Multi-asset backtesting pipeline

-Antonis Belias - Athens University of Economics and Business, Statistics department

The contents of this research

In this research paper I will go over my personal Multi-asset backtesting engine written in Python, its architectural design, the choices I made when designing it and the benefits that every quant aspirant can gain by using something equivalent. I will also include tests of how my personal strategy which is included in this pipeline performs against the S&P 500. Some of its key features are:

- It's built modularly, and is therefore easily extensible and customizable.
- Works with YAML config files for rapid, optimized testing.
- Is written with vectorized code where possible to efficiently work through multiple years of data for multiple stocks.
- Supports multiple assets.
- Includes portfolio-level weighting of the different assets based on the volatility and trends of each one.
- Automates plotting (the plots get shown and saved automatically) & performance metrics (Sharpe, Sortino, drawdown, calmar, annualized volatility, annualized returns)
- Includes realistic assumptions about fees and slippage.
- Prioritizes a strict avoidance of look-ahead bias.
- Includes a personal hybrid strategy as an example, but that's easily switchable with one's own strategies.

A general overview

This is an end to end pipeline:

- It starts off with the main execution file calling the Data handling module. That, in return, reads the names of the assets listed in the config files, downloads the price of the assets from Yahoo finance, and saves each one locally in a CSV file, in a designated data folder.
- Then, the main execution file calls the strategy module for every asset, which then generates signals for each day of each asset (1 = Long , 0 = Sell, -1 = Short). In the same loop, a risk module is called, whose job is to overwrite the signal of days that appear to be overly dangerous to 0 (Sell).
- The last piece of the strategic aspect of this pipeline is the weight distribution module, which assigns a normalized weight to each asset each day, that scales inversely with volatility and gets boosted depending on how strong the current trend is (if said trend is positive).

- Finally, the performance metrics of the entire portfolio are calculated. This includes: Sharpe ratio, Sortino ratio, Calmar ratio, annualized returns, annualized volatility, total gains (%), max drawdown, as well as statistics like the win rate of trades, the loss rate, the average profits per successful trade and the average losses per losing trade.

The architecture

```
Multi_asset_backtesting_pipeline/
│-config/
│   ├assets.yaml
│   └backtest.yaml
│
│-data/
│    ├raw/
│    │ └AAPL_2010-01-01_2025-01-01.csv
│    ├processed/
│    │ └AAPL_2010-01-01_2025-01-01.csv
│
│-engine/
│    ├asset_weight_assignment.py
│    ├data_handler.py
│    ├performance.py
│    ├risk.py
│    ├runner.py
│
│-reports/
│    ├plots/
│    │
│      ├2021...2025.png
│      └plotting.py
│
│-strategies/
│   ├hybrid-strategy.py
│
│-Dockerfile
│
│-requirements.txt
│
│-README.md
```

Explaining the architecture in more detail:

There are 5 main moving parts to this pipeline.

1) The config files:

Every single file that contributes to the function of this pipeline takes their constants from the config files. This means that if one wanted to change what assets the engine operates on, what the initial investment the engine had access to was, what the rate of fees and slippage was, on what time-periods the strategy operates in etc, all of it is possible without any unnecessary hustle by simply changing the necessary values in the config files. Do you want the strategy you're testing to start with say $50,000.00 instead of the $100,000.00 it starts off with by default? It is as simple as just changing the initial_investment variable in the backtest.yaml file and every single module will function with the new value in mind.

2) The data handling and caching:

The data folder includes raw data directly downloaded from Yahoo Finance. The data then gets cleaned up (dropping missing values, calculating percentage returns to make the strategy implementation more straightforward) and gets saved in the processed folder. The entire process is handled by the 'data_handler' module in the 'engine' folder. To provide a benchmark, the module also downloads S&P 500 data for the same period, allowing direct comparison of the strategy's results.

3) The signal generation:

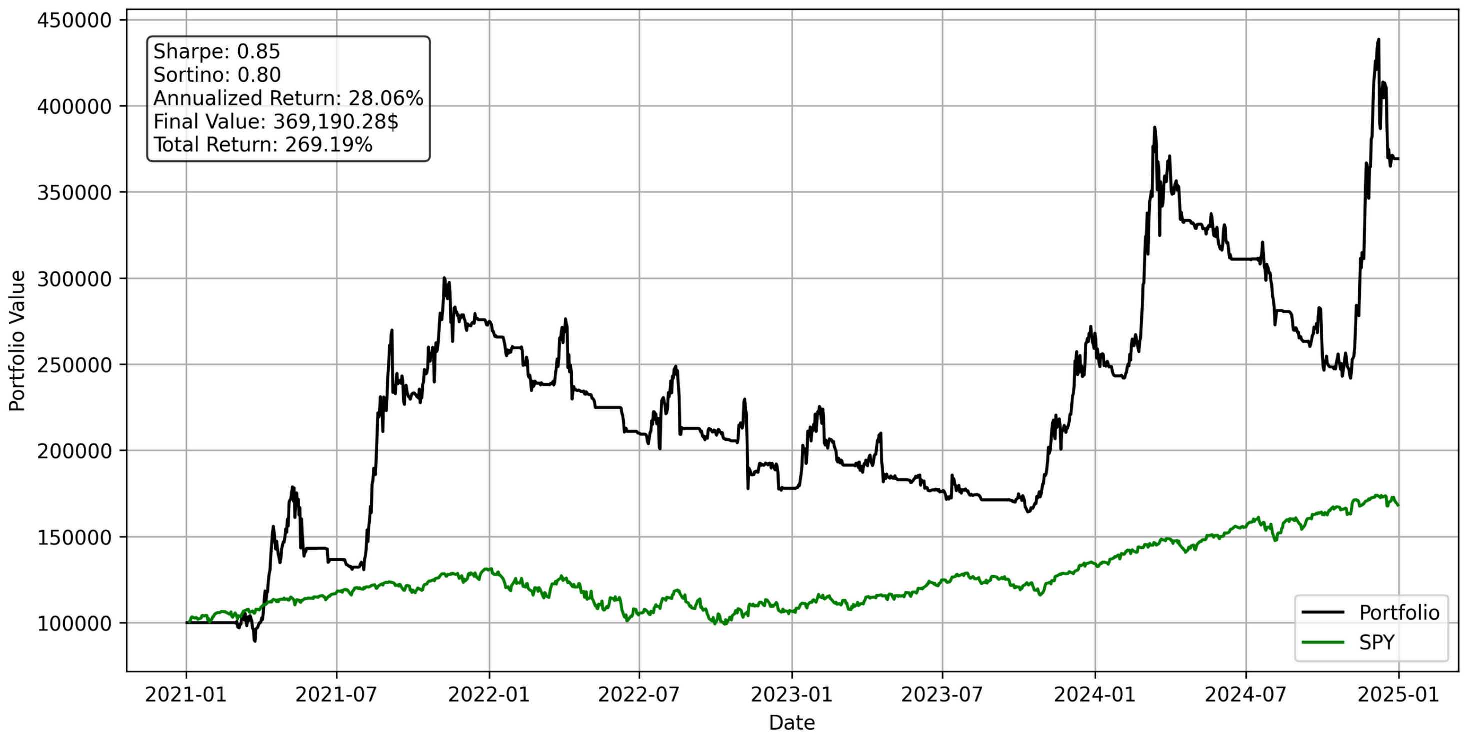
For this job, that importantly is done per-asset and not across the whole portfolio, the ones responsible are the 'hybrid_strategy' module and the 'risk' module. The hybrid strategy I have been talking about consists of a combination of Simple Moving Averages and Mean Reversion. The choice on which one of the two to use is based on the amount of volatility we are experiencing in the current regime as well as the direction of the trend we are in. The 'risk' module's job is to overwrite the signals that the strategy generates on three conditions: a) The volatility of the current regime is higher than the standard, b) we are currently on a downtrend, and c) the drawdown we are experiencing is at least 15% down from the portfolio's current maximum. On those days the signal gets turned to 0, independently of what the strategy actually considered to be right. It is functionally a stop-loss/killswitch.

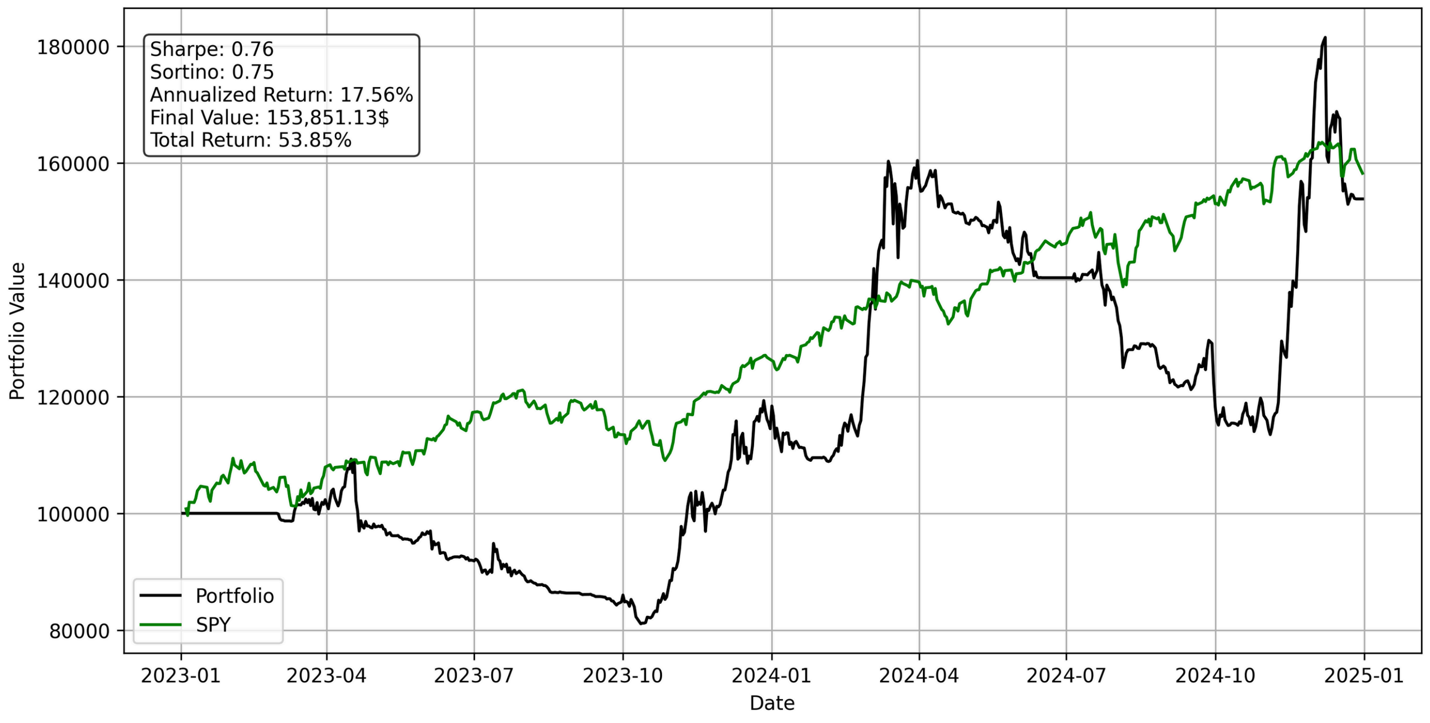
4) The simulated execution over historical data:

The 'performance' module and the 'asset_weight_assignment' are the ones responsible for this. First, for every day of trading, every asset gets assigned a normalized weight that scales inversely with the volatility that asset is experiencing. If an asset is in an upwards trend, it also gets a boost from the intensity of said trend. After every asset has a weight assigned for every day, the 'performance' module uses those weights effectively as the position size that every asset has. Importantly, it dynamically calculates the fees and slippage by calculating the difference in the position sizing of an asset from yesterday to today, it doesn't simply apply them once per trade. This same module then creates a dataframe with the aggregated portfolio wide results, which include the PnL of the portfolio as well as a dictionary of the performance metrics listed above. (In the general overview.)

5) The final results:

The results get printed by the main ‘runner’ module, and the ‘plotting’ module creates a graph comparing the current strategy to the S&P 500 from the same period. The plots get automatically saved in the reports/plots folder. Part of the performance metrics like the Sharpe and Sortino ratios also show up on the top left of the plots to act as a more complete stand-alone result.





These two plots above provide us a visual example of the kind of graphs the module generates.

Weaknesses of the pipeline and the strategy:

- The weighing logic works well only with homogeneous assets, because it relies on the idea that assets change volatility depending on the market,and it isn’t capable of understanding the concept that some assets have fundamentally different volatility ranges by default.
- The risk module is inconsistent, sometimes stopping a strategy from riding big bull trends and not entirely preventing big losses in bear trends, although when I tested the strategy without it the performance definitely got worse, so it is contributing, just not enough.
- The data is saved as CSVs instead of parquets, but I couldn’t get the extension to work and prioritized shipping a finished product over a fully optimized one.
- The way data is saved is inefficient. If I download Apple’s stock prices from 2021 to 2025, and then I want to download the prices of the same stock from 2022 to 2025, even though the data is technically saved in my folder as a subsection of a piece of data I saved in the past, it will still download it and not read the part of the data it needs from the disk.
- The actual strategy is mediocre at best. It is too conservative with less volatile assets and leaves volatile ones overly loose. I had to make due with very simplistic individual parts as at the time of writing this I haven’t started university. I needed to work around my lack of formal advanced mathematics knowledge.

Potential future fixes (For each previously mentioned weakness):

- (Weighing logic) There could be an extra attribute per asset which includes its class so that there are slightly different calculations for different assets (commodities, crypto, stocks etc) depending on how we actually expect that asset to act in terms of its volatility.
- (Lacking risk management) More advanced risk modeling using adaptive thresholds or potentially using ML to predict sudden drops
- (inefficient data handling) Making sure that if the data we are looking for is saved, even if the file it is in includes extra data from other time periods, we still read it from the disk and just filter out the part of the CSV we don’t need.
- (Underwhelming performance) Including more sophisticated logic with more advanced mathematics concepts.

The final product:

The final product is a very good foundation for any strategy I and anyone using this decides to include in the future. It is lacking in multiple areas as mentioned above, but it will definitely serve as a productivity boost and a useful tool to people who want the ability to easily test their strategies across multiple assets using realistic market conditions. The entire pipeline is around 600 lines, and can be found at my GitHub.