4
Financial Feature Engineering – How to Research Alpha Factors

Algorithmic trading strategies are driven by signals that indicate when to buy or sell assets to generate superior returns relative to a benchmark, such as an index. The portion of an asset's return that is not explained by exposure to this benchmark is called **alpha**, and hence the signals that aim to produce such uncorrelated returns are also called **alpha factors**.

If you are already familiar with ML, you may know that feature engineering is a key ingredient for successful predictions. This is no different in trading. Investment, however, is particularly rich in decades of research into how markets work, and which features may work better than others to explain or predict price movements as a result. This chapter provides an overview as a starting point for your own search for alpha factors.

This chapter also presents key tools that facilitate computing and testing alpha factors. We will highlight how the NumPy, pandas, and TA-Lib libraries facilitate the manipulation of data and present popular smoothing techniques like the wavelets and the Kalman filter, which help reduce noise in data.

We will also preview how you can use the trading simulator Zipline to evaluate the predictive performance of (traditional) alpha factors. We will discuss key alpha factor metrics like the information coefficient and factor turnover. An in-depth introduction to backtesting trading strategies that use machine learning follows in *Chapter 6*, *The Machine Learning Process*, which covers the ML4T workflow that we will use throughout this book to evaluate trading strategies.

In particular, this chapter will address the following topics:

- Which categories of factors exist, why they work, and how to measure them
- Creating alpha factors using NumPy, pandas, and TA-Lib
- How to denoise data using wavelets and the Kalman filter
- Using Zipline offline and on Quantopian to test individual and multiple alpha factors
- How to use Alphalens to evaluate predictive performance and turnover using, among other metrics, the **information coefficient** (**IC**)

> 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. The *Appendix, Alpha Factor Library*, contains additional information on financial feature engineering, including more than 100 worked examples that you can leverage for your own strategy..

Alpha factors in practice – from data to signals

Alpha factors are transformations of raw data that aim to predict asset price movements. They are designed to **capture risks that drive asset returns**. A factor may combine one or several inputs, but outputs a single value for each asset, every time the strategy evaluates the factor to obtain a signal. Trade decisions may rely on relative factor values across assets or patterns for a single asset.

The design, evaluation, and combination of alpha factors are critical steps during the research phase of the algorithmic trading strategy workflow, which is displayed in *Figure 4.1*:

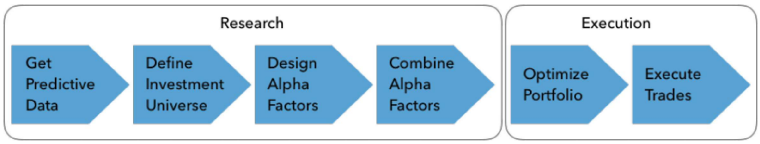

Figure 4.1: Alpha factor research and execution workflow

This chapter focuses on the research phase; the next chapter covers the execution phase. The remainder of this book will then focus on how to leverage ML to learn new factors from data and effectively aggregate the signals from multiple alpha factors.

Alpha factors are transformations of market, fundamental, and alternative data that contain predictive signals. Some factors describe fundamental, economy-wide variables such as growth, inflation, volatility, productivity, and demographic risk. Other factors represent investment styles, such as value or growth, and momentum investing that can be traded and are thus priced by the market. There are also factors that explain price movements based on the economics or institutional setting of financial markets, or investor behavior, including known biases of this behavior.

The economic theory behind factors can be **rational** so that factors have high returns over the long run to compensate for their low returns during bad times. It can also be **behavioral**, where factor risk premiums result from the possibly biased, or not entirely rational, behavior of agents that is not arbitraged away.

There is a constant search for and discovery of new factors that may better capture known or reflect new drivers of returns. Jason Hsu, the co-founder of Research Affiliates, which manages close to $200 billion, identified some 250 factors that had been published with empirical evidence in reputable journals by 2015. He estimated that this number was likely to increase by 40 factors per year.

To avoid false discoveries and ensure a factor delivers consistent results, it should have a meaningful **economic intuition** based on the various established factor categories like momentum, value, volatility, or quality and their rationales, which we'll outline in the next section. This makes it more plausible that the factor reflects risks for which the market would compensate.

Alpha factors result from transforming raw market, fundamental, or alternative data using simple arithmetic, such as absolute or relative changes of a variable over time, ratios between data series, or aggregations over a time window like a simple or exponential moving average. They also include metrics that have emerged from the technical analysis of price and volume patterns, such as the **relative strength index** of demand versus supply and numerous metrics familiar from the fundamental analysis of securities. Kakushadze (2016) lists the formulas for 101 alpha factors, 80 percent of which were used in production at the WorldQuant hedge fund at the time of writing.

Historically, trading strategies applied simple ranking heuristics, value thresholds, or quantile cutoffs to one or several alpha factors computed across multiple securities in the investment universe. Examples include the value investing approach popularized in one of Warren Buffet's favorite books, *Security Analysis*, by Graham and Dodd (1934), which relies on metrics like the book-to-market ratio.

Modern research into alpha factors that predict above-market returns has been led by Eugene Fama (who won the 2013 Nobel Prize in Economics) and Kenneth French, who provided evidence on the size and value factors (1993). This work led to the three- and five-factor models, which we will discuss in *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*, using daily data on factor returns provided by the authors on their website. An excellent, more recent, overview of modern factor investing has been written by Andrew Ang (2014), who heads this discipline at BlackRock, which manages close to $7 trillion.

As we will see throughout this book, ML has proven quite effective in learning to extract signals directly from a more diverse and much larger set of input data without using prescribed formulas. As we will also see, however, alpha factors remain useful inputs for an ML model that combines their information content in a more optimal way than manually set rules.

As a result, algorithmic trading strategies today leverage a large number of signals, many of which may be weak individually but can yield reliable predictions when combined with other model-driven or traditional factors by an ML algorithm.

Building on decades of factor research

In an idealized world, risk factors should be independent of each other, yield positive risk premia, and form a complete set that spans all dimensions of risk and explains the systematic risks for assets in a given class.

In practice, these requirements hold only approximately, and there are important correlations between different factors. For instance, momentum is often stronger among smaller firms (Hou, Xue, and Zhang, 2015). We will show how to derive synthetic, data-driven risk factors using unsupervised learning—in particular, principal and independent component analysis —in *Chapter 13*, *Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*.

In this section, we will review a few key factor categories prominent in financial research and trading applications, explain their economic rationale, and present metrics typically used to capture these drivers of returns.

In the next section, we will demonstrate how to implement some of these factors using NumPy and pandas, use the TA-Lib library for technical analysis, and demonstrate how to evaluate factors using the Zipline backtesting library. We will also highlight some factors built into Zipline that are available on the Quantopian platform.

Momentum and sentiment – the trend is your friend

Momentum investing is among the most well-established factor strategies, underpinned by quantitative evidence since Jegadeesh and Titman (1993) for the US equity market. It follows the adage: *the trend is your friend or let your winners run*. Momentum factors are designed to go long on assets that have performed well, while going short on assets with poor performance over a certain period. Clifford Asness, the founder of the $200 billion hedge fund AQR, presented evidence for momentum effects across eight different asset classes and markets much more recently (Asness, Moskowitz, and Pedersen, 2013).

The premise of strategies using this factor is that **asset prices exhibit a trend**, reflected in positive serial correlation. Such price momentum defies the hypothesis of efficient markets, which states that past price returns alone cannot predict future performance. Despite theoretical arguments to the contrary, price momentum strategies have produced positive returns across asset classes and are an important part of many trading strategies.

The chart in *Figure 4.2* shows the historical performance of portfolios formed based on their exposure to various alpha factors (using data from the Fama-French website). The factor **winner minus loser (WML)** represents the difference in performance between portfolios containing US stocks in the top and bottom three deciles, respectively, of the prior 2-12 months of returns:

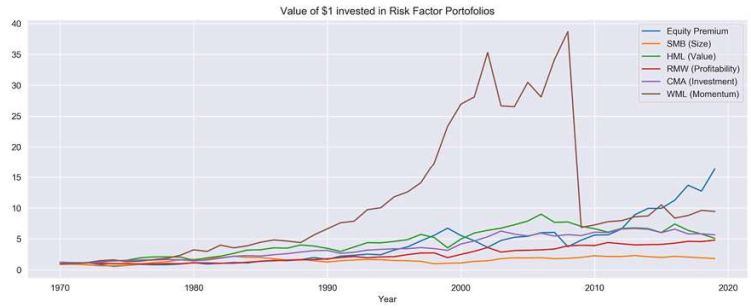

Figure 4.2: Returns on various risk factors

The momentum factor dramatically outperformed other prominent risk factors up to the 2008 crisis. The other factors include the **high-minus-low** (**HML**) value factor, the **robust-minus-weak** (**RMW**) profitability factor, and the **conservative-minus-aggressive** (**CMA**) investment factor. The equity premium is the difference between the market return (for example, the S&P 500) and the risk-free rate.

Why might momentum and sentiment drive excess returns?

Reasons for the momentum effect point to investor behavior, persistent supply and demand imbalances, a positive feedback loop between risk assets and the economy, or the market microstructure.

The **behavioral rationale** reflects the biases of underreaction (Hong, Lim, and Stein, 2000) and over-reaction (Barberis, Shleifer, and Vishny, 1998) to market news as investors process new information at different speeds. After an initial under-reaction to news, investors often extrapolate past behavior and create price momentum. The technology stocks rally during the late 90s market bubble was an extreme example. A fear and greed psychology also motivates investors to increase exposure to winning assets and continue selling losing assets (Jegadeesh and Titman, 2011).

Momentum can also have **fundamental drivers** such as a positive feedback loop between risk assets and the economy. Economic growth boosts equities, and the resulting wealth effect feeds back into the economy through higher spending, again fueling growth. Positive feedback between prices and the economy often extends momentum in equities and credit to longer horizons than for bonds, FOEX, and commodities, where negative feedback creates reversals, requiring a much shorter investment horizon. Another cause of momentum can be persistent demand-supply imbalances due to market frictions. One example is the delay of commodity production in adjusting to changing demand. Oil production may lag higher demand from a booming economy for years, and persistent supply shortages can trigger and support upward price momentum (Novy-Marx, 2015).

Over shorter, intraday horizons, **market microstructure** effects can also create price momentum as investors implement strategies that mimic their biases. For example, the trading wisdom to cut losses and let profits run has investors use trading strategies such as stop-loss, **constant pro-**

portion portfolio insurance (**CPPI**), dynamical delta hedging, or option-based strategies such as protective puts. These strategies create momentum because they imply an advance commitment to sell when an asset underperforms and buy when it outperforms.

Similarly, risk parity strategies (see the next chapter) tend to buy low-volatility assets that often exhibit positive performance and sell high-volatility assets that often have had negative performance (see the *Volatility and size anomalies* section later in this chapter). The automatic rebalancing of portfolios using these strategies tends to reinforce price momentum.

How to measure momentum and sentiment

Momentum factors are typically derived from changes in price time series by identifying trends and patterns. They can be constructed based on absolute or relative return by comparing a cross-section of assets or analyzing an asset's time series, within or across traditional asset classes, and at different time horizons.

A few popular illustrative indicators are listed in the following table (see the *Appendix* for formulas):

Factor	Description
Relative strength index (RSI)	RSI compares the magnitude of recent price changes across stocks to identify stocks as overbought or oversold. A high RSI (usually above 70) indicates overbought and a low RSI (typically below 30) indicates oversold. It first computes the average price change for a given number (often 14) of prior trading days with rising prices and falling prices , respectively, to compute $\text{RSI} = 100 - \frac{100}{1 + \frac{\Delta_p^{up}}{\Delta_p^{down}}}$.
Price momentum	This factor computes the total return for a given number of prior trading days. In academic literature, it is common to use the last 12 months except for the most recent month due to a short-term reversal effect that's frequently observed. However, shorter periods have also been widely used.
12-month price momentum volume adjustment	The indicator normalizes the total return over the previous 12 months by dividing it by the standard deviation of these returns.

Price acceleration	Price acceleration calculates the gradient of the price trend (adjusted for volatility) using linear regression on daily prices for a longer and a shorter period, for example, 1 year and 3 months of trading days, and compares the change in the slope as a measure of price acceleration.
Percent off 52-week high	This factor uses the percent difference between the most recent and the highest price for the last 52 weeks.

Additional sentiment indicators include the following metrics; inputs like analyst estimates can be obtained from data providers like Quandl or Bloomberg, among others:

Factor	Description
Earnings estimates count	This metric ranks stocks by the number of consensus estimates as a proxy for analyst coverage and information uncertainty. A higher value is more desirable.
N-month change in recommendation	This factor ranks stocks by the change in consensus recommendation over the prior *N* month, where improvements are desirable (regardless of whether they have moved from strong sell to sell or buy to strong buy and so on).
12-month change in shares outstanding	This factor measures the change in a company's split-adjusted share count over the last 12 months, where a negative change implies share buybacks and is desirable because it signals that management views the stock as cheap relative to its intrinsic and, hence, future value.
6-month change in target price	The metric tracks the 6-month change in mean analyst target price. A higher positive change is naturally more desirable.
Net earnings revisions	This factor expresses the difference between upward and downward revisions to earnings estimates as a percentage of the total number of revisions.
Short interest to shares outstanding	This measure is the percentage of shares outstanding currently being sold short, that is, sold by an investor who has borrowed the share and needs to repurchase it at a later day while speculating that its price will fall. Hence, a high level of short interest indicates negative sentiment and is expected to signal poor performance going forward.

There are also numerous data providers that aim to offer sentiment indicators constructed from social media, such as Twitter. We will create our own sentiment indicators using **natural language processing** in Part 3 of this book.

Value factors – hunting fundamental bargains

Stocks with low prices relative to their fundamental value tend to deliver returns in excess of a capitalization-weighted benchmark. Value factors reflect this correlation and are designed to send buy signals for undervalued assets that are relatively cheap and sell signals for overvalued assets. Hence, at the core of any value strategy is a model that estimates the asset's fair or fundamental value. Fair value can be defined as an absolute price level, a spread relative to other assets, or a range in which an asset should trade.

Relative value strategies

Value strategies rely on the mean-reversion of prices to the asset's fair value. They assume that prices only temporarily move away from fair value due to behavioral effects like overreaction or herding, or liquidity effects such as temporary market impact or long-term supply/demand friction. Value factors often exhibit properties opposite to those of momentum factors because they rely on mean-reversion. For equities, the opposite of value stocks is growth stocks that have a high valuation due to growth expectations.

Value factors enable a broad array of systematic strategies, including fundamental and market valuation and cross-asset relative value. They are often collectively labeled **statistical arbitrage** (**StatArb**) strategies, implemented as market-neutral long/short portfolios without exposure to other traditional or alternative risk factors.

Fundamental value strategies

Fundamental value strategies derive fair asset values from economic and fundamental indicators that depend on the target asset class. In fixed income, currencies, and commodities, indicators include levels and changes in the capital account balance, economic activity, inflation, or fund flows. For equities and corporate credit, value factors go back to Graham and Dodd's previously mentioned *Security Analysis*. Equity value approaches compare a stock price to fundamental metrics such as book value, top-line sales, bottom-line earnings, or various cash-flow metrics.

Market value strategies

Market value strategies use statistical or machine learning models to identify mispricing due to inefficiencies in liquidity provision. Statistical and index arbitrage are prominent examples that capture the reversion of temporary market impacts over short time horizons. (We will cover pairs trading in *Chapter 9*, *Time-Series Models for Volatility Forecasts and Statistical Arbitrage*). Over longer time horizons, market value trades also leverage seasonal effects in equities and commodities.

Cross-asset relative value strategies

Cross-asset relative value strategies focus on mispricing across asset classes. For example, convertible bond arbitrage involves trades on the relative value between the bond that can be turned into equity and the underlying stock of a single company. Relative value strategies also include trades between credit and equity volatility, using credit signals to trade equities or trades between commodities and related equities.

Why do value factors help predict returns?

There are both rational and behavioral explanations for the existence of the **value effect**, defined as the excess return on a portfolio of value stocks relative to a portfolio of growth stocks, where the former have a low market value and the latter have a high market value relative to fundamentals. We will cite a few prominent examples from a wealth of research (see, for example, Fama and French, 1998, and Asness, Moskowitz, and Pedersen, 2013).

In the **rational**, **efficient markets view**, the value premium compensates for higher real or perceived risks. Researchers have presented evidence that value firms have less flexibility to adapt to the unfavorable economic environments than leaner and more flexible growth companies, or that value stock risks relate to high financial leverage and more uncertain future earnings. Value and small-cap portfolios have also been shown to be more sensitive to macro shocks than growth and large-cap portfolios (Lakonishok, Shleifer, and Vishny, 1994).

From a **behavioral perspective**, the value premium can be explained by loss aversion and mental accounting biases. Investors may be less concerned about losses on assets with a strong recent performance due to the cushions offered by prior gains. This loss aversion bias induces investors to perceive the stock as less risky than before and discount its future cash flows at a lower rate. Conversely, poor recent performance may lead investors to raise the asset's discount rate.

These **differential return expectations** can produce a value premium: growth stocks with a high price multiple relative to fundamentals have done well in the past, but investors will require a lower average return going forward due to their biased perception of lower risks, while the inverse is true for value stocks.

How to capture value effects

A large number of valuation proxies are computed from fundamental data. These factors can be combined as inputs into a machine learning valuation model to predict asset prices. The following examples apply to equities, and we will see how some of these factors are used in the following chapters:

Factor	Description

Cash flow yield	The ratio divides the operational cash flow per share by the share price. A higher ratio implies better cash returns for shareholders (if paid out using dividends or share buybacks or profitably reinvested in the business).
Free cash flow yield	The ratio divides the free cash flow per share, which reflects the amount of cash available for distribution after necessary expenses and investments, by the share price. Higher and growing free cash flow yield is commonly viewed as a signal of outperformance.
Cash flow return on invested capital (CFROIC)	CFROIC measures a company's cash flow profitability. It divides operating cash flow by invested capital, defined as total debt plus net assets. A higher return means the business has more cash for a given amount of invested capital, generating more value for shareholders.
Cash flow to total assets	This ratio divides operational cash flow by total assets and indicates how much cash a company can generate relative to its assets, where a higher ratio is better, as with CFROIC.
Free cash flow to enterprise value	This ratio measures the free cash flow that a company generates relative to its enterprise value, measured as the combined value of equity and debt. The debt and equity values can be taken from the balance sheet, but market values often provide a more accurate picture assuming the corresponding assets are actively traded.
EBITDA to enterprise value	This ratio measures a company's **earnings before interest, taxes, depreciation, and amortization (EBITDA)**, which is a proxy for cash flow relative to its enterprise value.
Earnings yield	This ratio divides the sum of earnings for the past 12 months by the last market (close) price.
Earnings yield 1-year forward	Instead of using historical earnings, this ratio divides the average of earnings forecasted by stock analyst for the next 12 months by the last price.
PEG ratio	The **price/earnings to growth (PEG)** ratio divides a stock's **price-to-earnings (P/E)** ratio by the earnings growth rate for a given period. The ratio adjusts the price paid for a dollar of earnings (measured by the P/E ratio) by the company's earnings growth.
P/E 1-year forward rel-	Forecasts the P/E ratio relative to the corresponding sector P/E. It aims to alleviate the sector bias of the

ative to the sector	generic P/E ratio by accounting for sector differences in valuation.
Sales yield	The ratio measures the valuation of a stock relative to its ability to generate revenues. All else being equal, stocks with higher historical sales to price ratios are expected to outperform.
Sales yield forward	The forward sales-to-price ratio uses analyst sales forecast, combined to a (weighted) average.
Book value yield	The ratio divides the historical book value by the share price.
Dividend yield	The current annualized dividend divided by the last close price. Discounted cash flow valuation assumes a company's market value equates to the present value of its future cash flows.

Chapter 2, *Market and Fundamental Data – Sources and Techniques*, discussed how you can source the fundamental data used to compute these metrics from company filings.

Volatility and size anomalies

The **size effect** is among the older risk factors and relates to the excess performance of stocks with a low market capitalization (see *Figure 4.2* at the beginning of this section). More recently, the **low-volatility factor** has been shown to capture excess returns on stocks with below-average volatility, beta, or idiosyncratic risk. Stocks with a larger market capitalization tend to have lower volatility so that the traditional size factor is often combined with the more recent volatility factor.

The low volatility anomaly is an empirical puzzle that is at odds with the basic principles of finance. The **capital asset pricing model** (**CAPM**) and other asset pricing models assert that higher risk should earn higher returns (as we will discuss in detail in the next chapter), but in numerous markets and over extended periods, the opposite has been true, with less risky assets outperforming their riskier peers.

Figure 4.3 plots a rolling mean of the S&P 500 returns of 1990-2019 against the VIX index, which measures the implied volatility of at-the-money options on the S&P 100. It illustrates how stock returns and this measure of volatility have moved inversely with a negative correlation of -.54 over this period. In addition to this aggregate effect, there is also evidence that stocks with a greater sensitivity to changes in the VIX perform worse (Ang et al. 2006):

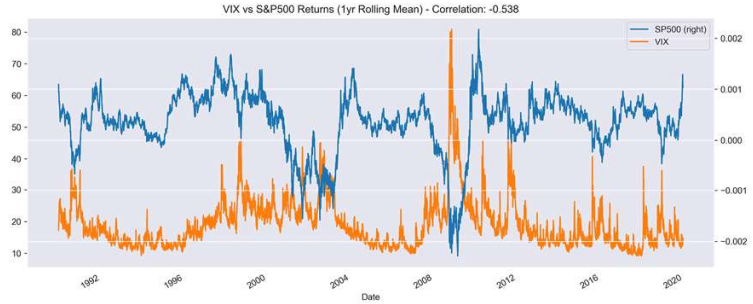

Figure 4.3: Correlation between the VIX and the S&P 500

Why do volatility and size predict returns?

The low volatility anomaly contradicts the hypothesis of efficient markets and the CAPM assumptions. Several behavioral explanations have been advanced to explain its existence.

The **lottery effect** builds on empirical evidence that individuals take on bets that resemble lottery tickets with a small expected loss but a large potential win, even though this large win may have a fairly low probability. If investors perceive that the risk-return profile of a low price, volatile stock is like a lottery ticket, then it could be an attractive bet. As a result, investors may overpay for high-volatility stocks and underpay for low-volatility stocks due to their biased preferences.

The **representativeness bias** suggests that investors extrapolate the success of a few, well-publicized volatile stocks to all volatile stocks while ignoring the speculative nature of such stocks.

Investors may also be **overconfident** in their ability to forecast the future, and their differences in opinions are higher for volatile stocks with more uncertain outcomes. Since it is easier to express a positive view by going long—that is, owning an asset—than a negative view by going short, optimists may outnumber pessimists and keep driving up the price of volatile stocks, resulting in lower returns.

Furthermore, investors behave differently during bull markets and crises. During bull markets, the dispersion of betas is much lower so that low-volatility stocks do not underperform much, if at all, whereas during crises, investors seek or keep low-volatility stocks and the beta dispersion increases. As a result, lower volatility assets and portfolios do better over the long term.

How to measure volatility and size

Metrics used to identify low-volatility stocks cover a broad spectrum, with realized volatility (standard deviation) on one end and forecast (implied) volatility and correlations on the other end. Some operationalize low volatility as low beta. The evidence in favor of the volatility anomaly appears robust for different metrics (Ang, 2014).

Quality factors for quantitative investing

Quality factors aim to capture the excess returns reaped by companies that are highly profitable, operationally efficient, safe, stable, and well-governed—in short, high quality. The markets also appear to reward relative earnings certainty and penalize stocks with high earnings volatility.

A portfolio tilt toward businesses with high quality has been long advocated by stock pickers that rely on fundamental analysis, but it is a relatively new phenomenon in quantitative investments. The main challenge is how to define the quality factor consistently and objectively using quantitative indicators, given the subjective nature of quality.

Strategies based on standalone quality factors tend to perform in a counter-cyclical way as investors pay a premium to minimize downside risks and drive up valuations. For this reason, quality factors are often combined with other risk factors in a multi-factor strategy, most frequently with value to produce the quality at a reasonable price strategy.

Long-short quality factors tend to have negative market beta because they are long quality stocks that are also low volatility, and short more volatile, low-quality stocks. Hence, quality factors are often positively correlated with low volatility and momentum factors, and negatively correlated with value and broad market exposure.

Why quality matters

Quality factors may signal outperformance because superior fundamentals such as sustained profitability, steady growth in cash flow, prudent leveraging, a low need for capital market financing, or low financial risk underpin the demand for equity shares and support the price of such companies in the long run. From a corporate finance perspective, a quality company often manages its capital carefully and reduces the risk of over-leveraging or over-capitalization.

A behavioral explanation suggests that investors under-react to information about quality, similar to the rationale for momentum, where investors chase winners and sell losers.

Another argument for quality premia is a herding argument, similar to growth stocks. Fund managers may find it easier to justify buying a company with strong fundamentals, even when it is getting expensive, rather than a more volatile (risky) value stock.

How to measure asset quality

Quality factors rely on metrics computed from the balance sheet and income statement, which indicate profitability reflected in high profit or cash flow margins, operating efficiency, financial strength, and competitiveness more broadly because it implies the ability to sustain a profitability position over time.

Hence, quality has been measured using gross profitability (which has been recently added to the Fama–French factor model; see *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*), return on invested capital, low earnings volatility, or a combination of various prof-

itability, earnings quality, and leverage metrics, with some options listed in the following table.

Earnings management is mainly exercised by manipulating accruals. Hence, the size of accruals is often used as a proxy for earnings quality: higher total accruals relative to assets make low earnings quality more likely. However, this is not unambiguous as accruals can reflect earnings manipulation just as well as accounting estimates of future business growth:

Factor	Description
Asset turnover	This factor measures how efficiently a company uses its assets, which require capital, to produce revenue and is calculated by dividing sales by total assets. A higher turnover is better.
Asset turnover 12-month change	This factor measures a change in management's efficiency in using assets to produce revenue over the last year. Stocks with the highest level of efficiency improvements are typically expected to outperform.
Current ratio	The current ratio is a liquidity metric that measures a company's ability to pay short-term obligations. It compares a company's current assets to its current liabilities, and a higher current ratio is better from a quality perspective.
Interest coverage	This factor measures how easily a company will be able to pay interest on its debt. It is calculated by dividing a company's **earnings before interest and taxes** (**EBIT**) by its interest expense. A higher ratio is desirable.
Leverage	A firm with significantly more debt than equity is considered to be highly leveraged. The debt-to-equity ratio is typically inversely related to prospects, with lower leverage being better.
Payout ratio	The share of earnings paid out in dividends to shareholders. Stocks with higher payout ratios are ranked higher.
Return on equity (ROE)	ROE is computed as the ratio of net income to shareholders' equity. Equities with higher historical returns on equity are ranked higher.

Equipped with a high-level categorization of alpha factors that have been shown to be associated with abnormal returns to varying degrees, we'll now start developing our own financial features from market, fundamental, and alternative data.

Engineering alpha factors that predict returns

Based on a conceptual understanding of key factor categories, their rationale, and popular metrics, a key task is to identify new factors that may better capture the risks embodied by the return drivers laid out previously, or to find new ones. In either case, it will be important to compare the performance of innovative factors to that of known factors to identify incremental signal gains.

Key tools that facilitate the transformation of data into factors include the Python libraries for numerical computing, NumPy and pandas, as well as the Python wrapper around the specialized library for technical analysis, TA-Lib. Alternatives include the expression alphas developed in Zura Kakushadze's 2016 paper, *101 Formulaic Alphas*, and implemented by the alphatools library. In addition, the Quantopian platform provides a large number of built-in factors to speed up the research process.

To apply one or more factors to an investment universe, we can use the Zipline backtesting library (which also includes some built-in factors) and evaluate their performance using the Alphalens library using metrics discussed in the following section.

How to engineer factors using pandas and NumPy

NumPy and pandas are the key tools for custom factor computations. This section demonstrates how they can be used to quickly compute the transformations that yield various alpha factors. If you are not familiar with these libraries, in particular pandas, which we will use throughout this book, please see the `README` for this chapter in the GitHub repo for links to documentation and tutorials.

The notebook `feature_engineering.ipynb` in the `alpha_factors_in_practice` directory contains examples of how to create various factors. The notebook uses data generated by the `create_data.ipynb` notebook in the `data` folder in the root directory of the GitHub repo, which is stored in HDF5 format for faster access. See the notebook `storage_benchmarks.ipynb` in the directory for *Chapter 2*, in the GitHub repo for a comparison of parquet, HDF5, and CSV storage formats for pandas DataFrames.

The NumPy library for scientific computing was created by Travis Oliphant in 2005 by integrating the older Numeric and Numarray libraries that had been developed since the mid-1990s. It is organized in a high-performance *n*-dimensional array data structure called `ndarray`, which enables functionality comparable to MATLAB.

The pandas library emerged in 2008 when Wes McKinney was working at AQR Capital Management. It provides the DataFrame data structure, which is based on NumPy's `ndarray`, but allows for more user-friendly data manipulation with label-based indexing. It includes a wide array of computational tools particularly well-suited to financial data, including

rich time-series operations with automatic date alignment, which we will explore here.

The following sections illustrate some steps in transforming raw stock price data into selected factors. See the notebook `feature_engineering.ipynb` for additional detail and visualizations that we have omitted here to save some space. See the resources listed in the `README` for this chapter on GitHub for links to the documentation and tutorials on how to use pandas and NumPy.

Loading, slicing, and reshaping the data

After loading the Quandl Wiki stock price data on US equities, we select the 2000-18 time slice by applying `pd.IndexSlice` to `pd.MultiIndex`, which contains timestamp and ticker information. We then select and unpivot the adjusted close price column using the `.stack()` method to convert the DataFrame into wide format, with tickers in the columns and timestamps in the rows:

```
idx = pd.IndexSlice
with pd.HDFStore('../../data/assets.h5') as store:
    prices = (store['quandl/wiki/prices']
              .loc[idx['2000':'2018', :], 'adj_close']
              .unstack('ticker'))
prices.info()
DatetimeIndex: 4706 entries, 2000-01-03 to 2018-03-27
Columns: 3199 entries, A to ZUMZ
```

Resampling – from daily to monthly frequency

To reduce training time and experiment with strategies for longer time horizons, we convert the business-daily data into month-end frequency using the available adjusted close price:

```
monthly_prices = prices.resample('M').last()
```

How to compute returns for multiple historical periods

To capture time-series dynamics like momentum patterns, we compute historical multi-period returns using the `pct_change(n_periods)` method, where `n_periods` identifies the number of lags. We then convert the wide result back into long format using `.stack()`, use `.pipe()` to apply the `.clip()` method to the resulting DataFrame, and winsorize returns at the [1%, 99%] levels; that is, we cap outliers at these percentiles.

Finally, we normalize returns using the geometric average. After using `.swaplevel()` to change the order of the `MultiIndex` levels, we obtain the compounded monthly returns over six different periods, ranging from 1 to 12 months:

```
outlier_cutoff = 0.01
data = pd.DataFrame()
lags = [1, 2, 3, 6, 9, 12]
```

```
for lag in lags:
    data[f'return_{lag}m'] = (monthly_prices
                              .pct_change(lag)
                              .stack()
                              .pipe(lambda x:
                                    x.clip(lower=x.quantile(outlier_cutoff),
                                           upper=x.quantile(1-outlier_cutoff)))
                              .add(1)
                              .pow(1/lag)
                              .sub(1)
                              )
data = data.swaplevel().dropna()
data.info()
MultiIndex: 521806 entries, (A, 2001-01-31 00:00:00) to (ZUMZ, 2018-03-
                             31 00:00:00)
Data columns (total 6 columns):
return_1m 521806 non-null float64
return_2m 521806 non-null float64
return_3m 521806 non-null float64
return_6m 521806 non-null float64
return_9m 521806 non-null float64
return_12m 521806 non-null float6
```

We can use these results to compute momentum factors based on the difference between returns over longer periods and the most recent monthly return, as well as for the difference between 3- and 12-month returns, as follows:

```
for lag in [2,3,6,9,12]:
    data[f'momentum_{lag}'] = data[f'return_{lag}m'].sub(data.return_1m)
data[f'momentum_3_12'] = data[f'return_12m'].sub(data.return_3m)
```

Using lagged returns and different holding periods

To use lagged values as input variables or features associated with the current observations, we use the `.shift()` method to move historical returns up to the current period:

```
for t in range(1, 7):
    data[f'return_1m_t-{t}'] = data.groupby(level='ticker').return_1m.shift(t)
```

Similarly, to compute returns for various holding periods, we use the normalized period returns computed previously and shift them back to align them with the current financial features:

```
for t in [1,2,3,6,12]:
    data[f'target_{t}m'] = (data.groupby(level='ticker')
                            [f'return_{t}m'].shift(-t))
```

The notebook also demonstrates how to compute various descriptive statistics for the different return series and visualize their correlation using the seaborn library.

Computing factor betas

We will introduce the Fama–French data to estimate the exposure of assets to common risk factors using linear regression in *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*. The five Fama–French factors, namely market risk, size, value, operating profitability, and investment, have been shown empirically to explain asset returns. They are commonly used to assess the exposure of a portfolio to well-known drivers of risk and returns, where the unexplained portion is then attributed to the manager's idiosyncratic skill. Hence, it is natural to include past factor exposures as financial features in models that aim to predict future returns.

We can access the historical factor returns using the pandas-datareader and estimate historical exposures using the `PandasRollingOLS` rolling linear regression functionality in the pyfinance library, as follows:

```
factors = ['Mkt-RF', 'SMB', 'HML', 'RMW', 'CMA']
factor_data = web.DataReader('F-F_Research_Data_5_Factors_2x3',
              'famafrench', start='2000')[0].drop('RF', axis=1)
factor_data.index = factor_data.index.to_timestamp()
factor_data = factor_data.resample('M').last().div(100)
factor_data.index.name = 'date'
factor_data = factor_data.join(data['return_1m']).sort_index()
T = 24
betas = (factor_data
         .groupby(level='ticker', group_keys=False)
         .apply(lambda x: PandasRollingOLS(window=min(T, x.shape[0]-1), y=x.return_1m, x=x.drop('r
```

As mentioned previously, we will explore both the Fama–French factor model and linear regression in *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*, in more detail. See the notebook `feature_engineering.ipynb` for additional examples, including the computation of lagged and forward returns.

How to add momentum factors

We can use the 1-month and 3-month results to compute simple momentum factors. The following code example shows how to compute the difference between returns over longer periods and the most recent monthly return, as well as for the difference between 3- and 12-month returns:

```
for lag in [2,3,6,9,12]:
    data[f'momentum_{lag}'] = data[f'return_{lag}m'].sub(data.return_1m)
data[f'momentum_3_12'] = data[f'return_12m'].sub(data.return_3m)
```

Adding time indicators to capture seasonal effects

Basic factors also include seasonal anomalies like the January effect, which has been observed to cause higher returns for stocks during this month, possibly for tax reasons. This and other seasonal effects can be modeled through indicator variables that represent specific time periods such as the year and/or the month. These can be generated as follows:

```
dates = data.index.get_level_values('date')
data['year'] = dates.year
data['month'] = dates.month
```

How to create lagged return features

If you want to use lagged returns, that is, returns from previous periods as input variables or features to train a model that learns return patterns to predict future returns, you can use the `.shift()` method to move historical returns up to the current period. The following example moves the returns for the periods 1 to 6 months ago up by the corresponding lag so that they are associated with the observation for the current month:

```
for t in range(1, 7):
    data[f'return_1m_t-{t}'] = data.groupby(level='ticker').return_1m.shift(t)
```

How to create forward returns

Similarly, you can create forward returns for the current period, that is, returns that will occur in the future, using `.shift()` with a negative period (assuming your data is sorted in ascending order):

```
for t in [1,2,3,6,12]:
    data[f'target_{t}m'] = (data.groupby(level='ticker')
                            [f'return_{t}m'].shift(-t))
```

We will use forward returns when we train ML models starting in *Chapter 6*, *The Machine Learning Process*.

How to use TA-Lib to create technical alpha factors

TA-Lib is an open source library written in C++ with a Python interface that is widely used by trading software developers. It contains standardized implementations of over 200 popular indicators for technical analysis; that is, these indicators only use market data, namely price and volume information.

TA-Lib is compatible with pandas and NumPy, rendering its usage very straightforward. The following examples demonstrate how to compute two popular indicators.

Bollinger Bands consist of a **simple moving average** (**SMA**) surrounded by bands two rolling standard deviations below and above the SMA. It was introduced for the visualization of potential overbought/oversold conditions when the price dipped outside the two bands on the upper or lower side, respectively. The inventor, John Bollinger, actually recommended a trading system of 22 rules that generate trade signals.

We can compute the Bollinger Bands and, for comparison, the **relative strength index** described earlier in this section on popular alpha factors

as follows.

We load the adjusted close for a single stock—in this case, AAPL:

```
with pd.HDFStore(DATA_STORE) as store:
    data = (store['quandl/wiki/prices']
            .loc[idx['2007':'2010', 'AAPL'],
                 ['adj_open', 'adj_high', 'adj_low', 'adj_close',
                  'adj_volume']]
            .unstack('ticker')
            .swaplevel(axis=1)
            .loc[:, 'AAPL']
            .rename(columns=lambda x: x.replace('adj_', '')))
```

Then, we pass the one-dimensional `pd.Series` through the relevant TA-Lib functions:

```
from talib import RSI, BBANDS
up, mid, low = BBANDS(data.close, timeperiod=21, nbdevup=2, nbdevdn=2,
                      matype=0)
rsi = RSI(adj_close, timeperiod=14)
```

Then, we collect the results in a DataFrame and plot the Bollinger Bands with the AAPL stock price and the RSI with the 30/70 lines, which suggest long/short opportunities:

```
data = pd.DataFrame({'AAPL': data.close, 'BB Up': up, 'BB Mid': mid,
                     'BB down': low, 'RSI': rsi})
fig, axes= plt.subplots(nrows=2, figsize=(15, 8))
data.drop('RSI', axis=1).plot(ax=axes[0], lw=1, title='Bollinger Bands')
data['RSI'].plot(ax=axes[1], lw=1, title='Relative Strength Index')
axes[1].axhline(70, lw=1, ls='--', c='k')
axes[1].axhline(30, lw=1, ls='--', c='k')
```

The result, shown in *Figure 4.4*, is rather mixed—both indicators suggested overbought conditions during the early post-crisis recovery when the price continued to rise:

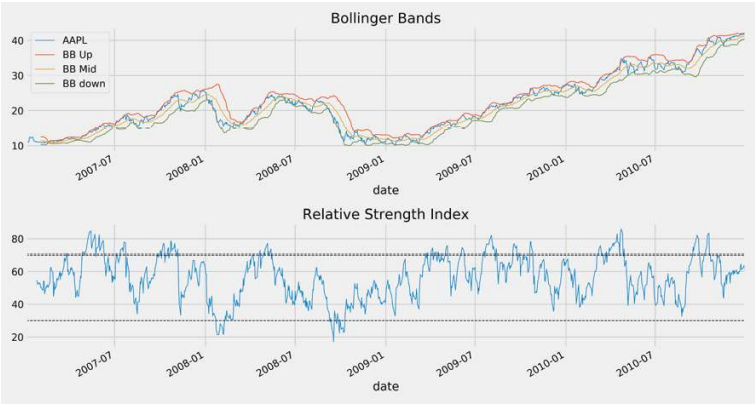

Figure 4.4: Bollinger Bands and relative strength index

Denoising alpha factors with the Kalman filter

The concept of **noise in data** relates to the domain of signal processing, which aims to retrieve the correct information from a signal sent, for example, through the air in the form of electromagnetic waves. As the waves move through space, environmental interference can be added to the originally pure signal in the form of noise, making it necessary to separate the two once received.

The Kalman filter was introduced in 1960 and has become very popular for many applications that require processing noisy data because it permits more accurate estimates of the underlying signal.

This technique is widely used to track objects in computer vision, to support the localization and navigation of aircraft and spaceships, and to control robotic motion based on noisy sensor data, besides its use in time series analysis.

Noise is used similarly in data science, finance, and other domains, implying that the raw data contains useful information, for instance, in terms of trading signals, that needs to be extracted and separated from irrelevant, extraneous information. Clearly, the fact that we do not know the true signal can make this separation rather challenging at times.

We will first review how the Kalman filter works and which assumptions it makes to achieve its objectives. Then, we will demonstrate how to apply it to financial data using the pykalman library.

How does the Kalman filter work?

The Kalman filter is a dynamic linear model of sequential data like a time series that adapts to new information as it arrives. Rather than using a fixed-size window like a moving average or a given set of weights like an exponential moving average, it incorporates new data into its estimates of the current value of the time series based on a probabilistic model.

More specifically, the Kalman filter is a probabilistic model of a sequence of observations z_1, z_2, ..., z_T and a corresponding sequence of hidden states x_1, x_2, ..., x_T (with the notation used by the pykalman library that we will demonstrate here). This can be represented by the following graph:

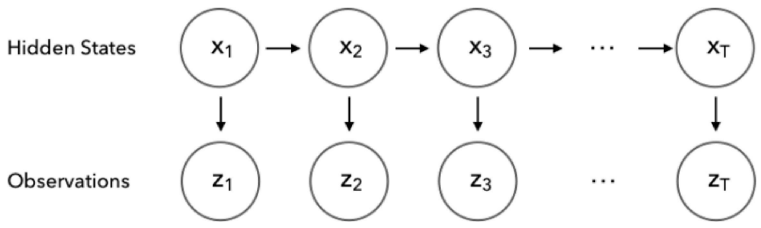

Figure 4.5: Kalman filter as a graphical model

Technically speaking, the Kalman filter takes a Bayesian approach that propagates the posterior distribution of the state variables x given their

measurements z over time (see *Chapter 10*, *Bayesian ML – Dynamic Sharpe Ratios and Pairs Trading*, for more details on Bayesian inference). We can also view it as an unsupervised algorithm for tracking a single object in a continuous state space, where we will take the object to be, for example, the value of or returns on a security, or an alpha factor (see *Chapter 13*, *Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*).

To recover the hidden states from a sequence of observations that may become available in real time, the algorithm iterates between two steps:

1. **Prediction step**: Estimate the current state of the process.
2. **Measurement step**: Use noisy observations to update its estimate by averaging the information from both steps in a way that weighs more certain estimates higher.

The basic idea behind the algorithm is as follows: certain assumptions about a dynamic system and a history of corresponding measurements will allow us to estimate the system's state in a way that maximizes the probability of the previous measurements.

To achieve its objective of recovering the hidden state, the Kalman filter makes the following assumptions:

- The system that we are modeling behaves in a linear fashion.
- The hidden state process is a Markov chain so that the current hidden state x_t depends only on the most recent prior hidden state x_{t-1}.
- Measurements are subject to Gaussian, uncorrelated noise with constant covariance.

As a result, the Kalman filter is similar to a hidden Markov model, except that the state space of the latent variables is continuous, and both hidden and observed variables have normal distributions, denoted as $\mathcal{N}(\mu, \sigma)$ with mean μ and standard

In mathematical terms, the key components of the model (and corresponding parameters in the pykalman implementation) are:

- The initial hidden state has a normal distribution: $x_0 \sim \mathcal{N}(\mu_0, \Sigma_0)$ with `initial_state_mean` , μ and `initial_state_covariance` , Σ.
- The hidden state x_{t+1} is an affine transformation of x_t with `transition_matrix` A, `transition_offset` b, and added Gaussian noise with `transition_covariance` Q:
  $x_{t+1} = A_t x_t + b_t + \epsilon_{t+1}^1, \;\; \epsilon_t^1 \sim \mathcal{N}(0, Q)$.
- The observation z_t is an affine transformation of the hidden state x_t with `observation_matrix` C, `observation_offset` d, and added Gaussian noise with `observation_covariance` R:
  $z_t = C_t x_t + d_t + \epsilon_t^2, \;\; \epsilon_t^2 \sim \mathcal{N}(0, R)$.

Among the advantages of a Kalman filter is that it flexibly adapts to non-stationary data with changing distributional characteristics (see *Chapter 9*, *Time-Series Models for Volatility Forecasts and Statistical Arbitrage*, for more details on stationarity).

Key disadvantages are the assumptions of linearity and Gaussian noise that financial data often violate. To address these shortcomings, the Kalman filter has been extended to systems with nonlinear dynamics in the form of the extended and the unscented Kalman filters. The particle filter is an alternative approach that uses sampling-based Monte Carlo approaches to estimate non-normal distributions.

How to apply a Kalman filter using pykalman

The Kalman filter is particularly useful for rolling estimates of data values or model parameters that change over time. This is because it adapts its estimates at every time step based on new observations and tends to weigh recent observations more heavily.

Except for conventional moving averages, the Kalman filter does not require us to specify the length of a window used for the estimate. Rather, we start out with our estimate of the mean and covariance of the hidden state and let the Kalman filter correct our estimates based on periodic observations. The code examples for this section are in the notebook `kalman_filter_and_wavelets.ipynb`.

The following code example shows how to apply the Kalman filter to smoothen the S&P 500 stock price series for the 2008-09 period:

```
with pd.HDFStore(DATA_STORE) as store:
    sp500 = store['sp500/stooq'].loc['2008': '2009', 'close']
```

We initialize the `KalmanFilter` with unit covariance matrices and zero means (see the pykalman documentation for advice on dealing with the challenges of choosing appropriate initial values):

```
from pykalman import KalmanFilter
kf = KalmanFilter(transition_matrices = [1],
                  observation_matrices = [1],
                  initial_state_mean = 0,
                  initial_state_covariance = 1,
                  observation_covariance=1,
                  transition_covariance=.01)
```

Then, we run the `filter` method to trigger the forward algorithm, which iteratively estimates the hidden state, that is, the mean of the time series:

```
state_means, _ = kf.filter(sp500)
```

Finally, we add moving averages for comparison and plot the result:

```
sp500_smoothed = sp500.to_frame('close')
sp500_smoothed['Kalman Filter'] = state_means
for months in [1, 2, 3]:
    sp500_smoothed[f'MA ({months}m)'] = (sp500.rolling(window=months * 21)
                                         .mean())
```

```
ax = sp500_smoothed.plot(title='Kalman Filter vs Moving Average',
                         figsize=(14, 6), lw=1, rot=0)
```

The resulting plot in *Figure 4.6* shows that the Kalman filter performs similarly to a 1-month moving average but is more sensitive to changes in the behavior of the time series:

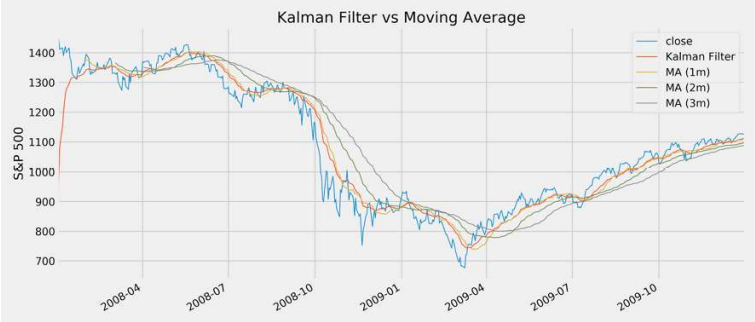

Figure 4.6: Kalman filter versus moving average

How to preprocess your noisy signals using wavelets

Wavelets are related to Fourier analysis, which combines sine and cosine waves at different frequencies to approximate noisy signals. While Fourier analysis is particularly useful to translate signals from the time to the frequency domain, wavelets are useful for filtering out specific patterns that may occur at different scales, which, in turn, may correspond to a frequency range.

Wavelets are functions or wave-like patterns that decompose a discrete or continuous-time signal into components of different scales. A wavelet transform, in turn, represents a function using wavelets as scaled and translated copies of a finite-length waveform. This transform has advantages over Fourier transforms for functions with discontinuities and sharp peaks, and to approximate non-periodic or non-stationary signals.

To denoise a signal, you can use wavelet shrinkage and thresholding methods. First, you choose a specific wavelet pattern to decompose a dataset. The wavelet transform yields coefficients that correspond to details in the dataset.

The idea of thresholding is simply to omit all coefficients below a particular cutoff, assuming that they represent minor details that are not necessary to represent the true signal. These remaining coefficients are then used in an inverse wavelet transformation to reconstruct the (denoised) dataset.

We'll now use the pywavelets library to apply wavelets to noisy stock data. The following code example illustrates how to denoise the S&P 500 returns using a forward and inverse wavelet transform with a Daubechies 6 wavelet and different threshold values.

First, we generate daily S&P 500 returns for the 2008-09 period:

```
signal = (pd.read_hdf(DATA_STORE, 'sp500/stooq')
          .loc['2008': '2009']
          .close.pct_change()
          .dropna())
```

Then, we select one of the Daubechies wavelets from the numerous built-in wavelet functions:

```
import pywt
pywt.families(short=False)
['Haar', 'Daubechies',  'Symlets',  'Coiflets',  'Biorthogonal',  'Reverse biorthogonal',  'Discre
```

The Daubechies 6 wavelet is defined by a scaling function ψ and the wavelet function φ itself (see the PyWavelet documentation for details and the accompanying notebook `kalman_filter_and_wavelets.ipynb` for plots of all built-in wavelet functions):

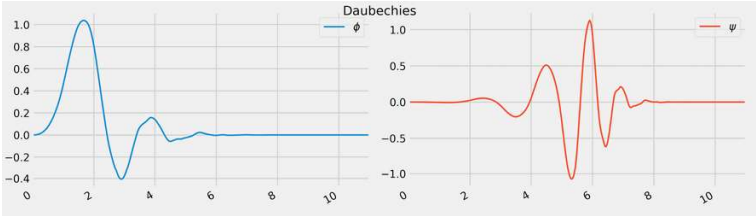

Figure 4.7: Daubechies wavelets

Given a wavelet function, we first decompose the return signal using the `.wavedec` function, which yields the coefficients for the wavelet transform. Next, we filter out all coefficients above a given threshold and then reconstruct the signal using only those coefficients using the inverse transform `.waverec`:

```
wavelet = "db6"
for i, scale in enumerate([.1, .5]):

    coefficients = pywt.wavedec(signal, wavelet, mode='per')
    coefficients[1:] = [pywt.threshold(i, value=scale*signal.max(), mode='soft') for i in coeffici
    reconstructed_signal = pywt.waverec(coefficients, wavelet, mode='per')
    signal.plot(color="b", alpha=0.5, label='original signal', lw=2,
                title=f'Threshold Scale: {scale:.1f}', ax=axes[i])
    pd.Series(reconstructed_signal, index=signal.index).plot(c='k', label='DWT smoothing}', linewi
```

The notebook shows how to apply this denoising technique with different thresholds, and the resulting plot, shown in *Figure 4.8*, clearly shows how a higher threshold value yields a significantly smoother series:

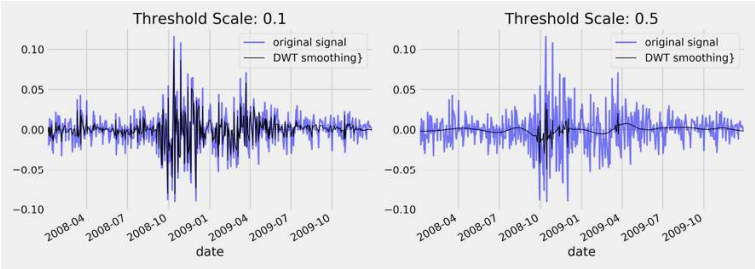

Figure 4.8: Wavelet denoising with different thresholds

From signals to trades – Zipline for backtests

The open source library Zipline is an event-driven backtesting system. It generates market events to simulate the reactions of an algorithmic trading strategy and tracks its performance. A particularly important feature is that it provides the algorithm with historical point-in-time data that avoids look-ahead bias.

The library has been popularized by the crowd-sourced quantitative investment fund Quantopian, which uses it in production to facilitate algorithm development and live-trading.

In this section, we'll provide a brief demonstration of its basic functionality. *Chapter 8*, *The ML4T Workflow – From Model to Strategy Backtesting*, contains a more detailed introduction to prepare us for more complex use cases.

How to backtest a single-factor strategy

You can use Zipline offline in conjunction with data bundles to research and evaluate alpha factors. When using it on the Quantopian platform, you will get access to a wider set of fundamental and alternative data. We will also demonstrate the Quantopian research environment in this chapter, and the backtesting IDE in the next chapter. The code for this section is in the `01_factor_research_evaluation` sub-directory of the GitHub repo folder for this chapter, including installation instructions and an environment tailored to Zipline's dependencies.

For installation, please see the instructions in this chapter's `README` on GitHub. After installation and before executing the first algorithm, you need to ingest a data bundle that, by default, consists of Quandl's community-maintained data on stock prices, dividends, and splits for 3,000 US publicly traded companies.

You need a Quandl API key to run the following code, which stores the data in your `home` folder under `~/.zipline/data/<bundle>` :

```
$ QUANDL_API_KEY=<yourkey> zipline ingest [-b <bundle>]
```

A single alpha factor from market data

We are first going to illustrate the Zipline alpha factor research workflow in an offline environment. In particular, we will develop and test a simple mean-reversion factor that measures how much recent performance has deviated from the historical average.

Short-term reversal is a common strategy that takes advantage of the weakly predictive pattern that stock prices are likely to revert back to a rolling mean over horizons from less than 1 minute to 1 month. See the notebook `single_factor_zipline.ipynb` for details.

To this end, the factor computes the *z*-score for the last monthly return relative to the rolling monthly returns over the last year. At this point, we will not place any orders to simply illustrate the implementation of a `CustomFactor` and record the results during the simulation.

Zipline includes numerous built-in factors for many common operations (see the *Quantopian* documentation linked on GitHub for details). While this is often convenient and sufficient, in other cases, we want to transform our available data differently. For this purpose, Zipline provides the `CustomFactor` class, which offers a lot of flexibility for us to specify a wide range of calculations. It does this using the various features available for the cross-section of securities and custom lookback periods using NumPy.

To this end, after some basic settings, `MeanReversion` subclasses `CustomFactor` and defines a `compute()` method. It creates default inputs of monthly returns over an also default year-long window so that the `monthly_return` variable will have 252 rows and one column for each security in the Quandl dataset on a given day.

The `compute_factors()` method creates a `MeanReversion` factor instance and creates long, short, and ranking pipeline columns. The former two contain Boolean values that can be used to place orders, and the latter reflects that overall ranking to evaluate the overall factor performance. Furthermore, it uses the built-in `AverageDollarVolume` factor to limit the computation to more liquid stocks:

```
from zipline.api import attach_pipeline, pipeline_output, record
from zipline.pipeline import Pipeline, CustomFactor
from zipline.pipeline.factors import Returns, AverageDollarVolume
from zipline import run_algorithm
MONTH, YEAR = 21, 252
N_LONGS = N_SHORTS = 25
VOL_SCREEN = 1000
class MeanReversion(CustomFactor):
    """Compute ratio of latest monthly return to 12m average,
       normalized by std dev of monthly returns"""
    inputs = [Returns(window_length=MONTH)]
    window_length = YEAR
    def compute(self, today, assets, out, monthly_returns):
        df = pd.DataFrame(monthly_returns)
        out[:] = df.iloc[-1].sub(df.mean()).div(df.std())
def compute_factors():
```

```
    """Create factor pipeline incl. mean reversion,
        filtered by 30d Dollar Volume; capture factor ranks"""
    mean_reversion = MeanReversion()
    dollar_volume = AverageDollarVolume(window_length=30)
    return Pipeline(columns={'longs'  : mean_reversion.bottom(N_LONGS),
                             'shorts' : mean_reversion.top(N_SHORTS),
                             'ranking':
                            mean_reversion.rank(ascending=False)},
                            screen=dollar_volume.top(VOL_SCREEN))
```

The result will allow us to place long and short orders. In the next chapter, we will learn how to build a portfolio by choosing a rebalancing period and adjusting portfolio holdings as new signals arrive.

The `initialize()` method registers the `compute_factors()` pipeline, and the `before_trading_start()` method ensures the pipeline runs on a daily basis. The `record()` function adds the pipeline's ranking column, as well as the current asset prices, to the performance DataFrame returned by the `run_algorithm()` function:

```
def initialize(context):
    """Setup: register pipeline, schedule rebalancing,
        and set trading params"""
    attach_pipeline(compute_factors(), 'factor_pipeline')
def before_trading_start(context, data):
    """Run factor pipeline"""
    context.factor_data = pipeline_output('factor_pipeline')
    record(factor_data=context.factor_data.ranking)
    assets = context.factor_data.index
    record(prices=data.current(assets, 'price'))
```

Finally, define the `start` and `end` Timestamp objects in UTC terms, set a capital base, and execute `run_algorithm()` with references to the key execution methods. The performance DataFrame contains nested data, for example, the prices column consists of a `pd.Series` for each cell. Hence, subsequent data access is easier when stored in pickle format:

```
start, end = pd.Timestamp('2015-01-01', tz='UTC'), pd.Timestamp('2018-
             01-01', tz='UTC')
capital_base = 1e7
performance = run_algorithm(start=start,
                            end=end,
                            initialize=initialize,
                            before_trading_start=before_trading_start,
                            capital_base=capital_base)
performance.to_pickle('single_factor.pickle')
```

We will use the factor and pricing data stored in the `performance` DataFrame to evaluate the factor performance for various holding periods in the next section, but first, we'll take a look at how to create more complex signals by combining several alpha factors from a diverse set of data sources on the Quantopian platform.

Built-in Quantopian factors

The accompanying notebook `factor_library_quantopian.ipynb` contains numerous example factors that are either provided by the Quantopian platform or computed from data sources available using the research API from a Jupyter Notebook.

There are built-in factors that can be used in combination with quantitative Python libraries—in particular, NumPy and pandas—to derive more complex factors from a broad range of relevant data sources such as US equity prices, Morningstar fundamentals, and investor sentiment.

For instance, the price-to-sales ratio is available as part of the Morningstar fundamentals dataset. It can be used as part of a pipeline that will be further described as we introduce the Zipline library.

Combining factors from diverse data sources

The Quantopian research environment is tailored to the rapid testing of predictive alpha factors. The process is very similar because it builds on Zipline but offers much richer access to data sources. The following code sample illustrates how to compute alpha factors not only from market data, as done previously, but also from fundamental and alternative data. See the notebook `multiple_factors_quantopian_research.ipynb` for details.

Quantopian provides several hundred Morningstar fundamental variables for free and also includes Stocktwits signals as an example of an alternative data source. There are also custom universe definitions such as `QTradableStocksUS`, which applies several filters to limit the backtest universe to stocks that were likely tradeable under realistic market conditions:

```
from quantopian.research import run_pipeline
from quantopian.pipeline import Pipeline
from quantopian.pipeline.data.builtin import USEquityPricing
from quantopian.pipeline.data.morningstar import income_statement,
     operation_ratios, balance_sheet
from quantopian.pipeline.data.psychsignal import stocktwits
from quantopian.pipeline.factors import CustomFactor,
     SimpleMovingAverage, Returns
from quantopian.pipeline.filters import QTradableStocksUS
```

We will use a custom `AggregateFundamentals` class to use the last reported fundamental data point. This aims to address the fact that fundamentals are reported quarterly, and Quantopian does not currently provide an easy way to aggregate historical data, say, to obtain the sum of the last four quarters, on a rolling basis:

```
class AggregateFundamentals(CustomFactor):
    def compute(self, today, assets, out, inputs):
        out[:] = inputs[0]
```

We will again use the custom `MeanReversion` factor from the preceding code. We will also compute several other factors for the given universe

definition using the `rank()` method's `mask` parameter:

```
def compute_factors():
    universe = QTradableStocksUS()
    profitability = (AggregateFundamentals(inputs=
                     [income_statement.gross_profit],
                                        window_length=YEAR) /
                     balance_sheet.total_assets.latest).rank(mask=universe)
    roic = operation_ratios.roic.latest.rank(mask=universe)
    ebitda_yield = (AggregateFundamentals(inputs=
                              [income_statement.ebitda],
                                           window_length=YEAR) /
                    USEquityPricing.close.latest).rank(mask=universe)
    mean_reversion = MeanReversion().rank(mask=universe)
    price_momentum = Returns(window_length=QTR).rank(mask=universe)
    sentiment = SimpleMovingAverage(inputs=[stocktwits.bull_minus_bear],
                                    window_length=5).rank(mask=universe)
    factor = profitability + roic + ebitda_yield + mean_reversion +
             price_momentum + sentiment
    return Pipeline(
            columns={'Profitability'      : profitability,
                     'ROIC'               : roic,
                     'EBITDA Yield'       : ebitda_yield,
                     "Mean Reversion (1M)": mean_reversion,
                     'Sentiment'          : sentiment,
                     "Price Momentum (3M)": price_momentum,
                     'Alpha Factor'       : factor})
```

This algorithm simply averages how the six individual factors rank each asset to combine their information. This is a fairly naive method that does not account for the relative importance and incremental information each factor may provide when predicting future returns. The ML algorithms of the following chapters will allow us to do exactly this, using the same backtesting framework.

Execution also relies on `run_algorithm()`, but the `return` DataFrame on the Quantopian platform only contains the factor values created by the `Pipeline`. This is convenient because this data format can be used as input for Alphalens, the library that's used for the evaluation of the predictive performance of alpha factors.

> **Using TA-Lib with Zipline**
>
> The TA-Lib library includes numerous technical factors. A Python implementation is available for local use, for example, with Zipline and Alphalens, and it is also available on the Quantopian platform. The notebook also illustrates several technical indicators available using TA-Lib.

Separating signal from noise with Alphalens

Quantopian has open sourced the Python Alphalens library for the performance analysis of predictive stock factors. It integrates well with the

Zipline backtesting library and the portfolio performance and risk analysis library pyfolio, which we will explore in the next chapter.

Alphalens facilitates the analysis of the predictive power of alpha factors concerning the:

- Correlation of the signals with subsequent returns
- Profitability of an equal or factor-weighted portfolio based on a (subset of) the signals
- Turnover of factors to indicate the potential trading costs
- Factor performance during specific events
- Breakdowns of the preceding by sector

The analysis can be conducted using tearsheets or individual computations and plots. The tearsheets are illustrated in the online repository to save some space.

Creating forward returns and factor quantiles

To utilize Alphalens, we need to provide two inputs:

- Signals for a universe of assets, like those returned by the ranks of the `MeanReversion` factor
- The forward returns that we would earn by investing in an asset for a given holding period

See the notebook `06_performance_eval_alphalens.ipynb` for details.

We will recover the prices from the `single_factor.pickle` file as follows (and proceed in the same way for `factor_data`; see the notebook):

```
performance = pd.read_pickle('single_factor.pickle')
prices = pd.concat([df.to_frame(d) for d, df in performance.prices.items()],axis=1).T
prices.columns = [re.findall(r"\[(.+)\]", str(col))[0] for col in
                  prices.columns]
prices.index = prices.index.normalize()
prices.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 755 entries, 2015-01-02 to 2017-12-29
Columns: 1661 entries, A to ZTS
dtypes: float64(1661)
```

We can generate the Alphalens input data, namely the factor signal and forward returns described previously, in the required format from the Zipline output using the `get_clean_factor_and_forward_returns` utility function. This function returns the signal quintiles and the forward returns for the given holding periods:

```
HOLDING_PERIODS = (5, 10, 21, 42)
QUANTILES = 5
alphalens_data = get_clean_factor_and_forward_returns(factor=factor_data,
                                      prices=prices,
                                      periods=HOLDING_PERIODS,
```

```
                                    quantiles=QUANTILES)
Dropped 14.5% entries from factor data: 14.5% in forward returns computation and 0.0% in binning p
```

The `alphalens_data` DataFrame contains the returns on an investment in the given asset on a given date for the indicated holding period, as well as the factor value—that is, the asset's `MeanReversion` ranking on that date and the corresponding quantile value:

date	asset	5D	10D	21D	42D	fac-tor	factor_quantile
	A	-1.87%	-1.11%	-4.61%	5.28%	2618	4
	AAL	-0.06%	-8.03%	-9.63%	-10.39%	1088	2
1/2/2015	AAP	-1.32%	0.23%	-1.63%	-2.39%	791	1
	AAPL	-2.82%	-0.07%	8.51%	18.07%	2917	5
	ABBV	-1.88%	-0.20%	-7.88%	-8.24%	2952	5

The forward returns and the signal quantiles are the basis for evaluating the predictive power of the signal. Typically, a factor should deliver markedly different returns for distinct quantiles, such as negative returns for the bottom quintile of the factor values and positive returns for the top quantile.

Predictive performance by factor quantiles

As a first step, we would like to visualize the average period return by factor quantile. We can use the built-in function `mean_return_by_quantile` from the performance module and `plot_quantile_returns_bar` from the plotting module:

```
from alphalens.performance import mean_return_by_quantile
from alphalens.plotting import plot_quantile_returns_bar
mean_return_by_q, std_err = mean_return_by_quantile(alphalens_data)
plot_quantile_returns_bar(mean_return_by_q);
```

The result is a bar chart that breaks down the mean of the forward returns for the four different holding periods based on the quintile of the factor signal.

As you can see in *Figure 4.9*, the bottom quintiles yielded markedly more negative results than the top quintiles, except for the longest holding period:

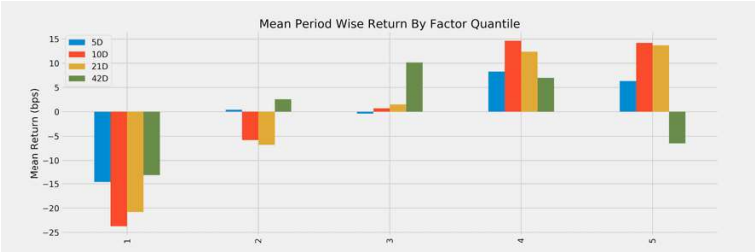

Figure 4.9: Mean period return by factor quantile

The **10D** holding period provides slightly better results for the first and fourth quartiles on average across the trading period.

We would also like to see the performance over time of investments driven by each of the signal quintiles. To this end, we calculate daily as opposed to average returns for the **5D** holding period. Alphalens adjusts the period returns to account for the mismatch between daily signals and a longer holding period (for details, see the Alphalens documentation):

```
from alphalens.plotting import plot_cumulative_returns_by_quantile
mean_return_by_q_daily, std_err =
     mean_return_by_quantile(alphalens_data, by_date=True)
plot_cumulative_returns_by_quantile(mean_return_by_q_daily['5D'],
     period='5D');
```

The resulting line plot in *Figure 4.10* shows that, for most of this 3-year period, the top two quintiles significantly outperformed the bottom two quintiles. However, as suggested by the previous plot, the signals by the fourth quintile produced slightly better performance than those by the top quintile due to their relative performance during 2017:

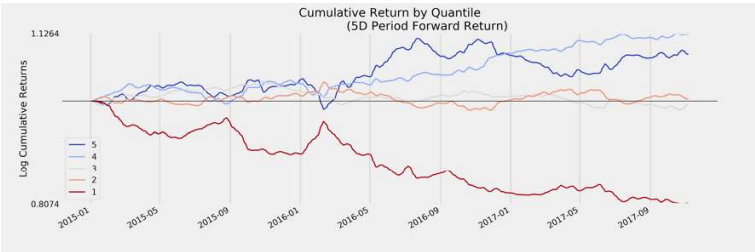

Figure 4.10: Cumulative return by quantile for a 5-day holding period

A factor that is useful for a trading strategy shows the preceding pattern, where cumulative returns develop along clearly distinct paths, because this allows for a long-short strategy with lower capital requirements and, correspondingly, lower exposure to the overall market.

However, we also need to take the dispersion of period returns into account, rather than just the averages. To this end, we can rely on the built-in `plot_quantile_returns_violin`:

```
from alphalens.plotting import plot_quantile_returns_violin
```

```
plot_quantile_returns_violin(mean_return_by_q_daily);
```

This distributional plot, shown in *Figure 4.11*, highlights that the range of daily returns is fairly wide. Despite different means, the separation of the distributions is very limited so that, on any given day, the differences in performance between the different quintiles may be rather limited:

Figure 4.11: Distribution of the period-wise return by factor quintile

While we focus on the evaluation of a single alpha factor, we are simplifying things by ignoring practical issues related to trade execution that we will relax when we address proper backtesting in the next chapter. Some of these include:

- The transaction costs of trading
- Slippage, or the difference between the price at decision and trade execution, for example, due to the market impact

The information coefficient

Most of this book is about the design of alpha factors using ML models. ML is about optimizing some predictive objective, and in this section, we will introduce the key metrics used to measure the performance of an alpha factor. We will define **alpha** as *the average return in excess of a benchmark*.

This leads to the **information ratio** (**IR**), which measures the average excess return per unit of risk taken by dividing alpha by the tracking risk. When the benchmark is the risk-free rate, the IR corresponds to the well-known Sharpe ratio, and we will highlight crucial statistical measurement issues that arise in the typical case when returns are not normally distributed. We will also explain the fundamental law of active management, which breaks the IR down into a combination of forecasting skill and a strategy's ability to effectively leverage these forecasting skills.

The goal of alpha factors is the accurate directional prediction of future returns. Hence, a natural performance measure is the correlation between an alpha factor's predictions and the forward returns of the target assets.

It is better to use the non-parametric Spearman rank correlation coefficient, which measures how well the relationship between two variables can be described using a monotonic function, as opposed to the Pearson correlation, which measures the strength of a linear relationship.

We can obtain the **information coefficient (IC)** using Alphalens, which relies on `scipy.stats.spearmanr` under the hood (see the repo for an example of how to use `scipy` directly to obtain *p*-values). The `factor_information_coefficient` function computes the period-wise correlation and `plot_ic_ts` creates a time-series plot with a 1-month moving average:

```
from alphalens.performance import factor_information_coefficient
from alphalens.plotting import plot_ic_ts
ic = factor_information_coefficient(alphalens_data)
plot_ic_ts(ic[['5D']])
```

The time series plot in *Figure 4.12* shows extended periods with significantly positive moving average IC. An IC of 0.05 or even 0.1 allows for significant outperformance if there are sufficient opportunities to apply this forecasting skill, as the fundamental law of active management will illustrate:

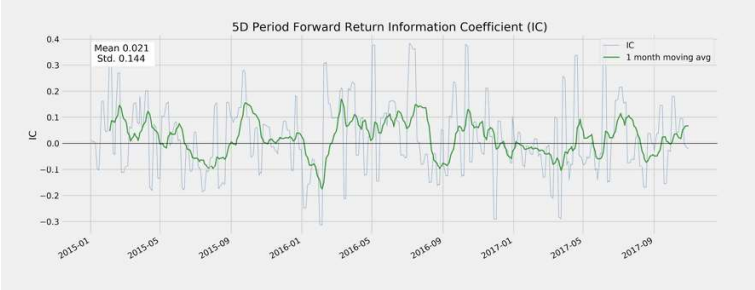

Figure 4.12: Moving average of the IC for 5-day horizon

A plot of the annual mean IC highlights how the factor's performance was historically uneven:

```
ic = factor_information_coefficient(alphalens_data)
ic_by_year = ic.resample('A').mean()
ic_by_year.index = ic_by_year.index.year
ic_by_year.plot.bar(figsize=(14, 6))
```

This produces the chart shown in *Figure 4.13*:

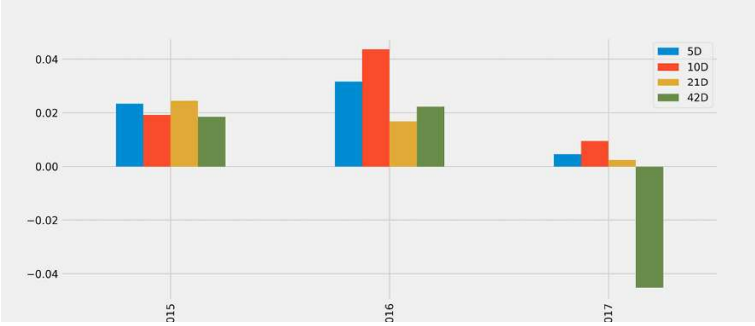

Figure 4.13: IC by year

An information coefficient below 0.05, as in this case, is low but significant and can produce positive residual returns relative to a benchmark, as we will see in the next section. The command `create_summary_tear_sheet(alphalens_data)` creates IC summary statistics.

The risk-adjusted IC results from dividing the mean IC by the standard deviation of the IC, which is also subjected to a two-sided *t*-test with the null hypothesis *IC = 0* using `scipy.stats.ttest_1samp` :

	5D	10D	21D	42D
IC mean	0.021	0.025	0.015	0.001
IC std.	0.144	0.13	0.12	0.12
Risk-adjusted IC	0.145	0.191	0.127	0.01
t-stat (IC)	3.861	5.107	3.396	0.266
p-value (IC)	0	0	0.001	0.79
IC skew	0.384	0.251	0.115	0.134
IC kurtosis	0.019	-0.584	-0.353	-0.494

Factor turnover

Factor turnover measures how frequently the assets associated with a given quantile change, that is, how many trades are required to adjust a portfolio to the sequence of signals. More specifically, it measures the share of assets currently in a factor quantile that was not in that quantile in the last period. The following table is produced by this command:

```
create_turnover_tear_sheet(alphalens_data)
```

The share of assets that were to join a quintile-based portfolio is fairly high, suggesting that the trading costs pose a challenge to reaping the benefits from the predictive performance:

Mean turnover	5D	10D	21D	42D
Quantile 1	0.587	0.826	0.828	0.41
Quantile 2	0.737	0.801	0.81	0.644
Quantile 3	0.764	0.803	0.808	0.679
Quantile 4	0.737	0.803	0.808	0.641

Quantile 5	0.565	0.802	0.809	0.393

An alternative view on factor turnover is the correlation of the asset rank due to the factor over various holding periods, also part of the tear sheet:

	5D	10D	21D	42D
Mean factor rank autocorrelation	0.713	0.454	-0.011	-0.016

Generally, more stability is preferable to keep trading costs manageable.

Alpha factor resources

The research process requires designing and selecting alpha factors with respect to the predictive power of their signals. An algorithmic trading strategy will typically build on multiple alpha factors that send signals for each asset. These factors may be aggregated using an ML model to optimize how the various signals translate into decisions about the timing and sizing of individual positions, as we will see in subsequent chapters.

Alternative algorithmic trading libraries

Additional open source Python libraries for algorithmic trading and data collection include the following (see GitHub for links):

- **QuantConnect** is a competitor to Quantopian.
- **WorldQuant** offers online competition and recruits community contributors to a crowd-sourced hedge fund.
- **Alpha Trading Labs** offers an s high-frequency focused testing infrastructure with a business model similar to Quantopian.
- The **Python Algorithmic Trading Library** (**PyAlgoTrade**) focuses on backtesting and offers support for paper trading and live trading. It allows you to evaluate an idea for a trading strategy with historical data and aims to do so with minimal effort.
- **pybacktest** is a vectorized backtesting framework that uses pandas and aims to be compact, simple, and fast. (The project is currently on hold.)
- **ultrafinance** is an older project that combines real-time financial data collection and the analysis and backtesting of trading strategies.
- **Trading with Python** offers courses and a collection of functions and classes for quantitative trading.
- **Interactive Brokers** offers a Python API for live trading on their platform.

Summary

In this chapter, we introduced a range of alpha factors that have been used by professional investors to design and evaluate strategies for decades. We laid out how they work and illustrated some of the economic

mechanisms believed to drive their performance. We did this because a solid understanding of how factors produce excess returns helps innovate new factors.

We also presented several tools that you can use to generate your own factors from various data sources and demonstrated how the Kalman filter and wavelets allow us to smoothen noisy data in the hope of retrieving a clearer signal.

Finally, we provided a glimpse of the Zipline library for the event-driven simulation of a trading algorithm, both offline and on the Quantopian online platform. You saw how to implement a simple mean reversion factor and how to combine multiple factors in a simple way to drive a basic strategy. We also looked at the Alphalens library, which permits the evaluation of the predictive performance and trading turnover of signals.

The portfolio construction process, in turn, takes a broader perspective and is aims at the optimal sizing of positions from a risk and return perspective. In the next chapter, *Portfolio Optimization and Strategy Evaluation*, we will turn to various strategies to balance risk and returns in a portfolio process. We will also look in more detail at the challenges of backtesting trading strategies on a limited set of historical data, as well as how to address these challenges.