Published in Towards Data Science

Yao Lei Xu Follow

Jul 24, 2020 · 8 min read · Member-only

Backtest Trading Strategies with Pandas — Vectorized Backtesting

A fast and efficient approach for evaluating trading strategies

Aha! You found a pattern in the market that can potentially make you the next Warren Buffett… Now what? After designing a new trading strategy that can systematically profit from that pattern, it is now the time to ask yourself: "What if I had this strategy five years ago?".

Today, I will discuss how vectorized backtesting can help you "answer" that question, how to implement it using Pandas, and what subtleties should you be aware of. For those who have some experience in this area, feel free to jump to section 5 directly.

(You can find the full code and additional resources here)

1. What Is Backtesting?

> *Backtesting is used to simulate the past performance of a trading strategy.*

The core concept of backtesting is to simulate a given trading strategy by going back in time and execute the trades as if you were in the past. The generated profits are often compared against a benchmark performance through some metrics (e.g. max drawdown, Sharpe ratio, annualized returns, etc.). Depending on the nature of the strategy, different benchmarks and metrics should be used, but that is a whole topic on its own so I won't be going into details here.

2. What Backtesting Is Not

> *Backtesting is not an exact indicator of past performance and should not be used as a research tool, especially in inexperienced hands.*

Funnily enough, backtesting is not a good indicator of how rich you would be today if you could travel back in time. Time-travelling aside, it is virtually impossible to replicate past performance exactly, since there are a number of factors that are just way too complicated to be modelled exactly (slippage, market impact of trades, etc.). In addition, in inexperienced hands, a backtest can be full of biases (e.g. look-ahead bias, survivor bias, etc.) which can make a strategy look much better than reality (see section 5.3).

Finally, backtesting is not a research tool. Do not randomly change your strategy parameters just because it looks better in the backtest. You should consider the historical performance of the market as one of many possible realizations of a random variable. Fitting your parameters to the backtest without sound economic logic will inevitably lead to over-fitting.

> *Vectorized backtesting is drastically faster than event-driven backtesting and should be used for the explorative stage of strategy research.*

There are two main methods for backtesting a strategy: (i) event-driven and (ii) vectorized. An event-driven backtest often involves the use of a loop that iterates over time, simulating an agent that sends orders depending on the market signals. This loop based approach can be extremely flexible, allowing you to simulate potential delays in orders execution, slippage costs, etc.

A vectorized backtest instead gathers strategy related data and organizes them into vectors and matrices, which are then processed through linear algebra methods (the thing your computer is really good with) to simulate past performance.

Due to the expressive power of linear algebra, vectorized backtests are DRASTICALLY FASTER than event-driven backtests, although it lacks in terms of flexibility. Usually, vectorized backtests are better suited for an initial explorative strategy analysis, whereas event-driven backtests are suited for a more in-depth analysis.

4. A Note on Log-Returns

> *Strategy performance can be computed using the cumulative sum of log-returns.*

Before diving into the python code, it is important to understand the concept of log-returns, which is defined as the difference in log-value of an asset during time interval (*t-1, t*):

$$r_t = log(\frac{p_t}{p_{t-1}}) = log(p_t) - log(p_{t-1})$$

Log-returns are incredibly useful in quantitative finance for a number of reasons (stationarity, log-normality, etc.), but for the purpose of backtesting, we will use its time-additive property to compute cumulative returns (i.e. past performance).

For instance, consider the log-returns of an asset over the days *(t=3,2,1)*. To compute the total return of that asset over the entire period, we simply sum-up all daily returns *(r[3]+r[2]+r[1])*. This follows since the log price information gets cancelled out as shown below:

$$\begin{aligned} & r_3 + r_2 + r_1 \\ & = (log(p_3) - log(p_2)) + (log(p_2) - log(p_1)) + (log(p_1) - log(p_0)) \\ & = log(p_3) - log(p_0) \\ & = log(\frac{p_3}{p_0}) \end{aligned}$$

The above argument applies to your strategy too. For instance, if your strategy generated log-returns (*r[0], r[1], …, r[T]*) over *T* days, then the backtest of the strategy can be computed through a simple cumulative sum of the log-returns followed by the exponential function.

5. Vectorized Backtesting with Pandas

5.1. Single Asset Backtest

Let's say we want to backtest the performance of a price moving average crossover strategy, where you are long the underlying asset if the short-term price average is above the long-term price average and vice-versa. We can compute the strategy performance in 6 steps:

(1) Compute the log-returns of the underlying asset. In this example, we are looking at the log-price differences of the AAPL stock, which is stored in a Pandas Series indexed by time.

```
rs = prices.apply(np.log).diff(1)
```

(2) Compute the signal

```
w1 = 5 # short-term moving average window
w2 = 22 # long-term moving average window
ma_x = prices.rolling(w1).mean() - prices.rolling(w2).mean()
```

(3) Translate the signal to actual trading positions (long or short)

```
pos = ma_x.apply(np.sign) # +1 if long, -1 if short

fig, ax = plt.subplots(2,1)
ma_x.plot(ax=ax[0], title='Moving Average Cross-Over')
pos.plot(ax=ax[1], title='Position')
```

Image by author

(4) Compute the strategy returns by adjusting the log-returns of the underlying asset according to the positions (the calculation of the strategy returns is not exact, but is often a good enough approximation at a practical

```
my_rs = pos.shift(1)*rs
```

(5) Compute the backtest by using the cumulative sum and applying the exponential function.

```
my_rs.cumsum().apply(np.exp).plot(title='Strategy Performance')
```

Image by author

5.2. Multi Assets Backtest

Similarly to the single asset case, we can compute the backtest for a portfolio of assets using Pandas. The only difference here is that we are working with a Pandas DataFrame instead of a Pandas Series.

Let's first compute the signals and the positions for each of the asset as shown in the code below. Notice that we have scaled the positions such at any given time, the total sum of the portfolio positions do not exceed 100% (i.e. assuming no leverage is used).

```
rs = prices.apply(np.log).diff(1)

w1 = 5
w2 = 22
ma_x = prices.rolling(w1).mean() - prices.rolling(w2).mean()

pos = ma_x.apply(np.sign)
pos /= pos.abs().sum(1).values.reshape(-1,1)

fig, ax = plt.subplots(2,1)
ma_x.plot(ax=ax[0], title='Moving Average Cross-Overs')
ax[0].legend(bbox_to_anchor=(1.1, 1.05))
pos.plot(ax=ax[1], title='Position')
ax[1].legend(bbox_to_anchor=(1.1, 1.05))
```

Image by author

We can look at how the strategy performed in terms of individual assets:

```
my_rs = (pos.shift(1)*rs)

my_rs.cumsum().apply(np.exp).plot(title='Strategy Performance')
```

Image by author

We can also look at how the strategy performed overall at a portfolio level:

Image by author

5.3. A Quick Note on Look-Ahead Bias

In the above examples, the the position vector/matrix is always shifted by 1 time step. This is done to avoid look-ahead bias, which can make a strategy look much better than reality.

The idea is that: at time t, you have a signal that contains information up to and including t, but you can only realistically trade that signal at $t+1$ (e.g. if your signal contains earnings information of a company released on Monday night, then you can only trade that information on Tuesday morning).

As shown below, the look-ahead bias can make the strategy look way more promising than reality (especially for a "momentum" strategy).

```
my_rs1 = (pos*rs).sum(1)
my_rs2 = (pos.shift(1)*rs).sum(1)
```

Image by author

5.4. Evaluate Robustness to Delays

With vectorized backtesting, it is much easier to evaluate a strategy's robustness to delays (i.e. opening your trades T time steps after the signal). This is however an analysis that is more useful at a high-frequency trading level, where the delays can severely impact the performance.

```
lags = range(1, 11)
lagged_rs = pd.Series(dtype=float, index=lags)

for lag in lags:
    my_rs = (pos.shift(lag)*rs).sum(1)
    my_rs.cumsum().apply(np.exp).plot()
    lagged_rs[lag] = my_rs.sum()

plt.title('Strategy Performance with Lags')
plt.legend(lags, bbox_to_anchor=(1.1, 0.95))
```

Image by author

5.5. Simulating Transaction Costs

Transaction costs are often quoted in fixed percentages. To incorporate these in the backtest, we can simply compute how much did the portfolio change between successive time steps, then subtract the related transaction costs directly from our strategy returns.

```
tc_pct = 0.01 # assume transaction cost of 1%
```

```
my_rs1 = (pos.shift(1)*rs).sum(1) # don't include costs
my_rs2 = (pos.shift(1)*rs).sum(1) - my_tcs # include costs

my_rs1.cumsum().apply(np.exp).plot()
my_rs2.cumsum().apply(np.exp).plot()
plt.legend(['without transaction costs', 'with transaction costs'])
```

Image by author

I hope you enjoyed the article! Follow me if you would like to see more content like this.

Also, check out my website for the full code and additional resources.

Note from Towards Data Science's editors: *While we allow independent authors to publish articles in accordance with our rules and guidelines, we do not endorse each author's contribution. You should not rely on an author's works without seeking professional advice. See our Reader Terms for details.*

ANNOUNCEMENT: SPORTE X

Diversification