8
The ML4T Workflow – From Model to Strategy Backtesting

Now, it's time to **integrate the various building blocks** of the **machine learning for trading (ML4T)** workflow that we have so far discussed separately. The goal of this chapter is to present an end-to-end perspective of the process of designing, simulating, and evaluating a trading strategy driven by an ML algorithm. To this end, we will demonstrate in more detail how to backtest an ML-driven strategy in a historical market context using the Python libraries backtrader and Zipline.

The **ultimate objective of the ML4T workflow** is to gather evidence from historical data. This helps us decide whether to deploy a candidate strategy in a live market and put financial resources at risk. This process builds on the skills you developed in the previous chapters because it relies on your ability to:

- Work with a diverse set of data sources to engineer informative factors
- Design ML models that generate predictive signals to inform your trading strategy
- Optimize the resulting portfolio from a risk-return perspective

A realistic simulation of your strategy also needs to faithfully represent how security markets operate and how trades are executed. Therefore, the institutional details of exchanges, such as which order types are available and how prices are determined, also matter when you design a backtest or evaluate whether a backtesting engine includes the requisite features for accurate performance measurements. Finally, there are several methodological aspects that require attention to avoid biased results and false discoveries that will lead to poor investment decisions.

More specifically, after working through this chapter, you will be able to:

- Plan and implement end-to-end strategy backtesting
- Understand and avoid critical pitfalls when implementing backtests
- Discuss the advantages and disadvantages of vectorized versus event-driven backtesting engines
- Identify and evaluate the key components of an event-driven backtester
- Design and execute the ML4T workflow using data sources at both minute and daily frequencies, with ML models trained separately or as part of the backtest
- Use Zipline and backtrader

You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images.

How to backtest an ML-driven strategy

In a nutshell, the ML4T workflow, illustrated in *Figure 8.1*, is about backtesting a trading strategy that leverages machine learning to generate trading signals, select and size positions, or optimize the execution of trades. It involves the following steps, with a specific investment universe and horizon in mind:

1. Source and prepare market, fundamental, and alternative data
2. Engineer predictive alpha factors and features
3. Design, tune, and evaluate ML models to generate trading signals
4. Decide on trades based on these signals, for example, by applying rules
5. Size individual positions in the portfolio context
6. Simulate the resulting trades triggered using historical market data
7. Evaluate how the resulting positions would have performed

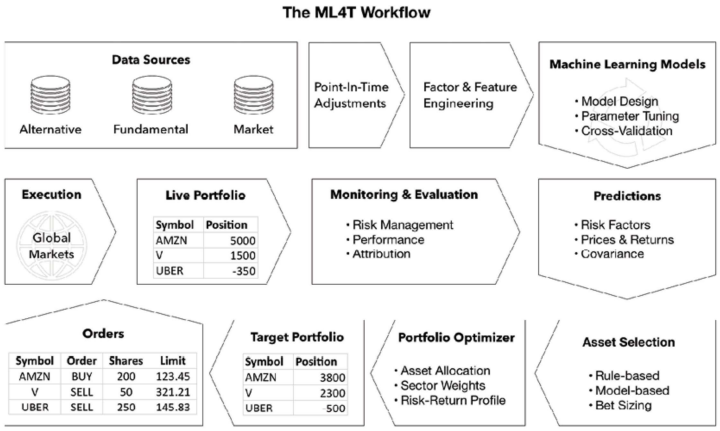

Figure 8.1: The ML4T workflow

When we discussed the ML process in *Chapter 6*, *The Machine Learning Process*, we emphasized that the model's learning should generalize well to new applications. In other words, the predictions of an ML model trained on a given set of data should perform equally well when provided new input data. Similarly, the (relative) **backtest performance of a strategy should be indicative of future market performance**.

Before we take a look at how backtesting engines run historical simulations, we need to review several methodological challenges. Failing to properly address them will render results unreliable and lead to poor decisions about the strategy's live implementation.

Backtesting pitfalls and how to avoid them

Backtesting simulates an algorithmic strategy based on historical data, with the goal of producing performance results that generalize to new market conditions. In addition to the generic uncertainty around predictions in the context of ever-changing markets, several implementation aspects can bias the results and increase the risk of mistaking in-sample performance for patterns that will hold out-of-sample.

These aspects are under our control and include the selection and preparation of data, unrealistic assumptions about the trading environment, and the flawed application and interpretation of statistical tests. The risks of false backtest discoveries multiply with increasing computing power, bigger datasets, and more complex algorithms that facilitate the misidentification of apparent signals in a noisy sample.

In this section, we will outline the most serious and common methodological mistakes. Please refer to the literature on multiple testing for further detail, in particular, a series of articles by Marcos Lopez de Prado collected in *Advances in Financial Machine Learning (2018)*. We will also introduce the deflated **Sharpe ratio (SR)**, which illustrates how to adjust metrics that result from repeated trials when using the same set of financial data for your analysis.

Getting the data right

Data issues that undermine the validity of a backtest include **look-ahead bias**, **survivorship bias**, **outlier control**, as well as the **selection of the sample period**. We will address each of these in turn.

Look-ahead bias – use only point-in-time data

At the heart of an algorithmic strategy are trading rules that trigger actions based on data. Look-ahead bias emerges when we develop or evaluate trading rules **using historical information before it was known or available**. The resulting performance measures will be misleading and not representative of the future when data availability differs during live strategy execution.

A common cause of this bias is the failure to account for corrections or restatements of reported financials after their initial publication. Stock splits or reverse splits can also generate look-ahead bias. For example, when computing the earnings yield, **earnings-per-share (EPS)** data is usually reported on a quarterly basis, whereas market prices are available at a much higher frequency. Therefore, adjusted EPS and price data need to be synchronized, taking into account when the available data was, in fact, released to market participants.

The **solution** involves the careful validation of the timestamps of all data that enters a backtest. We need to guarantee that conclusions are based

only on point-in-time data that does not inadvertently include information from the future. High-quality data providers ensure that these criteria are met. When point-in-time data is not available, we need to make (conservative) assumptions about the lag in reporting.

Survivorship bias – track your historical universe

Survivorship bias arises when the backtest data contains only securities that are currently active while **omitting assets that have disappeared** over time, due to, for example, bankruptcy, delisting, or acquisition. Securities that are no longer part of the investment universe often did not perform well, and failing to include these cases positively skew the backtest result.

The **solution**, naturally, is to verify that datasets include all securities available over time, as opposed to only those that are still available when running the test. In a way, this is another way of ensuring the data is truly point-in-time.

Outlier control – do not exclude realistic extremes

Data preparation typically includes some treatment of outliers such as winsorizing, or clipping, extreme values. The challenge is to **identify outliers that are truly not representative** of the period under analysis, as opposed to any extreme values that are an integral part of the market environment at that time. Many market models assume normally distributed data when extreme values are observed more frequently, as suggested by fat-tailed distributions.

The **solution** involves a careful analysis of outliers with respect to the probability of extreme values occurring and adjusting the strategy parameters to this reality.

Sample period – try to represent relevant future scenarios

A backtest will not yield representative results that generalize to the future if the sample data does not **reflect the current (and likely future) environment**. A poorly chosen sample data might lack relevant market regime aspects, for example, in terms of volatility or volumes, fail to include enough data points, or contain too many or too few extreme historical events.

The **solution** involves using sample periods that include important market phenomena or generating synthetic data that reflects the relevant market characteristics.

Getting the simulation right

Practical issues related to the implementation of the historical simulation include:

- Failure to **mark to market** to accurately reflect market prices and account for drawdowns

- **Unrealistic assumptions** about the availability, cost, or market impact of trades
- Incorrect **timing of signals and trade execution**

Let's see how to identify and address each of these issues.

Mark-to-market performance – track risks over time

A strategy needs to **meet investment objectives and constraints at all times**. If it performs well over the course of the backtest but leads to unacceptable losses or volatility over time, this will (obviously) not be practical. Portfolio managers need to track and report the value of their positions, called mark to market, on a regular basis and possibly in real time.

The solution involves plotting performance over time or calculating (rolling) risk metrics, such as the **value at risk (VaR)** or the Sortino ratio.

Transaction costs – assume a realistic trading environment

Markets do not permit the execution of all trades at all times or at the targeted price. A backtest that assumes **trades that may not actually be available** or would have occurred at less favorable terms will produce biased results.

Practical shortcomings include a strategy that assumes short sales when there may be no counterparty, or one that underestimates the market impact of trades (slippage) that are large or deal in less liquid assets, or the costs that arise due to broker fees.

The **solution** includes a limitation to a liquid universe and/or realistic parameter assumptions for trading and slippage costs. This also safeguards against misleading conclusions from unstable factor signals that decay fast and produce a high portfolio turnover.

Timing of decisions – properly sequence signals and trades

Similar to look-ahead bias, the simulation could make **unrealistic assumptions about when it receives and trades on signals**. For instance, signals may be computed from close prices when trades are only available at the next open, with possibly quite different prices. When we evaluate performance using the close price, the backtest results will not represent realistic future outcomes.

The **solution** involves careful orchestration of the sequence of signal arrival, trade execution, and performance evaluation.

Getting the statistics right

The most prominent challenge when backtesting validity, including published results, is the discovery of spurious patterns due to multiple testing. Selecting a strategy based on the tests of different candidates on the same data will bias the choice. This is because a positive outcome is more likely caused by the stochastic nature of the performance measure itself.

In other words, the strategy overfits the test sample, producing deceptively positive results that are unlikely to generalize to future data that's encountered during live trading.

Hence, backtest performance is only informative if the number of trials is reported to allow for an assessment of the risk of selection bias. This is rarely the case in practical or academic research, inviting doubts about the validity of many published claims.

Furthermore, the risk of backtest overfitting does not only arise from running numerous tests but also affects strategies designed based on prior knowledge of what works and doesn't. Since the risks include the knowledge of backtests run by others on the same data, backtest-overfitting is very hard to avoid in practice.

Proposed **solutions** include prioritizing tests that can be justified using investment or economic theory, rather than arbitrary data-mining efforts. It also implies testing in a variety of contexts and scenarios, including possibly on synthetic data.

The minimum backtest length and the deflated SR

Marcos Lopez de Prado (`http://www.quantresearch.info/`) has published extensively on the risks of backtesting and how to detect or avoid it. This includes an online simulator of backtest-overfitting (`http://datagrid.lbl.gov/backtest/`, *Bailey, et al. 2015*).

Another result includes an estimate of the minimum length of the backtest period that an investor should require to avoid selecting a strategy that achieves a certain SR for a given number of in-sample trials, but has an expected out-of-sample SR of zero. The result implies that, for example, 2 years of daily backtesting data does not support conclusions about more than seven strategies. 5 years of data expands this number to 45 strategy variations. See *Bailey, Borwein, and Prado (2016)* for implementation details.

Bailey and Prado (2014) also derived a deflated SR to compute the probability that the SR is statistically significant while controlling for the inflationary effect of multiple testing, non-normal returns, and shorter sample lengths. (See the `multiple_testing` subdirectory for the Python implementation of `deflated_sharpe_ratio.py` and references for the derivation of the related formulas.)

Optimal stopping for backtests

In addition to limiting backtests to strategies that can be justified on theoretical grounds as opposed to mere data-mining exercises, an important question is when to stop running additional tests.

One way to answer this question relies on the solution to the **secretary problem** from the optimal stopping theory. This problem assumes we are selecting an applicant based on interview results and need to decide whether to hold an additional interview or choose the most recent candi-

date. In this context, the optimal rule is to always reject the first n/e candidates and then select the first candidate that surpasses all the previous options. Using this rule results in a $1/e$ probability of selecting the best candidate, irrespective of the size n of the candidate pool.

Translating this rule directly to the backtest context produces the following **recommendation**: test a random sample of $1/e$ (roughly 37 percent) of reasonable strategies and record their performance. Then, continue with the tests until a strategy outperforms those tested before. This rule applies to tests of several alternatives, with the goal of choosing a near-best as soon as possible while minimizing the risk of a false positive. See the resources listed on GitHub for additional information.

How a backtesting engine works

Put simply, a backtesting engine iterates over historical prices (and other data), passes the current values to your algorithm, receives orders in return, and keeps track of the resulting positions and their value.

In practice, there are numerous requirements for creating a realistic and robust simulation of the ML4T workflow that was depicted in *Figure 8.1* at the beginning of this chapter. The difference between vectorized and event-driven approaches illustrates how the faithful reproduction of the actual trading environment adds significant complexity.

Vectorized versus event-driven backtesting

A vectorized backtest is the most basic way to evaluate a strategy. It simply multiplies a signal vector that represents the target position size with a vector of returns for the investment horizon to compute the period performance.

Let's illustrate the vectorized approach using the daily return predictions that we created using ridge regression in the previous chapter. Using a few simple technical factors, we predicted the returns for the next day for the 100 stocks with the highest recent dollar trading volume (see *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*, for details).

We'll transform the predictions into signals for a very simple strategy: on any given trading day, we will go long on the 10 highest positive predictions and go short on the lowest 10 negative predictions. If there are fewer positive or negative predictions, we'll hold fewer long or short positions. The notebook `vectorized_backtest` contains the following code example, and the script `data.py` creates the input data stored in `backtest.h5`.

First, we load the data for our strategy, as well as S&P 500 prices (which we convert into daily returns) to benchmark the performance:

```
sp500 = web.DataReader('SP500', 'fred', '2014', '2018').pct_change()
data = pd.read_hdf('00_data/backtest.h5', 'data')
data.info()
```

```
MultiIndex: 187758 entries, ('AAL', Timestamp('2014-12-09 00:00:00')) to ('ZTS', Timestamp('2017-
Data columns (total 6 columns):
 #   Column     Non-Null Count   Dtype
---  ------     --------------   -----
 0   predicted  74044 non-null   float64
 1   open       187758 non-null  float64
 2   high       187758 non-null  float64
 3   low        187758 non-null  float64
 4   close      187758 non-null  float64
 5   volume     187758 non-null  float64
```

The data combines daily return predictions and OHLCV market data for 253 distinct stocks over the 2014-17 period, with 100 equities for each day. Now, we can compute the daily forward returns and convert these and the predictions into wide format, with one ticker per column:

```
daily_returns = data.open.unstack('ticker').sort_index().pct_change()
fwd_returns = daily_returns.shift(-1)
predictions = data.predicted.unstack('ticker')
```

The next step is to select positive and negative predictions, rank them in descending and ascending fashion, and create long and short signals using an integer mask that identifies the top 10 on each side with identifies the predictions outside the top 10 with a one, and a zero:

```
long_signals = (predictions.where(predictions>0).rank(axis=1, ascending=False) > 10).astype(int)
short_signals = (predictions.where(predictions<0).rank(axis=1) > 10).astype(int)
```

We can then multiply the binary DataFrames with the forward returns (using their negative inverse for the shorts) to get the daily performance of each position, assuming equal-sized investments. The daily average of these returns corresponds to the performance of equal-weighted long and short portfolios, and the sum reflects the overall return of a market-neutral long-short strategy:

```
long_returns = long_signals.mul(fwd_returns).mean(axis=1)
short_returns = short_signals.mul(-fwd_returns).mean(axis=1)
strategy = long_returns.add(short_returns).to_frame('strategy')
```

When we compare the results, as shown in *Figure 8.2*, our strategy performed well compared to the S&P 500 for the first 2 years of the period – that is, until the benchmark catches up and our strategy underperforms during 2017.

The strategy returns are also less volatile with a standard deviation of 0.002 compared to 0.008 for the S&P 500; the correlation is low and negative at -0.093:

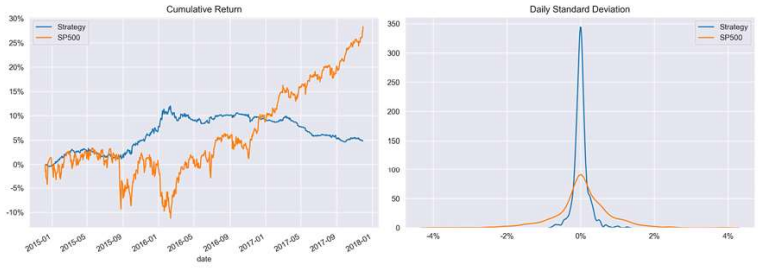

Figure 8.2: Vectorized backtest results

While this approach permits a quick back-of-the-envelope evaluation, it misses important features of a robust, realistic, and user-friendly backtest engine; for example:

- We need to manually align the timestamps of predictions and returns (using pandas' built-in capabilities) and do not have any safeguards against inadvertent look-ahead bias.
- There is no explicit position sizing and representation of the trading process that accounts for costs and other market realities, or an accounting system that tracks positions and their performance.
- There is also no performance measurement other than what we compute after the fact, and risk management rules like stop-loss are difficult to simulate.

That's where event-driven backtesting comes in. An event-driven backtesting engine explicitly simulates the time dimension of the trading environment and imposes significantly more structure on the simulation. This includes the use of historical calendars that define when trades can be made and when quotes are available. The enforcement of timestamps also helps to avoid look-ahead bias and other implementation errors mentioned in the previous section (but there is no guarantee).

Generally, event-driven systems aim to capture the actions and constraints encountered by a strategy more closely and, ideally, can readily be converted into a live trading engine that submits actual orders.

Key implementation aspects

The requirements for a realistic simulation may be met by a **single platform** that supports all steps of the process in an end-to-end fashion, or by **multiple tools** that each specialize in different aspects.

For instance, you could handle the design and testing of ML models that generate signals using generic ML libraries like scikit-learn, or others that we will encounter in this book, and feed the model outputs into a separate backtesting engine. Alternatively, you could run the entire ML4T workflow end-to-end on a single platform like Quantopian or QuantConnect.

The following sections highlight key items and implementation details that need to be addressed to put this process into action.

Data ingestion – format, frequency, and timing

The first step in the process concerns the sources of data. Traditionally, algorithmic trading strategies focused on market data, namely the OHLCV price and volume data that we discussed in *Chapter 2*, *Market and Fundamental Data – Sources and Techniques*. Today, data sources are more diverse and raise the question of how many different **storage formats and data types** to support, and whether to use a proprietary or custom format or rely on third-party or open source formats.

Another aspect is the **frequency of data sources** that can be used and whether sources at different frequencies can be combined. Common options in increasing order of computational complexity and memory and storage requirements include daily, minute, and tick frequency. Intermediate frequencies are also possible. Algorithmic strategies tend to perform better at higher frequencies, even though quantamental investors are gaining ground, as discussed in *Chapter 1*, *Machine Learning for Trading – From Idea to Execution*. Regardless, institutional investors will certainly require tick frequency.

Finally, data ingestion should also address **point-in-time constraints** to avoid look-ahead bias, as outlined in the previous section. The use of trading calendars helps limit data to legitimate dates and times; adjustments to reflect corporate actions like stock splits and dividends or restatements that impact prices revealed at specific times need to be made prior to ingestion.

Factor engineering – built-in factors versus libraries

To facilitate the engineering of alpha factors for use in ML models, many backtesting engines include computational tools suitable for numerous standard transformations like moving averages and various technical indicators. A key advantage of **built-in factor engineering** is the easy conversion of the backtesting pipeline into a live trading engine that applies the same computations to the input data.

The **numerical Python libraries** (pandas, NumPy, TA-Lib) presented in *Chapter 4*, *Financial Feature Engineering – How to Research Alpha Factors*, are an alternative to **pre-compute factors**. This can be efficient when the goal is to reuse factors in various backtests that amortize the computational cost.

ML models, predictions, and signals

As mentioned earlier, the ML workflow discussed in *Chapter 6*, *The Machine Learning Process*, can be embedded in an end-to-end platform that integrates the model design and evaluation part into the backtesting process. While convenient, this is also costly because model training becomes part of the backtest when the goal is perhaps to fine-tune trading rules.

Similar to factor engineering, you can decouple these aspects and design, train, and evaluate ML models using generic libraries for this purpose,

and also provide the relevant predictions as inputs to the backtester. We will mostly use this approach in this book because it makes the exposition more concise and less repetitive.

Trading rules and execution

A realistic strategy simulation requires a faithful representation of the trading environment. This includes access to relevant exchanges, the availability of the various order types discussed in *Chapter 2*, *Market and Fundamental Data – Sources and Techniques*, and the accounting for transaction costs. Costs include broker commissions, bid-ask spreads, and slippage, giving us the difference between the target execution price and the price that's eventually obtained. It is also important to ensure trades execute with delays that reflect liquidity and operating hours.

Performance evaluation

Finally, a backtesting platform needs to facilitate performance evaluation. It can provide standard metrics derived from its accounting of transactions, or provide an output of the metrics that can be used with a library like **pyfolio** that's suitable for this purpose.

In the next two sections, we will explore two of the most popular backtesting libraries, namely backtrader and Zipline.

backtrader – a flexible tool for local backtests

backtrader is a popular, flexible, and user-friendly Python library for local backtests with great documentation, developed since 2015 by Daniel Rodriguez. In addition to a large and active community of individual traders, there are several banks and trading houses that use backtrader to prototype and test new strategies before porting them to a production-ready platform using, for example, Java. You can also use backtrader for live trading with several brokers of your choice (see the backtrader documentation and *Chapter 23*, *Conclusions and Next Steps*).

We'll first summarize the key concepts of backtrader to clarify the big picture of the backtesting workflow on this platform, and then demonstrate its usage for a strategy driven by ML predictions.

Key concepts of backtrader's Cerebro architecture

backtrader's **Cerebro** (Spanish for "brain") architecture represents the key components of the backtesting workflow as (extensible) Python objects. These objects interact to facilitate processing input data and the computation of factors, formulate and execute a strategy, receive and execute orders, and track and measure performance. A Cerebro instance orchestrates the overall process from collecting inputs, executing the backtest bar by bar, and providing results.

The library uses conventions for these interactions that allow you to omit some detail and streamline the backtesting setup. I highly recommend browsing the documentation to dive deeper if you plan on using backtrader to develop your own strategies.

Figure 8.3 outlines the key elements in the Cerebro architecture, and the following subsections summarize their most important functionalities:

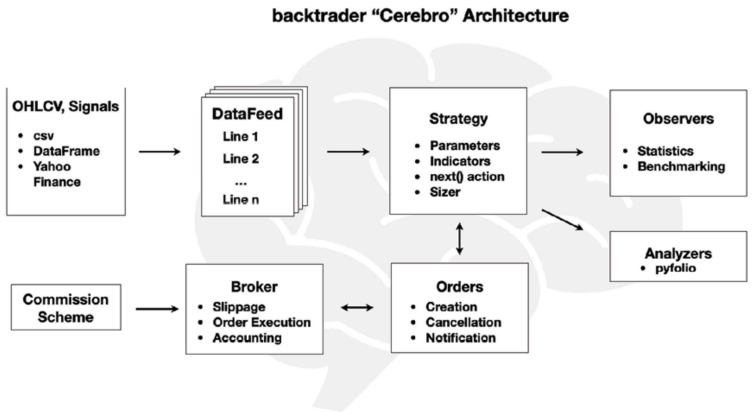

Figure 8.3: The backtrader Cerebro architecture

Data feeds, lines, and indicators

Data feeds are the raw material for a strategy and contain information about individual securities, such as OHLCV market data with a timestamp for each observation, but you can customize the available fields. backtrader can ingest data from various sources, including CSV files and pandas DataFrames, and from online sources like Yahoo Finance. There are also extensions you can use to connect to online trading platforms like Interactive Brokers to ingest live data and execute transactions. The compatibility with DataFrame objects implies that you can load data from accessible by pandas, ranging from databases to HDF5 files. (See the demonstration in the *How to use backtrader in practice* section; also, see the *I/O* section of the pandas documentation.)

Once loaded, we add the data feeds to a Cerebro instance, which, in turn, makes it available to one or more strategies in the order received. Your strategy's trading logic can access each data feed by name (for example, the ticker) or sequence number and retrieve the current and past values of any field of the data feed. Each field is called a **line**.

backtrader comes with over 130 common technical **indicators** that allow you to compute new values from lines or other indicators for each data feed to drive your strategy. You can also use standard Python **operations** to derive new values. Usage is fairly straightforward and well explained in the documentation.

From data and signals to trades – strategy

The **Strategy** object contains your trading logic that places orders based on data feed information that the Cerebro instance presents at every bar during backtest execution. You can easily test variations by configuring a Strategy to accept arbitrary parameters that you define when adding an instance of your Strategy to your Cerebro.

For every bar of a backtest, the Cerebro instance calls either the `.prenext()` or `.next()` method of your Strategy instance. The role of `.prenext()` is to address bars that do not yet have complete data for all feeds, for example, before there are enough periods to compute an indicator like a built-in moving average or if there is otherwise missing data. The default is to do nothing, but you can add trading logic of your choice or call `next()` if your main Strategy is designed to handle missing values (see the *How to use backtrader in practice* section).

You can also use backtrader without defining an explicit Strategy and instead use a simplified Signals interface. The Strategy API gives you more control and flexibility, though; see the backtrader documentation for details on how to use the Signals API.

A Strategy outputs orders: let's see how backtrader handles these next.

Commissions instead of commission schemes

Once your Strategy has evaluated current and past data points at each bar, it needs to decide which orders to place. backtrader lets you create several standard **order** types that Cerebro passes to a Broker instance for execution and provides a notification of the result at each bar.

You can use the Strategy methods `buy()` and `sell()` to place market, close, and limit orders, as well as stop and stop-limit orders. Execution works as follows:

- **Market order**: Fills at the next open bar
- **Close order**: Fills at the next close bar
- **Limit order**: Executes only if a price threshold is met (for example, only buy up to a certain price) during an (optional) period of validity
- **Stop order**: Becomes a market order if the price reaches a given threshold
- **Stop limit order**: Becomes a limit order once the stop is triggered

In practice, stop orders differ from limit orders because they cannot be seen by the market prior to the price trigger. backtrader also provides target orders that compute the required size, taking into account the current position to achieve a certain portfolio allocation in terms of the number of shares, the value of the position, or the percentage of portfolio value. Furthermore, there are **bracket orders** that combine, for a long order, a buy with two limit sell orders that activate as the buy executes. Should one of the sell orders fill or cancel, the other sell order also cancels.

The **Broker** handles order execution, tracks the portfolio, cash value, and notifications and implements transaction costs like commission and slippage. The Broker may reject trades if there is not enough cash; it can be

important to sequence buys and sells to ensure liquidity. backtrader also has a `cheat_on_open` feature that permits looking ahead to the next bar, to avoid rejected trades due to adverse price moves by the next bar. This feature will, of course, bias your results.

In addition to **commission schemes** like a fixed or percentage amount of the absolute transaction value, you can implement your own logic, as demonstrated later, for a flat fee per share.

Making it all happen – Cerebro

The Cerebro control system synchronizes the data feeds based on the bars represented by their timestamp, and runs the trading logic and broker actions on an event-by-event basis accordingly. backtrader does not impose any restrictions on the frequency or the trading calendar and can use multiple time frames in parallel.

It also vectorizes the calculation for indicators if it can preload source data. There are several options you can use to optimize operations from a memory perspective (see the Cerebro documentation for details).

How to use backtrader in practice

We are going to demonstrate backtrader using the daily return predictions from the ridge regression from *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*, as we did for the vectorized backtest earlier in this chapter. We will create the Cerebro instance, load the data, formulate and add the Strategy, run the backtest, and review the results.

The notebook `backtesting_with_backtrader` contains the following code examples and some additional details.

How to load price and other data

We need to ensure that we have price information for all the dates on which we would like to buy or sell stocks, not only for the days with predictions. To load data from a pandas DataFrame, we subclass backtrader's `PandasData` class to define the fields that we will provide:

```
class SignalData(PandasData):
    """
    Define pandas DataFrame structure
    """
    cols = OHLCV + ['predicted']
    # create lines
    lines = tuple(cols)
    # define parameters
    params = {c: -1 for c in cols}
    params.update({'datetime': None})
    params = tuple(params.items())
```

We then instantiate a `Cerebro` class and use the `SignalData` class to add one data feed for each ticker in our dataset that we load from HDF5:

```
cerebro = bt.Cerebro()  # create a "Cerebro" instance
idx = pd.IndexSlice
data = pd.read_hdf('00_data/backtest.h5', 'data').sort_index()
tickers = data.index.get_level_values(0).unique()
for ticker in tickers:
    df = data.loc[idx[ticker, :], :].droplevel('ticker', axis=0)
    df.index.name = 'datetime'
    bt_data = SignalData(dataname=df)
    cerebro.adddata(bt_data, name=ticker)
```

Now, we are ready to define our Strategy.

How to formulate the trading logic

Our `MLStrategy` subclasses backtrader's `Strategy` class and defines parameters that we can use to modify its behavior. We also create a log file to create a record of the transactions:

```
class MLStrategy(bt.Strategy):
    params = (('n_positions', 10),
              ('min_positions', 5),
              ('verbose', False),
              ('log_file', 'backtest.csv'))
    def log(self, txt, dt=None):
        """ Logger for the strategy"""
        dt = dt or self.datas[0].datetime.datetime(0)
        with Path(self.p.log_file).open('a') as f:
            log_writer = csv.writer(f)
            log_writer.writerow([dt.isoformat()] + txt.split(','))
```

The core of the strategy resides in the `.next()` method. We go long/short on the `n_position` stocks with the highest positive/lowest negative forecast, as long as there are at least `min_positions` positions. We always sell any existing positions that do not appear in the new long and short lists and use `order_target_percent` to build equal-weights positions in the new targets (log statements are omitted to save some space):

```
    def prenext(self):
        self.next()
    def next(self):
        today = self.datas[0].datetime.date()
        positions = [d._name for d, pos in self.getpositions().items() if pos]
        up, down = {}, {}
        missing = not_missing = 0
        for data in self.datas:
            if data.datetime.date() == today:
                if data.predicted[0] > 0:
                    up[data._name] = data.predicted[0]
                elif data.predicted[0] < 0:
                    down[data._name] = data.predicted[0]
        # sort dictionaries ascending/descending by value
        # returns list of tuples
        shorts = sorted(down, key=down.get)[:self.p.n_positions]
        longs = sorted(up, key=up.get, reverse=True)[:self.p.n_positions]
        n_shorts, n_longs = len(shorts), len(longs)
```

```
        # only take positions if at least min_n longs and shorts
        if n_shorts < self.p.min_positions or n_longs < self.p.min_positions:
            longs, shorts = [], []
        for ticker in positions:
            if ticker not in longs + shorts:
                self.order_target_percent(data=ticker, target=0)
         short_target = -1 / max(self.p.n_positions, n_short)
        long_target = 1 / max(self.p.top_positions, n_longs)
        for ticker in shorts:
            self.order_target_percent(data=ticker, target=short_target)
        for ticker in longs:
            self.order_target_percent(data=ticker, target=long_target)
```

Now, we need to configure our `Cerebro` instance and add our `Strategy`.

How to configure the Cerebro instance

We use a custom commission scheme that assumes we pay a fixed amount of $0.02 per share that we buy or sell:

```
class FixedCommisionScheme(bt.CommInfoBase):
    """
    Simple fixed commission scheme for demo
    """
    params = (
        ('commission', .02),
        ('stocklike', True),
        ('commtype', bt.CommInfoBase.COMM_FIXED),
    )
    def _getcommission(self, size, price, pseudoexec):
        return abs(size) * self.p.commission
```

Then, we define our starting cash amount and configure the broker accordingly:

```
cash = 10000
cerebro.broker.setcash(cash)
comminfo = FixedCommisionScheme()
cerebro.broker.addcommissioninfo(comminfo)
```

Now, all that's missing is adding the `MLStrategy` to our `Cerebro` instance, providing parameters for the desired number of positions and the minimum number of long/shorts. We'll also add a pyfolio analyzer so we can view the performance tearsheets we presented in *Chapter 5*, *Portfolio Optimization and Performance Evaluation*:

```
cerebro.addanalyzer(bt.analyzers.PyFolio, _name='pyfolio')
cerebro.addstrategy(MLStrategy, n_positions=10, min_positions=5,
                    verbose=True, log_file='bt_log.csv')
results = cerebro.run()
ending_value = cerebro.broker.getvalue()
f'Final Portfolio Value: {ending_value:,.2f}'
Final Portfolio Value: 10,502.32
```

The backtest uses 869 trading days and takes around 45 seconds to run. The following figure shows the cumulative return and the evolution of the portfolio value, as well as the daily value of long and short positions.

Performance looks somewhat similar to the preceding vectorized test, with outperformance relative to the S&P 500 benchmark during the first half and poor performance thereafter.

The `backtesting_with_backtrader` notebook contains the complete pyfolio results:

Figure 8.4: backtrader results

backtrader summary and next steps

backtrader is a very straightforward yet flexible and performant backtesting engine for local backtesting. You can load any dataset at the frequency you desire from a broad range of sources due to pandas compatibility. `Strategy` lets you define arbitrary trading logic; you just need to ensure you access the distinct data feeds as needed. It also integrates well with pyfolio for quick yet comprehensive performance evaluation.

In the demonstration, we applied our trading logic to predictions from a pre-trained model. We can also train a model during backtesting because we can access data prior to the current bar. Often, however, it is more efficient to decouple model training from strategy selection and avoid duplicating model training.

One of the reasons for backtrader's popularity is the ability to use it for live trading with a broker of your choosing. The community is very lively, and code to connect to brokers or additional data sources, including for cryptocurrencies, is readily available online.

Zipline – scalable backtesting by Quantopian

The backtesting engine Zipline powers Quantopian's online research, backtesting, and live (paper) trading platform. As a hedge fund, Quantopian aims to identify robust algorithms that outperform, subject to its risk management criteria. To this end, they use competitions to select the best strategies and allocate capital to share profits with the winners.

Quantopian first released Zipline in 2012 as version 0.5, and the latest version, 1.3, dates from July 2018. Zipline works well with its sister libraries Alphalens, pyfolio, and empyrical that we introduced in *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors* and *Chapter 5, Portfolio Optimization and Performance Evaluation*, and integrates well with NumPy, pandas, and numeric libraries, but may not always support the latest version.

Zipline is designed to operate at the scale of thousands of securities, and each can be associated with a large number of indicators. It imposes more structure on the backtesting process than backtrader to ensure data quality by eliminating look-ahead bias, for example, and optimize computational efficiency while executing a backtest. We'll take a look at the key concepts and elements of the architecture, shown in *Figure 8.5*, before we demonstrate how to use Zipline to backtest ML-driven models on the data of your choice.

Calendars and the Pipeline for robust simulations

Key features that contribute to the goals of scalability and reliability are data bundles that store OHLCV market data with on-the-fly adjustments for splits and dividends, trading calendars that reflect operating hours of exchanges around the world, and the powerful Pipeline API (see the following diagram). We will discuss their usage in the following sections to complement the brief Zipline introduction we gave in earlier chapters:

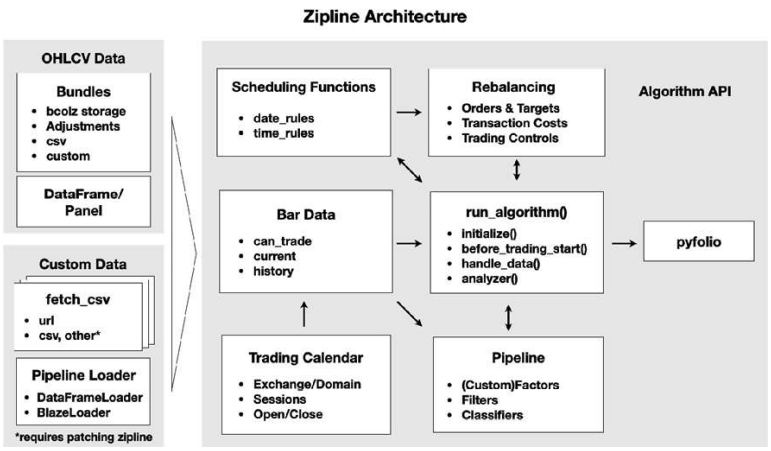

Figure 8.5: The Zipline architecture

Bundles – point-in-time data with on-the-fly adjustments

The principal data store is a **bundle** that resides on disk in compressed, columnar bcolz format for efficient retrieval, combined with metadata stored in an SQLite database. Bundles are designed to contain only OHLCV data and are limited to daily and minute frequency. A great feature is that bundles store split and dividend information, and Zipline computes **point-in-time adjustments**, depending on the time period you pick for your backtest.

Zipline relies on the **TradingCalendar** library (also maintained by Quantopian) for operational details on exchanges around the world, such as time zone, market open and closing times, or holidays. Data sources have domains (for now, these are countries) and need to conform to the assigned exchange calendar. Quantopian is actively developing support for international securities, and these features may evolve.

After installation, the command `zipline ingest -b bundle` lets you install the Quandl Wiki dataset (daily frequency) right away. The result ends up in the `.zipline` directory, which, by default, resides in your home folder. In addition, you can design your own bundles, as we'll see.

In addition to bundles, you can provide OHCLV data to an algorithm as a pandas DataFrame or Panel. (Panel is recently deprecated, but Zipline is a few pandas versions behind.) However, bundles are more convenient and efficient.

A shortcoming of bundles is that they do not let you store data other than price and volume information. However, two alternatives let you accomplish this: the `fetch_csv()` function downloads DataFrames from a URL and was designed for other Quandl data sources, for example, fundamentals. Zipline reasonably expects the data to refer to the same securities for which you have provided OHCLV data and aligns the bars accordingly. It's very easy to patch the library to load a local CSV or HDF5 using pandas, and the GitHub repository provides some guidance on how to do so.

In addition, `DataFrameLoader` and `BlazeLoader` permit you to feed additional attributes to a Pipeline (see the `DataFrameLoader` demo later in this chapter). `BlazeLoader` can interface with numerous sources, including databases. However, since the Pipeline API is limited to daily data, `fetch_csv()` will be critical to adding features at a minute frequency, as we will do in later chapters.

The Algorithm API – backtests on a schedule

The `TradingAlgorithm` class implements the Zipline Algorithm API and operates on `BarData` that has been aligned with a given trading calendar. After the initial setup, the backtest runs for a specified period and executes its trading logic as specific events occur. These events are driven by the daily or minutely trading frequency, but you can also schedule arbitrary functions to evaluate signals, place orders, and rebalance your portfolio, or log information about the ongoing simulation.

You can execute an algorithm from the command line, in a Jupyter Notebook, or by using the `run_algorithm()` method of the underlying `TradingAlgorithm` class. The algorithm requires an `initialize()` method that is called once when the simulation starts. It keeps state through a context dictionary and receives actionable information through a data variable containing point-in-time current and historical data.

You can add properties to the context dictionary, which is available to all other `TradingAlgorithm` methods, or register pipelines that perform

more complex data processing, such as computing alpha factors and filtering securities accordingly.

Algorithm execution occurs through optional methods that are either scheduled automatically by Zipline or at user-defined intervals. The method `before_trading_start()` is called daily before the market opens and primarily serves to identify a set of securities the algorithm may trade during the day. The method `handle_data()` is called at the given trading frequency, for example, every minute.

Upon completion, the algorithm returns a DataFrame containing portfolio performance metrics if there were any trades, as well as user-defined metrics. As demonstrated in *Chapter 5, Portfolio Optimization and Performance Evaluation*, the output is compatible with pyfolio so that you can quickly create performance tearsheets.

Known issues

Zipline currently requires the presence of Treasury curves and the S&P 500 returns for benchmarking (`https://github.com/quantopian/zipline/issues/2480`). The latter relies on the IEX API, which now requires registration to obtain a key. It is easy to patch Zipline to circumvent this and download data from the Federal Reserve, for instance. The GitHub repository describes how to go about this. Alternatively, you can move the SPY returns provided in `zipline/resources/market_data/SPY_benchmark.csv` to your `.zipline` folder, which usually lives in your home directory, unless you changed its location.

Live trading (`https://github.com/zipline-live/zipline`) your own systems is only available with Interactive Brokers and is not fully supported by Quantopian.

Ingesting your own bundles with minute data

We will use the NASDAQ100 2013-17 sample provided by AlgoSeek that we introduced in *Chapter 2, Market and Fundamental Data – Sources and Techniques,* to demonstrate how to write your own custom bundle. There are four steps:

1. Divide your OHCLV data into one file per ticker and store metadata and split and dividend adjustments.
2. Write a script to pass the result to an `ingest()` function, which, in turn, takes care of writing the bundle to bcolz and SQLite format.
3. Register the bundle in an extension.py script that lives in your `.zipline` directory in your home folder, and symlink the data sources.
4. For AlgoSeek data, we also provide a custom TradingCalendar because it includes trading activity outside NYSE market hours.

The directory `custom_bundles` contains the code examples for this section.

Getting your data ready to be bundled

In *Chapter 2, Market and Fundamental Data – Sources and Techniques*, we parsed the daily files containing the AlgoSeek NASDAQ 100 OHLCV data to obtain a time series for each ticker. We will use this result because Zipline also stores each security individually.

In addition, we obtain equity metadata using the pandas DataReader `get_nasdaq_symbols()` function. Finally, since the Quandl Wiki data covers the NASDAQ 100 tickers for the relevant period, we extract the split and dividend adjustments from that bundle's SQLite database.

The result is an HDF5 store containing price and volume data on some 135 tickers, as well as the corresponding meta and adjustment data. The script `algoseek_preprocessing.py` illustrates this process.

Writing your custom bundle ingest function

The Zipline documentation outlines the required parameters for an `ingest()` function, which kicks off the I/O process, but does not provide a lot of practical detail. The script `algoseek_1min_trades.py` shows how to get this part to work for minute data.

There is a `load_equities()` function that provides the metadata, a `ticker_generator()` function that feeds symbols to a `data_generator()`, which, in turn, loads and format each symbol's market data, and an `algoseek_to_bundle()` function, which integrates all the pieces and returns the desired `ingest()` function.

Time zone alignment matters because Zipline translates all data series to UTC; we add US/Eastern time zone information to the OHCLV data and convert it to UTC. To facilitate execution, we create symlinks for this script and the `algoseek.h5` data in the `custom_data` folder in the `.zipline` directory, which we'll add to the `PATH` in the next step so Zipline can find this information.

Registering your bundle

Before we can run `zipline ingest -b algoseek`, we need to register our custom bundle so Zipline knows what we are talking about. To this end, we'll add the following lines to an `extension.py` script in the `.zipline` file, which you may have to create first, alongside some inputs and settings (see the `extension.py` example).

The registration itself is fairly straightforward but highlights a few important details. First, Zipline needs to be able to import the `algoseek_to_bundle()` function, so its location needs to be on the search path, for example, by using `sys.path.append()`. Second, we reference a custom calendar that we will create and register in the next step. Third, we need to inform Zipline that our trading days are longer than the default 6 and a half hours of NYSE days to avoid misalignments:

```
register('algoseek',
        algoseek_to_bundle(),
        calendar_name='AlgoSeek',
        minutes_per_day=960
        )
```

Creating and registering a custom TradingCalendar

As mentioned previously, Quantopian also provides a TradingCalendar library to support trading around the world. The package contains numerous examples, and it is fairly straightforward to subclass one of the examples. Based on the NYSE calendar, we only need to override the open/close times and change the name:

```
class AlgoSeekCalendar(XNYSExchangeCalendar):
    """
    A calendar for trading assets before and after market hours
    Open Time: 4AM, US/Eastern
    Close Time: 19:59PM, US/Eastern
    """
    @property
    def name(self):
        return "AlgoSeek"
    @property
    def open_time(self):
        return time(4, 0)
    @property
    def close_time(self):
        return time(19, 59)
```

We put the definition into `extension.py` and add the following registration:

```
register_calendar(
        'AlgoSeek',
        AlgoSeekCalendar())
```

And now, we can refer to this trading calendar to ensure a backtest includes off-market hour activity.

The Pipeline API – backtesting an ML signal

The Pipeline API facilitates the definition and computation of alpha factors for a cross-section of securities from historical data. Pipeline significantly improves efficiency because it optimizes computations over the entire backtest period, rather than tackling each event separately. In other words, it continues to follow an event-driven architecture but vectorizes the computation of factors where possible.

A pipeline uses factors, filters, and classifiers classes to define computations that produce columns in a table with point-in-time values for a set of securities. Factors take one or more input arrays of historical bar data

and produce one or more outputs for each security. There are numerous built-in factors, and you can also design your own `CustomFactor` computations.

The following diagram depicts how loading the data using `DataFrameLoader`, computing the predictive `MLSignal` using the Pipeline API, and various scheduled activities integrate with the overall trading algorithm that's executed via the `run_algorithm()` function. We'll go over the details and the corresponding code in this section:

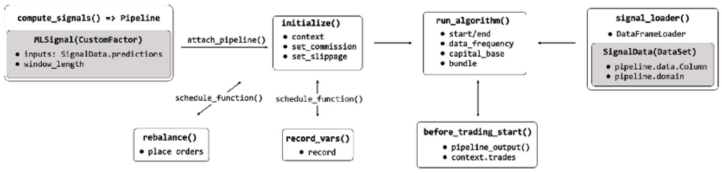

Figure 8.6: ML signal backtest using Zipline's Pipeline API

You need to register your pipeline with the `initialize()` method and execute it at each time step or on a custom schedule. Zipline provides numerous built-in computations, such as moving averages or Bollinger Bands, that can be used to quickly compute standard factors, but it also allows for the creation of custom factors, as we will illustrate next.

Most importantly, the Pipeline API renders alpha factor research modular because it separates the alpha factor computation from the remainder of the algorithm, including the placement and execution of trade orders and the bookkeeping of portfolio holdings, values, and so on.

We'll now illustrate how to load the lasso model daily return predictions, together with price data for our universe, into a pipeline and use a `CustomFactor` to select the top and bottom 10 predictions as long and short positions, respectively. The notebook `backtesting_with_zipline` contains the following code examples.

Our goal is to combine the daily return predictions with the OHCLV data from our Quandl bundle, and then to go long on up to 10 equities with the highest predicted returns and short on those with the lowest predicted returns, requiring at least five stocks on either side, similar to the backtrader example above.

Enabling the DataFrameLoader for our Pipeline

First, we load our predictions for the 2015-17 period and extract the Zipline IDs for the ~250 stocks in our universe during this period using the `bundle.asset_finder.lookup_symbols()` method, as shown in the following code:

```
def load_predictions(bundle):
    predictions = pd.read_hdf('../00_data/backtest.h5', 'data')[['predicted']].dropna()
    tickers = predictions.index.get_level_values(0).unique().tolist()
    assets = bundle.asset_finder.lookup_symbols(tickers, as_of_date=None)
```

```
    predicted_sids = pd.Int64Index([asset.sid for asset in assets])
    ticker_map = dict(zip(tickers, predicted_sids))
    return (predictions
            .unstack('ticker')
            .rename(columns=ticker_map)
            .predicted
            .tz_localize('UTC')), assets
bundle_data = bundles.load('quandl')
predictions, assets = load_predictions(bundle_data)
```

To make the predictions available to the Pipeline API, we need to define a `Column` with a suitable data type for a `DataSet` with an appropriate `domain`, like so:

```
class SignalData(DataSet):
    predictions = Column(dtype=float)
    domain = US_EQUITIES
```

While the bundle's OHLCV data can rely on the built-in `USEquityPricingLoader`, we need to define our own `DataFrameLoader`, as follows:

```
signal_loader = {SignalData.predictions:
                     DataFrameLoader(SignalData.predictions, predictions)}
```

In fact, we need to slightly modify the Zipline library's source code to bypass the assumption that we will only load price data. To this end, we add a `custom_loader` parameter to the `run_algorithm` method and ensure that this loader is used when the pipeline needs one of SignalData's `Column` instances.

Creating a pipeline with a custom ML factor

Our pipeline is going to have two Boolean columns that identify the assets we would like to trade as long and short positions. To get there, we first define a `CustomFactor` called `MLSignal` that just receives the current return predictions. The motivation is to allow us to use some of the convenient `Factor` methods designed to rank and filter securities:

```
class MLSignal(CustomFactor):
    """Converting signals to Factor
        so we can rank and filter in Pipeline"""
    inputs = [SignalData.predictions]
    window_length = 1
    def compute(self, today, assets, out, preds):
        out[:] = preds
```

Now, we can set up our actual pipeline by instantiating `CustomFactor`, which requires no arguments other than the defaults provided. We combine its `top()` and `bottom()` methods with a filter to select the highest positive and lowest negative predictions:

```
def compute_signals():
    signals = MLSignal()
    return Pipeline(columns={
        'longs' : signals.top(N_LONGS, mask=signals > 0),
        'shorts': signals.bottom(N_SHORTS, mask=signals < 0)},
            screen=StaticAssets(assets))
```

The next step is to initialize our algorithm by defining a few context variables, setting transaction cost parameters, performing schedule rebalancing and logging, and attaching our pipeline:

```
def initialize(context):
    """
    Called once at the start of the algorithm.
    """
    context.n_longs = N_LONGS
    context.n_shorts = N_SHORTS
    context.min_positions = MIN_POSITIONS
    context.universe = assets
    set_slippage(slippage.FixedSlippage(spread=0.00))
    set_commission(commission.PerShare(cost=0, min_trade_cost=0))
    schedule_function(rebalance,
                      date_rules.every_day(),
                      time_rules.market_open(hours=1, minutes=30))
    schedule_function(record_vars,
                      date_rules.every_day(),
                      time_rules.market_close())
    pipeline = compute_signals()
    attach_pipeline(pipeline, 'signals')
```

Every day before the market opens, we run our pipeline to obtain the latest predictions:

```
def before_trading_start(context, data):
    """
    Called every day before market open.
    """
    output = pipeline_output('signals')
    context.trades = (output['longs'].astype(int)
                      .append(output['shorts'].astype(int).mul(-1))
                      .reset_index()
                      .drop_duplicates()
                      .set_index('index')
                      .squeeze())
```

After the market opens, we place orders for our long and short targets and close all other positions:

```
def rebalance(context, data):
    """
    Execute orders according to schedule_function() date & time rules.
    """
    trades = defaultdict(list)
    for stock, trade in context.trades.items():
```

```
            if not trade:
                order_target(stock, 0)
            else:
                trades[trade].append(stock)
        context.longs, context.shorts = len(trades[1]), len(trades[-1])
        if context.longs > context.min_positions and context.shorts > context.min_positions:
            for stock in trades[-1]:
                order_target_percent(stock, -1 / context.shorts)
            for stock in trades[1]:
                order_target_percent(stock, 1 / context.longs)
```

Now, we are ready to execute our backtest and pass the results to pyfolio:

```
results = run_algorithm(start=start_date,
                        end=end_date,
                        initialize=initialize,
                        before_trading_start=before_trading_start,
                        capital_base=1e6,
                        data_frequency='daily',
                        bundle='quandl',
                        custom_loader=signal_loader) # need to modify zipline
returns, positions, transactions = pf.utils.extract_rets_pos_txn_from_zipline(results)
```

Figure 8.7 shows the plots for the strategy's cumulative returns (left panel) and the rolling Sharpe ratio, which are comparable to the previous backtrader example.

The backtest only takes around half the time, though:

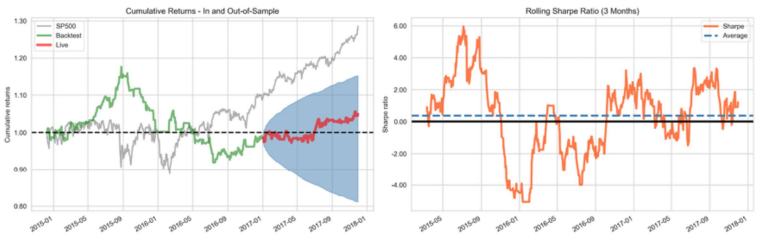

Figure 8.7: Zipline backtest results

The notebook `backtesting_with_zipline` contains the full pyfolio tearsheet with additional metrics and plots.

How to train a model during the backtest

We can also integrate the model training into our backtest. You can find the code for the following end-to-end example of our ML4T workflow in the `ml4t_with_zipline` notebook:

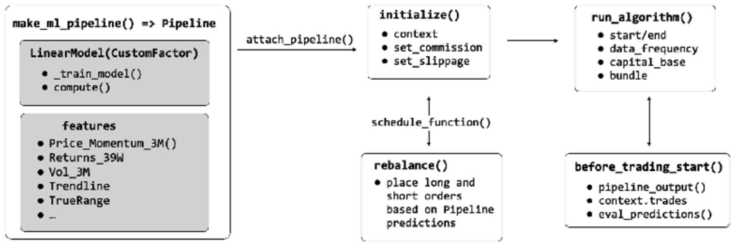

Figure 8.8: Flowchart of Zipline backtest with model training

The goal is to roughly replicate the ridge regression daily return predictions we used earlier and generated in *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*. We will, however, use a few additional pipeline factors to illustrate their usage. The principal new element is a `CustomFactor` that receives features and returns them as inputs to train a model and produce predictions.

Preparing the features – how to define pipeline factors

To create a **pipeline factor**, we need one or more input variables, a `window_length` that indicates the number of most recent data points for each input and security, and the computation we want to conduct.

A linear price trend that we estimate using linear regression (see *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*) works as follows: we use the 252 latest close prices to compute the regression coefficient on a linear time trend:

```
class Trendline(CustomFactor):
    # Linear 12-month price trend regression
    inputs = [USEquityPricing.close]
    window_length = 252
    def compute(self, today, assets, out, close):
        X = np.arange(self.window_length).reshape(-1, 1).astype(float)
        X -= X.mean()
        Y = close - np.nanmean(close, axis=0)
        out[:] = (X.T @ Y / np.var(X)) / self.window_length
```

We will use 10 custom and built-in factors as features for our model to capture risk factors like momentum and volatility (see notebook `ml4t_with_zipline` for details). Next, we'll come up with a `CustomFactor` that trains our model.

How to design a custom ML factor

Our `CustomFactor`, called `ML`, will have `StandardScaler` and a **stochastic gradient descent** (**SGD**) implementation of ridge regression as instance attributes, and we will train the model 3 days a week:

```
class LinearModel(CustomFactor):
    """Obtain model predictions"""
    train_on_weekday = [0, 2, 4]
    def __init__(self, *args, **kwargs):
```

```
        super().__init__(self, *args, **kwargs)
        self._scaler = StandardScaler()
        self._model = SGDRegressor(penalty='L2')
        self._trained = False
```

The `compute` method generates predictions (addressing potential missing values), but first checks if the model should be trained:

```
    def _maybe_train_model(self, today, returns, inputs):
        if (today.weekday() in self.train_on_weekday) or not self._trained:
            self._train_model(today, returns, inputs)
    def compute(self, today, assets, out, returns, *inputs):
        self._maybe_train_model(today, returns, inputs)
        # Predict most recent feature values
        X = np.dstack(inputs)[-1]
        missing = np.any(np.isnan(X), axis=1)
        X[missing, :] = 0
        X = self._scaler.transform(X)
        preds = self._model.predict(X)
        out[:] = np.where(missing, np.nan, preds)
```

The `_train_model` method is the centerpiece of the puzzle. It shifts the returns and aligns the resulting forward returns with the factor features, removing missing values in the process. It scales the remaining data points and trains the linear `SGDRegressor`:

```
    def _train_model(self, today, returns, inputs):
        scaler = self._scaler
        model = self._model
        shift_by = N_FORWARD_DAYS + 1
        outcome = returns[shift_by:].flatten()
        features = np.dstack(inputs)[:-shift_by]
        n_days, n_stocks, n_features = features.shape
        features = features.reshape(-1, n_features)
        features = features[~np.isnan(outcome)]
        outcome = outcome[~np.isnan(outcome)]
        outcome = outcome[np.all(~np.isnan(features), axis=1)]
        features = features[np.all(~np.isnan(features), axis=1)]
        features = scaler.fit_transform(features)
        model.fit(X=features, y=outcome)
        self._trained = True
```

The `make_ml_pipeline()` function preprocesses and combines the outcome, feature, and model parts into a pipeline with a column for predictions:

```
def make_ml_pipeline(universe, window_length=21, n_forward_days=5):
    pipeline_columns = OrderedDict()
    # ensure that returns is the first input
    pipeline_columns['Returns'] = Returns(inputs=[USEquityPricing.open],
                                          mask=universe,
                                          window_length=n_forward_days + 1)
    # convert factors to ranks; append to pipeline
    pipeline_columns.update({k: v.rank(mask=universe)
                             for k, v in features.items()})
```

```
    # Create ML pipeline factor.
    # window_length = length of the training period
    pipeline_columns['predictions'] = LinearModel(
        inputs=pipeline_columns.values(),
        window_length=window_length + n_forward_days,
        mask=universe)
    return Pipeline(screen=universe, columns=pipeline_columns)
```

Tracking model performance during a backtest

We obtain new predictions using the `before_trading_start()` function, which runs every morning before the market opens:

```
def before_trading_start(context, data):
    output = pipeline_output('ml_model')
    context.predicted_returns = output['predictions']
    context.predicted_returns.index.rename(['date', 'equity'], inplace=True)
    evaluate_predictions(output, context)
```

`evaluate_predictions` does exactly this: it tracks the past predictions of our model and evaluates them once returns for the relevant time horizon materialize (in our example, the next day):

```
def evaluate_predictions(output, context):
    # Look at past predictions to evaluate model performance out-of-sample
    # A day has passed, shift days and drop old ones
    context.past_predictions = {
        k - 1: v for k, v in context.past_predictions.items() if k > 0}
    if 0 in context.past_predictions:
        # Use today's forward returns to evaluate predictions
        returns, predictions = (output['Returns'].dropna()
                                .align(context.past_predictions[0].dropna(),
                                       join='inner'))
        if len(returns) > 0 and len(predictions) > 0:
            context.ic = spearmanr(returns, predictions)[0]
            context.rmse = np.sqrt(
                mean_squared_error(returns, predictions))
            context.mae = mean_absolute_error(returns, predictions)
            long_rets = returns[predictions > 0].mean()
            short_rets = returns[predictions < 0].mean()
            context.returns_spread_bps = (
                long_rets - short_rets) * 10000
    # Store current predictions
    context.past_predictions[N_FORWARD_DAYS] = context.predicted_returns
```

We also record the evaluation on a daily basis so we can review it after the backtest:

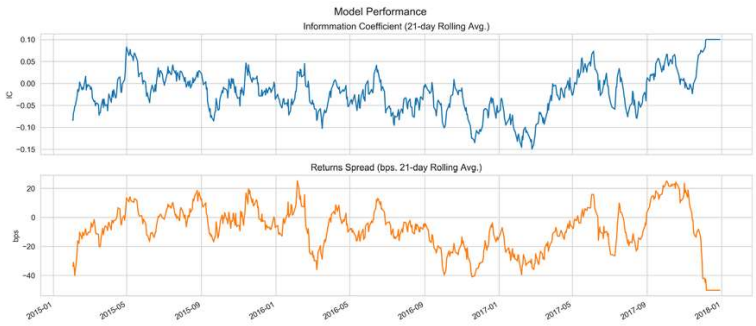

Figure 8.9: Model out-of-sample performance

The following plots summarize the backtest performance in terms of the cumulative returns and the rolling SR. The results have improved relative to the previous example (due to a different feature set), yet the model still underperforms the benchmark since mid-2016:

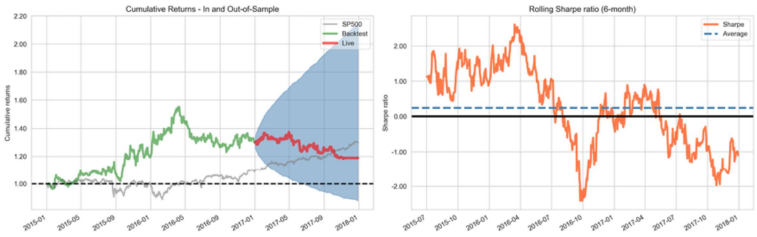

Figure 8.10: Zipline backtest performance with model training

Please see the notebook for additional details on how we define a universe, run the backtest, and rebalance and analyze the results using pyfolio.

Instead of how to use

The notebook `ml4t_quantopian` contains an example of how to backtest a strategy that uses a simple ML model in the Quantopian research environment. The key benefit of using Zipline in the Quantopian cloud is access to many additional datasets, including fundamental and alternative data. See the notebook for more details on the various factors that we can derive in this context.

Summary

In this chapter, we took a much closer look at how backtesting works, what challenges there are, and how to manage them. We demonstrated how to use the two popular backtesting libraries, backtrader and Zipline.

Most importantly, however, we walked through the end-to-end process of designing and testing an ML model, showed you how to implement trading logic that acts on the signals provided by the model's predictions, and saw how to conduct and evaluate backtests. Now, we are ready to con-

tinue exploring a much broader and more sophisticated array of ML models than the linear regressions we started with.

The next chapter will cover how to incorporate the time dimension into our models.