6

Vector-Based Backtesting with VectorBT

Now that we've touched on the fundamental Python tools for algorithmic trading, we'll move to the next phase of the workflow: backtesting. Since most strategies will not consistently make money, and those that do may only make money for a short time, quickly iterating through ideas is critical. This chapter demonstrates how to use vector-based backtesting for the simulation and optimization of trading strategies.

VectorBT is a high-performance, vector-based backtesting framework that allows for efficient evaluation of trading strategies by processing entire time-series data arrays at once, rather than one data point at a time. This method significantly speeds up backtesting operations, making it ideal for rapid strategy iteration. The technique is highly customizable, enabling traders to fine-tune parameters and assess multiple strategies concurrently. We will explore the optimization of these strategies with VectorBT.

In this chapter, we will explore the following recipes:

- Building technical strategies with VectorBT
- Conducting walk-forward optimization with VectorBT
- Optimizing the SuperTrend strategy with VectorBT Pro

Building technical strategies with VectorBT

This recipe introduces you to the powerful vector-based backtesting library VectorBT. One of the most compelling advantages of using VectorBT is its speed in running simulations. Whether you are testing a single strategy or optimizing across a multi-dimensional parameter space, VectorBT's performance is optimized to deliver results in a fraction of the time traditional methods would require.

Built on top of well-established libraries such as pandas, NumPy, and Numba, VectorBT seamlessly integrates into the data science ecosystem. It leverages pandas for its DataFrame structure, which is familiar to most quants. NumPy's numerical computing abilities provide the mathematical backbone, ensuring that heavy calculations are performed efficiently. However, the real game-changer is Numba, a **Just-In-Time** (**JIT**) compiler that translates Python functions to optimized machine code at runtime. Thanks to Numba, VectorBT can execute loops and mathematical operations at speeds comparable to those of a low-level language, all while allowing you to write in Python.

This recipe introduces VectorBT by building a simple moving average crossover strategy (the "Hello World" of trading strategy development).

Getting ready

There are two versions of VectorBT: a free open source library and a more full-featured Pro version. This recipe uses the free open source version.

You can install it using **`pip`**:

```
pip install vectorbt
```

How to do it…

VectorBT can be considered an extension of pandas. Let's see it in action:

1. Import the libraries needed for the analysis:

```
import pandas as pd
import vectorbt as vbt
```

2. Download data using the built-in data downloading class:

```
start = "2016-01-01 UTC"
end = "2020-01-01 UTC"
prices = vbt.YFData.download(
```

```
    ["META", "AAPL", "AMZN", "NFLX", "GOOG"],
    start=start,
    end=end
).get("Close")
```

The result is the following pandas DataFrame with the closing prices for the selected symbols:

symbol	META	AAPL	AMZN	NFLX	GOOG
Date					
2016-01-04 05:00:00+00:00	102.220001	24.009066	31.849501	109.959999	37.091999
2016-01-05 05:00:00+00:00	102.730003	23.407415	31.689501	107.660004	37.129002
2016-01-06 05:00:00+00:00	102.970001	22.949339	31.632500	117.680000	37.181000
2016-01-07 05:00:00+00:00	97.919998	21.980770	30.396999	114.559998	36.319500
2016-01-08 05:00:00+00:00	97.330002	22.096996	30.352501	111.389999	35.723499

Figure 6.1: A pandas DataFrame with market data

3. Build the moving average indicators using VectorBT's built-in MA class:

```
fast_ma = vbt.MA.run(prices, 10, short_name="fast")
slow_ma = vbt.MA.run(prices, 30, short_name="slow")
```

4. Next, we'll find the entry positions. In this example, these occur when the fast-moving average crosses above the slow-moving average:

```
entries = fast_ma.ma_crossed_above(slow_ma)
```

The result is the following pandas DataFrame containing boolean values where trades should be entered:

fast_window	10				
slow_window	30				
symbol	META	AAPL	AMZN	NFLX	GOOG
Date					
2016-01-04 05:00:00+00:00	False	False	False	False	False
2016-01-05 05:00:00+00:00	False	False	False	False	False
2016-01-06 05:00:00+00:00	False	False	False	False	False
2016-01-07 05:00:00+00:00	False	False	False	False	False
2016-01-08 05:00:00+00:00	False	False	False	False	False

Figure 6.2: A pandas DataFrame with locations of entry positions

5. Do the opposite for the exit positions:

```
exits = fast_ma.ma_crossed_below(slow_ma)
```

6. Run the backtest using the entry and exit signals:

```
pf = vbt.Portfolio.from_signals(prices, entries,exits)
```

7. Visualize the mean daily return for each symbol:

```
pf.total_return().groupby(
    "symbol").mean().vbt.barplot()
```

The result is an interactive Plotly bar chart with the mean daily returns for each symbol:

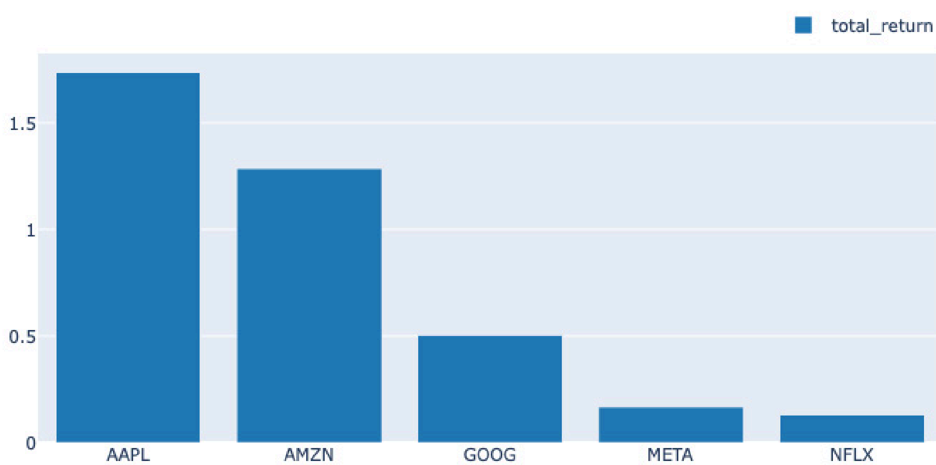

Figure 6.3: Visualizing the mean daily returns of each symbol

8. Inspect the returns for each symbol by simply holding each throughout the analysis period:

```
(
    vbt
    .Portfolio
    .from_holding(
        prices,
        freq='1d'
    )
    .total_return()
    .groupby("symbol")
    .mean()
    .vbt
    .barplot()
)
```

The result is the following bar chart with the mean daily returns for each symbol, assuming that they were simply held throughout the analysis period:

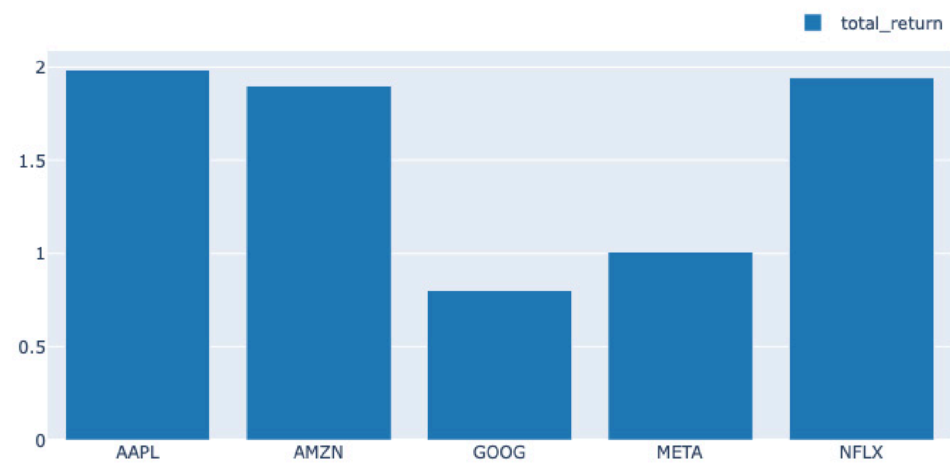

Figure 6.4: Returns from a simple holding strategy

How it works…

The code performs a simple moving average crossover strategy on the FAANG stocks to demonstrate the VectorBT backtesting framework. We download historical closing prices for the META, AAPL, AMZN, NFLX, and GOOG stocks using **`YFData.download`**. We then use the **`get`** method and the **`close`** key to return the closing prices from the **`YFData`** object.

From there, we calculate two moving averages: a fast moving average with a 10-day window and a slow moving average with a 30-day window. We do this using the **`MA`** class, which takes the price data, the window size, and a short name for the moving average as arguments. The **`ma_crossed_above`** method identifies points where the fast moving average crosses above the slow moving average, signaling a buy entry. Conversely, the **`ma_crossed_below`** method identifies points where the fast moving average crosses below the slow moving average, signaling a sell or exit.

Finally, the **`from_signals`** method is used to simulate the portfolio's performance based on these entry and exit signals and the historical price data. The resulting portfolio object, **`pf`**, contains various statistics and data that can be used for further analysis of the strategy's performance. We extract the total returns using the **`total_returns`** method, group by each symbol, and plot the mean returns.

There's more…

The power in VectorBT is not backtesting simple trading strategies. It's in VectorBT's capacity for running split tests and different iterations of parameters very quickly:

1. Let's split the data into four panels:

```
mult_prices, _ = prices.vbt.range_split(n=4)
```

The result is the following MultiIndex DataFrame, which has historic price data split across four separate split indexes:

split_idx	0					1					2					3				
symbol	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG
0	102.220001	24.009066	31.849501	109.959999	37.091999	116.860001	27.059305	37.683498	127.489998	39.306999	181.419998	40.776524	59.450500	201.070007	53.250000	131.740005	34.163826	75.014000	271.200012	50.803001
1	102.730003	23.407412	31.689501	107.660004	37.129002	118.690002	27.029024	37.859001	129.410004	39.345001	184.669998	40.769421	60.209999	205.050003	54.124001	137.949997	35.622265	78.769501	297.570007	53.535500
2	102.970001	22.949341	31.632500	117.680000	37.181000	120.669998	27.166475	39.022499	131.809998	39.701000	184.330002	40.958790	60.479500	205.630005	54.320000	138.050003	35.542969	81.475502	315.339996	53.419498
3	97.919998	21.980772	30.396999	114.559998	36.319500	123.410004	27.469334	39.799500	131.070007	40.307499	186.850006	41.425125	61.457001	209.990005	55.111500	142.529999	36.220528	82.829002	320.269989	53.813999
4	97.330002	22.097000	30.352501	111.389999	35.723499	124.900002	27.720940	39.846001	130.949997	40.332500	188.279999	41.271263	62.343498	212.050003	55.347000	144.229996	36.835613	82.971001	319.959991	53.733002
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
246	117.400002	27.091925	38.317001	125.580002	39.563000	177.199997	41.427494	58.417999	189.940002	53.006001	124.059998	35.278683	67.197998	233.880005	48.811001	205.119995	69.327438	89.460503	333.200012	67.178001
247	117.269997	27.145506	38.029499	125.589996	39.495499	175.990005	40.376480	58.838001	187.759995	52.837002	134.179993	37.763054	73.544998	253.669998	51.973000	207.789993	70.702927	93.438499	332.630005	68.019997
248	118.010002	27.317902	38.570000	128.350006	39.577499	177.619995	40.383587	59.112999	186.240005	52.468498	134.520004	37.517971	73.082001	255.570007	52.194000	208.100006	70.676102	93.489998	329.089996	67.594498
249	116.919998	27.201422	38.606499	125.889999	39.252499	177.919998	40.497204	59.305000	192.710007	52.407001	133.199997	37.537189	73.901001	256.079987	51.854000	204.410004	71.095573	92.344498	323.309998	66.806999
250	116.349998	27.194427	38.257500	125.330002	39.139500	176.460007	40.059280	58.473499	191.960007	52.320000	131.089996	37.900017	75.098503	267.660004	51.780499	205.250000	71.615021	92.391998	323.570007	66.850998

Figure 6.5: A pandas DataFrame with split indexes

2. Within each split, we can run different combinations of our fast and slow moving average windows:

```
fast_ma = vbt.MA.run(mult_prices, [10, 20],
    short_name="fast")
slow_ma = vbt.MA.run(mult_prices, [30, 30],
    short_name="slow")
```

3. Rerun the methods to find entries and exits:

```
entries = fast_ma.ma_crossed_above(slow_ma)
exits = fast_ma.ma_crossed_below(slow_ma)
```

4. Inspect the exits DataFrame. The result is a MultiIndex DataFrame that includes each combination of moving average windows within each split:

fast_window	10																			
slow_window	30																			
split_idx	0					1					2					3				
symbol	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG
0	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
246	False	False	False	False	False	False	False	False	False	False	True	False	False	False	False	False	False	False	False	False
247	False	False	False	False	False	True	False	False	False	False	False	False	False	False	False	False	False	False	False	False
248	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
249	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
250	True	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False

fast_window	20																			
slow_window	30																			
split_idx	0					1					2					3				
symbol	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG	META	AAPL	AMZN	NFLX	GOOG
0	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
246	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
247	True	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
248	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
249	False	False	True	False	False	False	False	False	False	False	False	False	False	False	True	False	False	False	False	False
250	False	False	False	False	False	False	True	False	False	False	False	False	True	False	False	False	False	False	False	False

Figure 6.6: A MultiIndex DataFrame with each combination of moving average windows within each split

5. Rerun the backtest analysis with the split data:

```
pf = vbt.Portfolio.from_signals(
    mult_prices,
    entries,
    exits,
    freq="1D"
)
```

6. Visualize the results by grouping total returns by split index and symbol, finding the mean, and plotting:

```
(
    pf
    .total_return()
    .groupby(
        ['split_idx', 'symbol']
    )
    .mean()
    .unstack(level=-1)
    .vbt
    .barplot()
)
```

The result is the following bar chart with the mean daily returns for each symbol across each split:

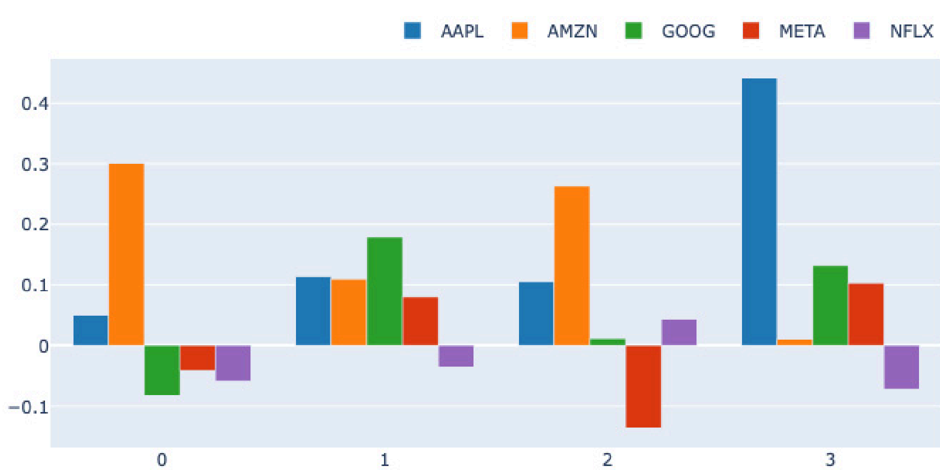

Figure 6.7: A visualization of mean daily returns grouped by split and symbol

7. VectorBT supports trading statistics based on the **orders** property on the **pf** object:

```
pf.orders.stats(group_by=True)
```

The result is a Series with various trading statistics:

```
Start                                0
End                                250
Period              251 days 00:00:00
Total Records                      291
Total Buy Orders                   157
Total Sell Orders                  134
Min Size                      0.253075
Max Size                      4.677407
Avg Size                      1.629121
Avg Buy Size                  1.637868
Avg Sell Size                 1.618872
Avg Buy Price               103.523933
Avg Sell Price              104.650348
Total Fees                         0.0
Min Fees                           0.0
Max Fees                           0.0
Avg Fees                           0.0
Avg Buy Fees                       0.0
Avg Sell Fees                      0.0
Name: group, dtype: object
```

Figure 6.8: Trading statistics from the backtest analysis

8. We can also extract specific performance metrics for each combination of split, moving average window, and symbol:

```
pf.sharpe_ratio()
```

The result is the following MultiIndex Series:

```
fast_window  slow_window  split_idx  symbol
10           30           0          META     -0.472595
                                     AAPL      1.125228
                                     AMZN      1.633329
                                     NFLX     -0.470585
                                     GOOG     -0.051110
                          1          META      0.449941
                                     AAPL      1.043138
                                     AMZN      0.611364
                                     NFLX     -0.359590
                                     GOOG      1.754603
```

Figure 6.9: The Sharpe ratio for each combination of split, moving average window, and symbol

See also

The free, open source version of VectorBT has good documentation, examples, and additional resources to help you get started. You can find it here: https://vectorbt.dev.

Conducting walk-forward optimization with VectorBT

Walk-forward optimization is an advanced technique in algorithmic trading that aims to address the issue of curve-fitting in strategy development. Unlike traditional backtesting, whereby a strategy is optimized once over a historical dataset and then applied to out-of-sample data, walk-forward optimization divides the entire dataset into multiple in-sample and out-of-sample periods. The strategy is optimized on each in-sample period. The optimized parameters are then validated on the corresponding out-of-sample period. This process is repeated, or "walked forward," through the entire dataset. The objective is to assess how well the strategy adapts to changing market conditions over time. By continually re-optimizing and validating the strategy, walk-forward optimization provides a more realistic representation of a strategy's robustness and potential for future performance. This method is computationally intensive but offers a more rigorous approach to strategy validation.

This recipe uses VectorBT's built-in **`rolling_split`** method to take advantage of the library's speed and let us run thousands of simulations in seconds.

Getting ready

This recipe uses the free, open source version of VectorBT that we installed in the last recipe.

How to do it…

We'll run the backtest analysis and test whether the out-of-sample Sharpe ratios are statistically greater than the in-sample results:

1. Import the libraries for the analysis. We'll use SciPy to run a statistical test to collect evidence of overfitting:

```
import numpy as np
import scipy.stats as stats
import vectorbt as vbt
```

2. Download market price data using the built-in downloader:

```
start = "2016-01-01 UTC"
end = "2020-01-01 UTC"
prices = vbt.YFData.download(
    "AAPL",
    start=start,
    end=end
).get("Close")
```

3. Create data splits for the walk-forward optimization. This code segments the prices into 30 splits, each two years long, and reserves 180 days for the test:

```
(in_price, in_indexes), (out_price, out_indexes) = prices.vbt.rolling_split(
    n=30,
    window_len=365 * 2,
    set_lens=(180,),
    left_to_right=False,
)
```

4. Now create the function that returns the Sharpe ratios for all combinations of moving average windows:

```
def simulate_all_params(price, windows, **kwargs):
    fast_ma, slow_ma = vbt.MA.run_combs(
        price,
        windows,
        r=2,
        short_names=["fast", "slow"]
    )
    entries = fast_ma.ma_crossed_above(slow_ma)
    exits = fast_ma.ma_crossed_below(slow_ma)
    pf = vbt.Portfolio.from_signals(price, entries,
        exits, **kwargs)
    return pf.sharpe_ratio()
```

5. Create two helper functions that return the indexes and parameters where the performance is maximized:

```
def get_best_index(performance):
    return performance[
        performance.groupby("split_idx").idxmax()
    ].index
def get_best_params(best_index, level_name):
    return best_index.get_level_values(
        level_name).to_numpy()
```

6. Implement a function that runs the backtest given the best moving average values and returns the associated Sharpe ratio:

```
def simulate_best_params(price, best_fast_windows,
    best_slow_windows, **kwargs):
        fast_ma = vbt.MA.run(
            price,
            window=best_fast_windows,
            per_column=True
        )
        slow_ma = vbt.MA.run(
            price,
            window=best_slow_windows,
            per_column=True
        )
        entries = fast_ma.ma_crossed_above(slow_ma)
        exits = fast_ma.ma_crossed_below(slow_ma)
        pf = vbt.Portfolio.from_signals(
            price, entries, exits, **kwargs)
        return pf.sharpe_ratio()
```

7. Finally, we will run the analysis by passing in a range of moving average windows to **`simulate_all_params`**. This returns the Sharpe ratio for every combination of moving average windows for every data split. In other words, these are the in-sample Sharpe ratios:

```
windows = np.arange(10, 40)
in_sharpe = simulate_all_params(
    in_price,
    windows,
    direction="both",
    freq="d"
)
```

8. Next, we will get the best in-sample moving average windows and combine them into a single array:

```
in_best_index = get_best_index(in_sharpe)
in_best_fast_windows = get_best_params(
    in_best_index,
    "fast_window"
)
in_best_slow_windows = get_best_params(
    in_best_index,
    "slow_window"
)
in_best_window_pairs = np.array(
```

```
    list(
        zip(
            in_best_fast_windows,
            in_best_slow_windows
        )
    )
)
```

9. The last step is to retrieve the out-of-sample Sharpe ratios using the optimized moving average windows:

```
out_test_sharpe = simulate_best_params(
    out_price,
    in_best_fast_windows,
    in_best_slow_windows,
    direction="both",
    freq="d"
)
```

The result is the following MultiIndex Series that identifies the optimal moving average window values for each split along with the associated Sharpe ratio:

ma_window	ma_window	split_idx	
10	11	0	0.104954
12	13	1	0.318318
		2	0.971219
10	11	3	1.386785
12	13	4	1.303272
10	11	5	2.133298
		6	2.043526

Figure 6.10: Maximized Sharpe ratios across each data split

How it works…

The code executes a walk-forward optimization on a moving average crossover strategy for AAPL stock. We begin by fetching historical closing prices for AAPL for the period from January 1, 2016, to January 1, 2020. The data is then partitioned into in-sample and out-of-sample sets using VectorBT's **`rolling_split`** method. The in-sample set is designated for optimization, while the out-of-sample set is reserved for validation.

The **`rolling_split`** method in VectorBT is designed to split a time series into rolling in-sample and out-of-sample periods for walk-forward optimization or other time-based analyses. These are the parameters we use in the recipe:

- **`n`**: This refers to the number of splits. It determines how many in-sample and out-of-sample periods will be created.
- **`window_len`**: This describes the length of each rolling window in the time series. This is often specified in terms of the number of time steps (e.g., days).
- **`set_lens`**: This is a tuple specifying the length of each in-sample and out-of-sample set within each rolling window. The sum of these lengths should not exceed **`window_len`**.
- **`left_to_right`**: Determines whether to resolve **`set_lens`** from left to right. Otherwise, the first set will be variable.

In the optimization phase, the **`simulate_all_params`** function is responsible for strategy backtesting across a range of moving average window sizes. It calculates both fast and slow moving averages, generates entry and exit signals, and simulates the portfolio's performance, returning the Sharpe ratio as the performance metric.

> *TIP*

> *You can select any performance metric to optimize for. The* **`pf`** *portfolio object has dozens of different metrics to choose from. You can execute* **`dir(pf)`** *to inspect the properties and methods of the object.*

Next, we use the **`get_best_index`** and **`get_best_params`** functions to identify the optimal moving average window sizes based on the highest Sharpe ratios achieved during the in-sample testing.

Finally, the code proceeds to the out-of-sample testing phase. The **`simulate_best_params`** function takes the optimal moving average window sizes identified in the in-sample phase and applies them to the out-of-sample dataset. It then simulates the portfolio's performance using these parameters, again computing the Sharpe ratio as the performance metric.

The code is structured to leverage VectorBT's efficient backtesting capabilities, which is particularly advantageous in a walk-forward optimization context where multiple iterations of backtesting and re-optimization are required.

There's more…

It's common to overfit backtesting models to market noise. This is especially acute when brute force optimizing technical analysis strategies. To collect evidence to this effect, we can use a one-sided independent t-test to assess the statistical significance between the means of Sharpe ratios for in-sample and out-of-sample datasets:

```
in_sample_best = in_sharpe[in_best_index].values
out_sample_test = out_test_sharpe.values
t, p = stats.ttest_ind(
    a=out_sample_test,
    b=in_sample_best,
    alternative="greater"
)
```

First, the line **`in_sample_best = in_sharpe[in_best_index].values`** filters the Sharpe ratios stored in **`in_sharpe`** to include only those corresponding to the best-performing parameter sets. It then extracts these filtered Sharpe ratios as a NumPy array and stores them in the **`in_sample_best`** variable. We do the same for the out-of-sample dataset.

The **`ttest_ind`** function from the SciPy stats module takes the two independent **`out_sample_test`** and **`in_sample_best`** samples as its arguments. The **`alternative="greater"`** parameter specifies that the test is one-sided, which we use to evaluate whether the mean Sharpe ratio of the out-of-sample set is statistically greater than that of the in-sample set. The function returns the calculated t-statistic and the p-value.

The results give us a t-statistic of approximately **`-1.085`** and a p-value of approximately **`0.859`**. The negative value of the t-statistic suggests that the mean of the out-of-sample Sharpe ratios is negative. Further, the high p-value tells us there is not enough statistical evidence to conclude that the out-of-sample Sharpe ratios are greater than the in-sample Sharpe ratios. The negative t-statistic and the high p-value together suggest that the strategy may not perform as well on new, unseen data as it does on the data on which it was optimized. This could be a warning sign regarding the strategy's robustness and its ability to generalize to new data. Ideally, you'd hope to see a t-statistic over **`1.0`** and a p-value under **`0.05`**.

See also

Using VectorBT to quickly iterate on optimized strategies, coupled with SciPy to test the statistical significance of those strategies, is a powerful workflow. You can learn more about SciPy's stats module here: **https://docs.scipy.org/doc/scipy/reference/stats.html#module-scipy.stats**.

Optimizing the SuperTrend strategy with VectorBT Pro

The SuperTrend indicator is a trend-following indicator that is used in technical analysis to identify the direction of an asset's momentum. It is constructed using two primary components: the **Average True Range** (**ATR**) and a multiplier. The ATR measures the asset's volatility over a specified period, while the multiplier is a user-defined constant that adjusts the sensitivity of the indicator.

The SuperTrend is calculated as follows:

1. Compute the ATR for a given period.
2. Calculate the **basic upper band** as the sum of the high and low prices, divided by 2, plus the product of the multiplier and the ATR.
3. Calculate the **basic lower band** as the sum of the high and low prices, divided by 2, minus the product of the multiplier and the ATR.
4. The SuperTrend is then defined as the upper band when the price is below it, and as the lower band when the price is above it.

The indicator flips between the upper and lower bands, signaling a change in trend. When the price is above the SuperTrend line, it suggests an uptrend and a buy signal is generated. Conversely, when the price is below the SuperTrend line, it suggests a downtrend and a sell signal is generated.

To build the SuperTrend indicator, we will introduce VectorBT Pro. VectorBT Pro is a more full-featured version of VectorBT that offers enhancements such as pulling data from AlphaVantage and Polygon, as well as synthetic data generators, multi-threading, stop laddering, time stops, order delays, portfolio optimization with RiskFolio-Lib and PyPortfolioOpt, and more.

This recipe will demonstrate how to construct a custom indicator using integrations with TA-Lib and Numba.

Getting ready

We need to install a few dependencies to get VectorBT working. First is TA-Lib which is a technical analysis library. The second is H5DF which is a data storage solution.

For Windows, Unix/Linux, and Mac Intel users

If you're running on an Intel x86 chip, you can use **conda**:

```
conda install -c conda-forge pytables h5py ta-lib -y
```

For Apple Silicon users

If you have a Mac with an M-series chip, you need to install some dependencies first. The easiest way is to use Homebrew (https://brew.sh).

Install the dependencies with Homebrew:

```
brew install hdf5 ta-lib
```

Install the Python dependencies with **conda**:

```
conda install -c conda-forge pytables h5py ta-lib -y
```

Now we're ready to install VectorBT Pro.

VectorBT Pro has a small monthly subscription fee. Details can be found at https://vectorbt.pro/become-a-member/. After you've been added to the list of collaborators and have accepted the repository invitation, the next step is to create a Personal Access Token for your GitHub account to access the Pro repository. To do so, follow these steps:

1. Go to https://github.com/settings/tokens.
2. Click on **Generate a new token**.
3. Enter a name (such as `terminal`).

4. Set the expiration to a fixed number of days.
5. Select the **repo** scope.
6. Generate the token and save it in a safe place.

Once your token has been created, you can install Pro using **`pip`**:

```
pip install -U "vectorbtpro[base] @ git+https://github.com/polakowo/vectorbt.pro.
```

When you're prompted for a password, use the token that you generated in the previous steps. For more installation details, see the *Getting Started* guide on the Pro website, which is accessible after signing up.

Next, we'll need TA-Lib. TA-Lib is a technical analysis library. Because it's written in C++ with Python wrappers, we need a little special handling to get it installed.

How to do it…

We'll examine a few key features of VectorBT Pro, including the multi-threaded data downloading and indicator factory:

1. Import the libraries we need for the analysis. Note the import of **`vectorbtpro`** instead of **`vectorbt`**:

```
import talib
import numpy as np
from numba import njit
import vectorbtpro as vbt
```

2. Use VectorBT Pro's multi-threading capability to download the market data in 517 milliseconds:

```
start = "2016-01-01 UTC"
end = "2020-01-01 UTC"
with vbt.Timer() as timer:
    prices = vbt.YFData.pull(
        ["META", "AAPL", "AMZN", "NFLX", "GOOG"],
        start=start,
        end=end,
        execute_kwargs=dict(engine="threadpool")
    )
print(timer.elapsed())
```

3. Extract the high, low, and closing prices from the **`prices`** object:

```
high = prices.get("High")
low = prices.get("Low")
close = prices.get("Close")
```

4. Implement a helper function that calculates the basic bands:

```
def get_basic_bands(med_price, atr, multiplier):
    matr = multiplier * atr
    upper = med_price + matr
    lower = med_price - matr
    return upper, lower
```

5. Implement the function that calculates the final bands. This function returns the trend and direction, as well as both the long and short positions. Note the **`@njit`** decorator, which compiles the code using Numba to achieve machine language-like speeds:

```
@njit
def get_final_bands(close, upper, lower):
    trend = np.full(close.shape, np.nan)
    dir_ = np.full(close.shape, 1)
    long = np.full(close.shape, np.nan)
    short = np.full(close.shape, np.nan)
    for i in range(1, close.shape[0]):
        if close[i] > upper[i - 1]:
            dir_[i] = 1
        elif close[i] < lower[i - 1]:
            dir_[i] = -1
        else:
            dir_[i] = dir_[i - 1]
            if dir_[i] > 0 and lower[i] < lower[i - 1]:
                lower[i] = lower[i - 1]
            if dir_[i] < 0 and upper[i] > upper[i - 1]:
                upper[i] = upper[i - 1]
        if dir_[i] > 0:
            trend[i] = long[i] = lower[i]
        else:
            trend[i] = short[i] = upper[i]
    return trend, dir_, long, short
```

6. Put it all together in the final function, which returns the output from the **`get_final_bands`** function. Note the use of TA-Lib's **`MEDPRICE`** and **`ATR`** methods, which further speed up the analysis:

```
def supertrend(high, low, close, period=7,
    multiplier=3):
        avg_price = talib.MEDPRICE(high, low)
```

```
        atr = talib.ATR(high, low, close, period)
        upper, lower = get_basic_bands(avg_price, atr,
            multiplier)
        return get_final_bands(close, upper, lower)
```

7. Use VectorBT Pro's indicator factory class to convert our **`supertrend`** function into an indicator that we can use with Pro's analysis capabilities:

```
SuperTrend = vbt.IF(
    class_name="SuperTrend",
    short_name="st",
    input_names=["high", "low", "close"],
    param_names=["period", "multiplier"],
    output_names=["supert", "superd", "superl",
        "supers"],
).with_apply_func(
    supertrend,
    takes_1d=True,
    period=7,
    multiplier=3,
)
```

8. We will then create a custom class that inherits the indicator that we just built using VectorBT Pro's indicator factory. This class includes a method plot that encapsulates the formatting and setup of a plot:

```
class SuperTrend(SuperTrend):
    def plot(
        self,
        column=None,
        close_kwargs=None,
        superl_kwargs=None,
        supers_kwargs=None,
        fig=None,
        **layout_kwargs
    ):
        close_kwargs = close_kwargs if close_kwargs else {}
        superl_kwargs = superl_kwargs if superl_kwargs else {}
        supers_kwargs = supers_kwargs if supers_kwargs else {}
        close = self.select_col_from_obj(self.close,
            column).rename("Close")
        supers = self.select_col_from_obj(self.supers,
            column).rename("Short")
        superl = self.select_col_from_obj(self.superl,
            column).rename("Long")
        fig = close.vbt.plot(fig=fig, **close_kwargs,
```

```
            **layout_kwargs)
        supers.vbt.plot(fig=fig, **supers_kwargs)
        superl.vbt.plot(fig=fig, **superl_kwargs)
        return fig
```

9. We're now ready to create an instance of our SuperTrend indicator and visualize where the indicator suggests long and short positions:

```
st = SuperTrend.run(high, low, close)
st.loc["2016-01-01":"2020-01-01",
    "AAPL"].plot().show()
```

The result is a Plotly chart with the closing price for AAPL and the locations of long and short signals:

Figure 6.11: A chart with the long and short signals for AAPL

10. The next step is to run the backtest. First, let's find the entry and exit signals:

```
entries = (~st.superl.isnull()).vbt.signals.fshift()
exits = (~st.supers.isnull()).vbt.signals.fshift()
```

The result is two DataFrames with boolean values for each day indicating where a long or short position exists:

symbol	META	AAPL	AMZN	NFLX	GOOG
Date					
2016-01-04 00:00:00-05:00	False	False	False	False	False
2016-01-05 00:00:00-05:00	False	False	False	False	False
2016-01-06 00:00:00-05:00	False	False	False	False	False
2016-01-07 00:00:00-05:00	False	False	False	False	False
2016-01-08 00:00:00-05:00	False	False	False	False	False

Figure 6.12: A DataFrame with entry locations

11. Finally, we can run the backtest and inspect the risk and performance results:

```
pf = vbt.Portfolio.from_signals(
    close=close,
    entries=entries,
    exits=exits,
    fees=0.001,
    freq="1d"
)
pf.stats(group_by=True)
```

    The result is a Series with the consolidated statistics of the strategy:

```
Start                              2016-01-04 00:00:00-05:00
End                                2019-12-31 00:00:00-05:00
Period                                    1006 days 00:00:00
Start Value                                            500.0
Min Value                                         458.811175
Max Value                                         744.719854
End Value                                         739.035201
Total Return [%]                                    47.80704
Benchmark Return [%]                               152.73135
Total Time Exposure [%]                            86.481113
Max Gross Exposure [%]                                 100.0
Max Drawdown [%]                                   11.348404
Max Drawdown Duration                      184 days 00:00:00
Total Orders                                             161
Total Fees Paid                                    18.243214
Total Trades                                              83
Win Rate [%]                                       56.410256
Best Trade [%]                                     27.086949
Worst Trade [%]                                   -22.352306
Avg Winning Trade [%]                               7.681076
Avg Losing Trade [%]                                 -5.1919
Avg Winning Trade Duration    50 days 13:38:10.909090909
Avg Losing Trade Duration     16 days 22:35:17.647058823
Profit Factor                                       1.843733
Expectancy                                          2.132568
Sharpe Ratio                                        0.967803
Calmar Ratio                                        0.905523
Omega Ratio                                         1.177398
Sortino Ratio                                         1.3654
Name: group, dtype: object
```

Figure 6.13: A sample of the Series with the risk and performance statistics for the SuperTrend strategy

How it works…

The code fetches historical financial data on META, AAPL, AMZN, NFLX, and GOOG for the date range from January 1, 2016, to January 1, 2020. It uses the **pull** method from the **YFData** class to download this data. The **execute_kwargs=dict(engine="threadpool")** argument specifies that a thread pool should be used for concurrent execution, which speeds up the data retrieval process. The last data pre-processing step is to simply extract the high, low, and closing prices from the **YFData** object.

The **get_basic_bands** function calculates the basic upper and lower bands used in the SuperTrend indicator. It takes three arguments: **med_price**, which is the median price of the asset, **atr**, the ATR, and **multiplier**, a user-defined constant to adjust sensitivity. The function returns both the upper and lower bands.

Next, we implement the heart of the SuperTrend indicator. The **get_final_bands** function is JIT compiled, which is optimized using Numba's **@njit** decorator for better performance. It calculates the final upper and lower bands, as well as the trend direction for our SuperTrend indicator. It takes three arrays as input: **close** for closing prices, **upper** for the basic upper band, and **lower** for the basic lower band. The function returns four arrays: **trend** (final band based on trend direction), **dir_** (trend direction), **long** (lower band during uptrends), and **short** (upper band during downtrends).

supertrend calculates the SuperTrend indicator using high, low, and close prices. It takes five arguments: **high**; **low**; **close** for the high, low, and closing prices, respectively; **period** for the number of periods to consider for the ATR; and **multiplier** to adjust the sensitivity of the bands. The **supertrend** function returns the final upper and lower bands, the trend direction, and the bands used during uptrends and downtrends, as calculated by **get_final_bands**.

Next, we define a custom indicator class, **SuperTrend**, using VectorBT Pro's **IndicatorFactory** class. This class encapsulates the SuperTrend indicator logic for easier reuse and integration within the Pro ecosystem. The **with_apply_func** method attaches the previously defined **supertrend** function to this custom indicator class. This function will be called when the **SuperTrend** indicator is run.

Finally, we run the backtest. We first run the custom **SuperTrend** indicator on high, low, and close price data, generating SuperTrend values, trend directions, and bands for both long and short positions. Finally, a portfolio is constructed using the entry and exit signals with VectorBT Pro's **`from_signals`** method. The portfolio uses the close prices for trade execution, incorporates a trading fee of 0.1%, and assumes a daily trading frequency.

There's more…

Since we encapsulated the SuperTrend indicator logic using Pro's indicator factory, we can optimize it. The two parameters in question are **`period`** and **`multiplier`**:

1. First, let's create ranges of values for each of the parameters:

```
periods = np.arange(4, 20)
multipliers = np.arange(20, 41) / 10
```

2. Then we call the run method on the SuperTrend indicator, passing in the market prices, as well as our arrays of periods and multipliers:

```
st = SuperTrend.run(
    high, low, close,
    period=periods,
    multiplier=multipliers,
    param_product=True,
)
```

3. Pro runs through every combination of period and multiplier. We then run through the same code as before that identifies the entries, exits, and runs the backtest. The difference is that now, our DataFrames containing entry and exit positions have a MultiIndex index for each combination of period and multiplier:

st_period	4					...	19				
st_multiplier	2.0					...	4.0				
symbol	META	AAPL	AMZN	NFLX	GOOG	...	META	AAPL	AMZN	NFLX	GOOG
Date											
2016-01-04 00:00:00-05:00	False	False	False	False	False	...	False	False	False	False	False
2016-01-05 00:00:00-05:00	False	False	False	False	False	...	False	False	False	False	False
2016-01-06 00:00:00-05:00	False	False	False	False	False	...	False	False	False	False	False
2016-01-07 00:00:00-05:00	False	False	False	False	False	...	False	False	False	False	False
2016-01-08 00:00:00-05:00	False	False	False	False	False	...	False	False	False	False	False

Figure 6.14: A DataFrame with entry locations for each stock for each combination of period and multiplier

4. Once we have created the entries and exits, we run the backtest:

```
pf = vbt.Portfolio.from_signals(
    close=close,
    entries=entries,
    exits=exits,
    fees=0.001,
    freq="1d"
)
```

5. To visualize the parameter hotspots (the combinations with the maximum Sharpe ratio), use a heatmap:

```
pf.sharpe_ratio.vbt.heatmap(
    x_level="st_period",
    y_level="st_multiplier",
    slider_level="symbol"
)
```

The result is an interactive heatmap that lets us select which asset to visualize. In this case, we select AAPL:

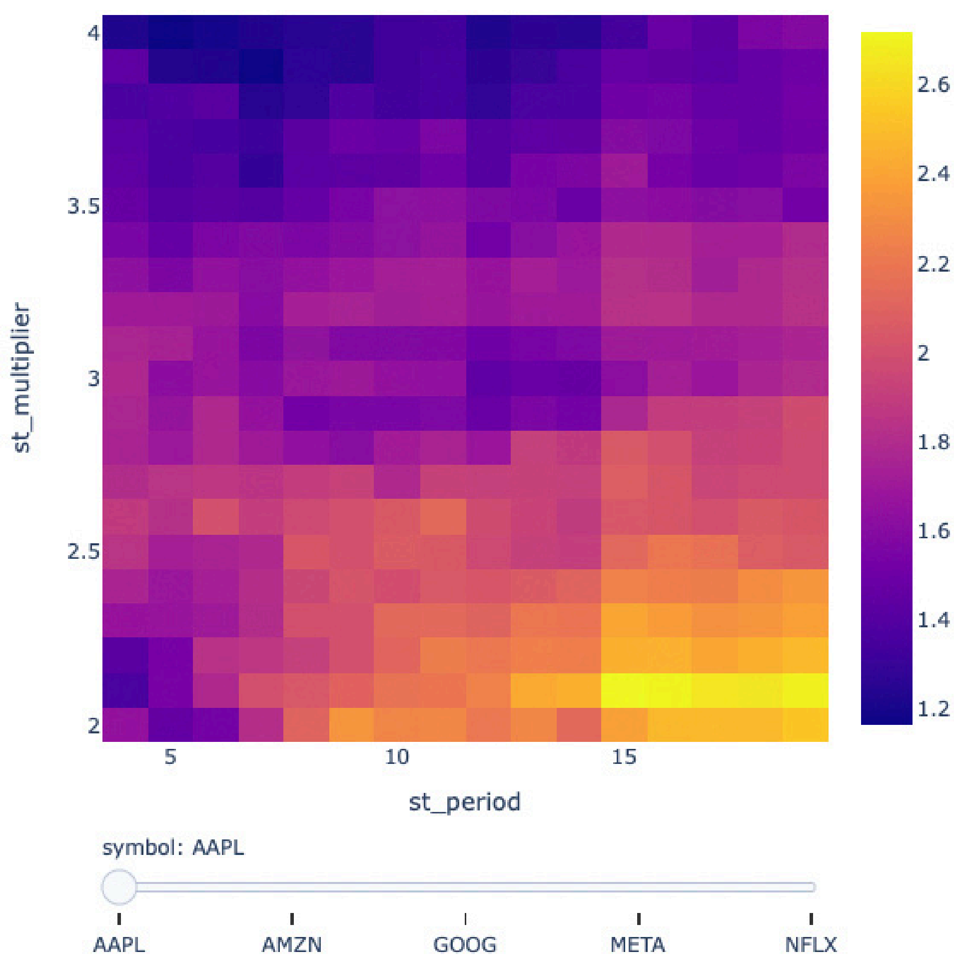

Figure 6.15: An interactive heatmap showing the Sharpe ratio at each parameter combination

> *TIP*

> *The* `pf` *object has many risk and performance metrics available for plotting. To get an idea of what's available, print* `dir(pf)` *to see the object attributes. You can easily plot a different metric on a heatmap by replacing* `sharpe_ratio` *with whichever metric you're interested in.*

See also

We've only begun to scratch the surface of VectorBT and VectorBT Pro. It's designed for speed, which is perfect for optimizing features of our trading strategies.

To learn more about the SuperTrend indicator, consult the VectorBT Pro information page at https://vectorbt.pro/become-a-member/.