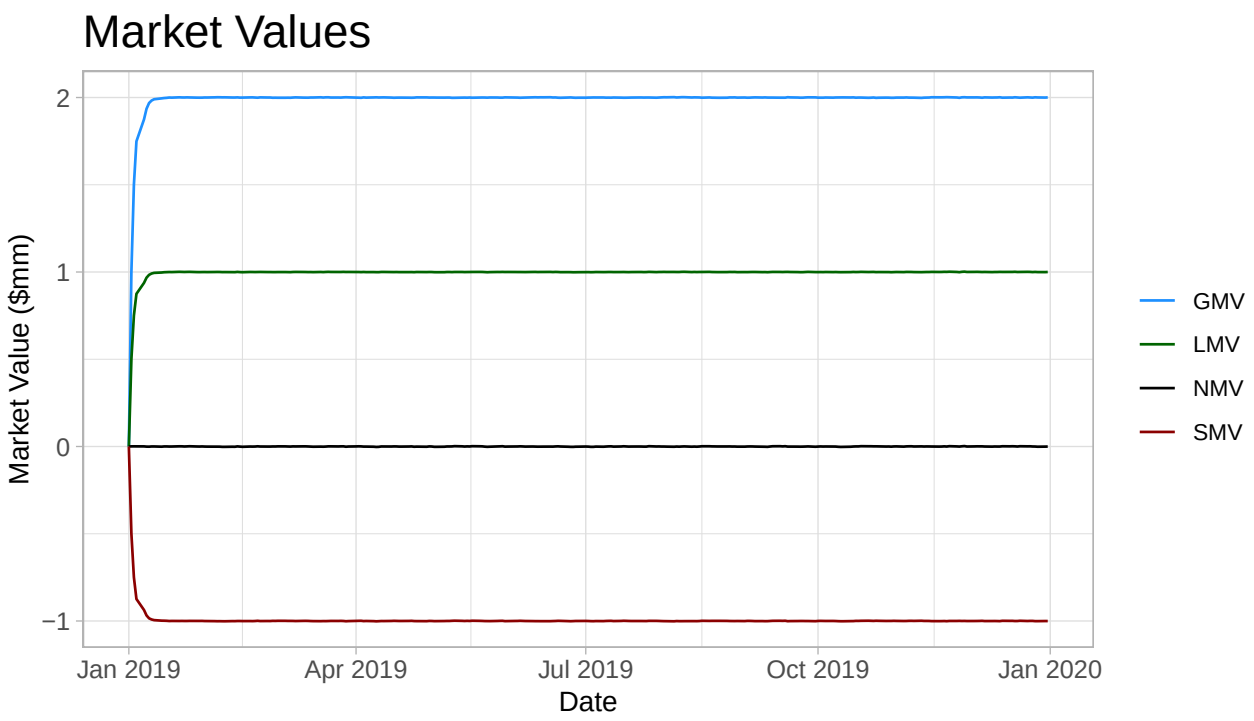
```
res$plotPerformance()
```



Market values

The `plotMarketValue()` method plots gross market value (GMV), net market value (NMV), long market value (LMV) and short market value (SMV) over time:

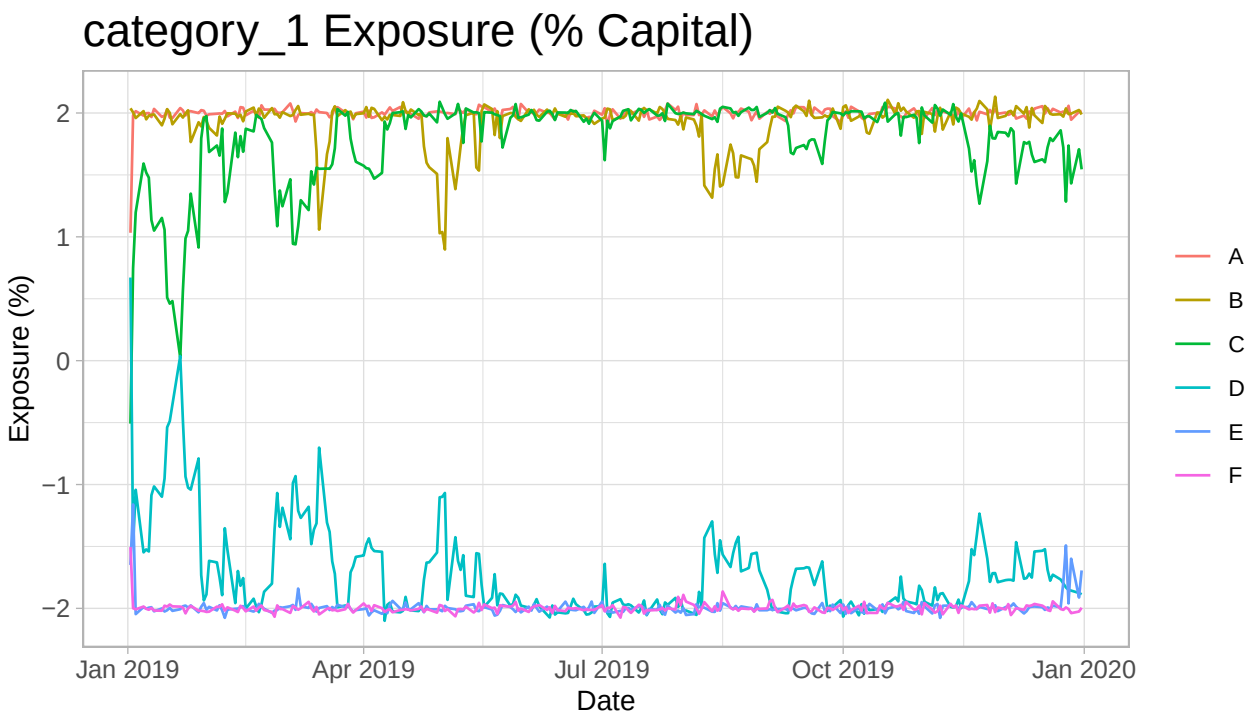
```
res$plotMarketValue()
```



Category exposures

The `plotCategoryExposure()` method shows the exposure over time within each level of a given category. Below we plot the exposure within the levels of `category_1`, which in our backtest has an exposure constraint of +/-2%:

```
res$plotCategoryExposure("category_1")
```



Note that there are some cases where the exposure to a level of `category_1` falls outside of +/-2% despite

the category exposure constraint we impose during portfolio construction. This can be due to the following:

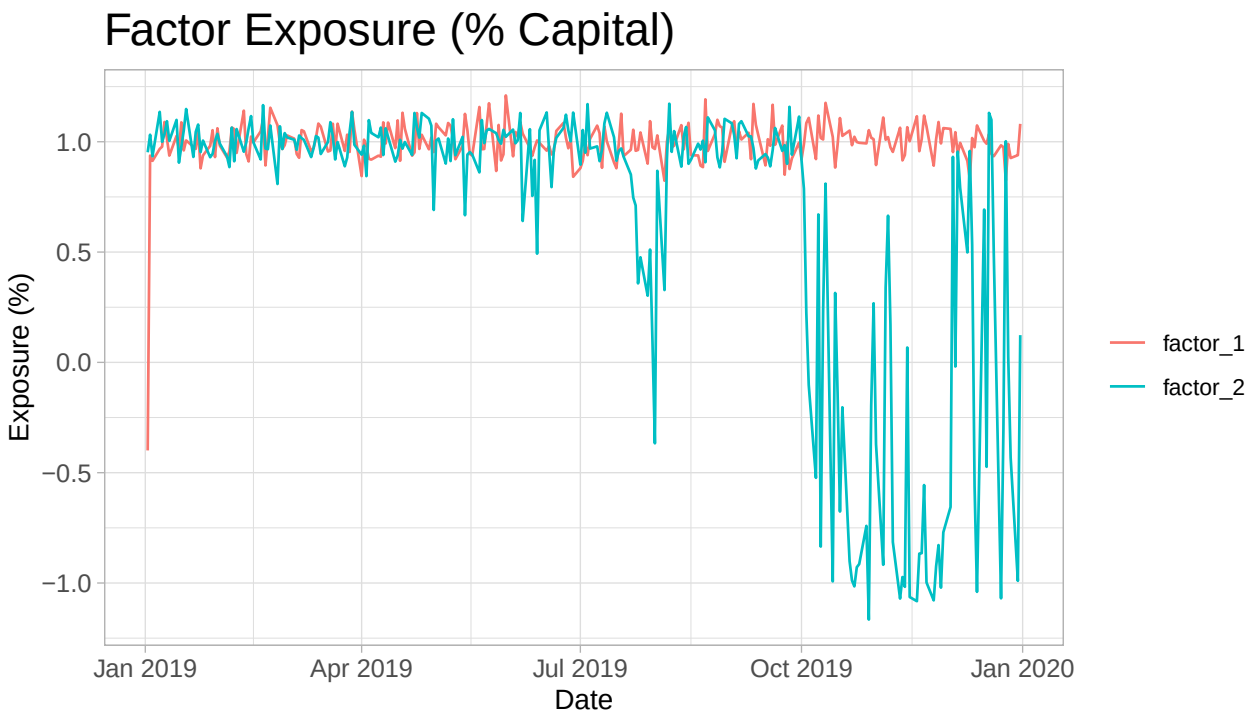
- Prices for securities in the level of the category are rising (in the case of an exposure that is too positive) or falling (in the case of an exposure that is too negative). The portfolio construction step uses prices as of the *start* of the day, while the plot above shows exposures at the *end* of the day. So even if constraints are within bounds using starting market values they could be out of bounds at the end of the day due to price movement.
- Lack of liquidity. The trades that we need to make to bring an exposure back within bounds could be left unfilled due to a lack of liquidity. Recall that we configured our backtest to only allow fills up to 4% of the number of shares traded in the market.
- Loosened constraints. It could be the case that no set of trades can be found to bring an exposure that has drifted back within bounds and that the constraint needed to be loosened.

Exploring these scenarios is possible by looking at lower-level backtest results but is outside the scope of this vignette.

Factor exposures

The `plotFactorExposure()` method shows the portfolio exposure over time to one or more factors. Below we plot the exposure to `factor_1` and `factor_2`, each of which in our backtest has an exposure constraint of +/-1%:

```
res$plotFactorExposure(c("factor_1", "factor_2"))
```



Here we can also see spikes of exposure outside the constrained range +/-1%. As discussed in the previous section, price movement, lack of liquidity and constraint loosening are possible explanations for these spikes in end-of-day exposure. In the case of factor constraints, another possible explanation is a significant day-over-day change in factor values. Again, exploring these scenarios is possible by diving more deeply into the backtest's result data, but is outside the scope of this document.