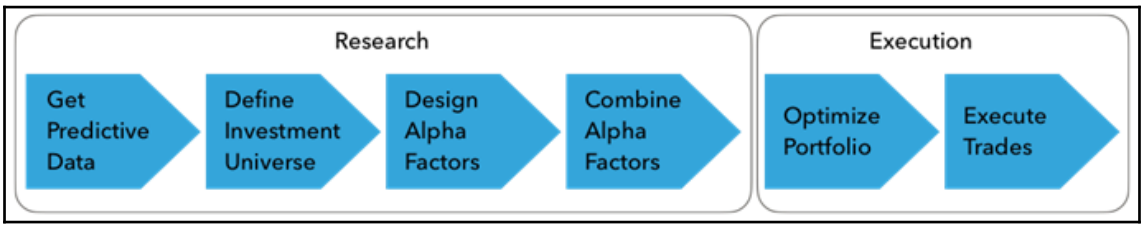
4
Alpha Factor Research

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

Alpha factors aim to predict the price movements of assets in the investment universe based on the available market, fundamental, or alternative data. A factor may combine one or several input variables, but assumes a single value for each asset every time the strategy evaluates the factor. Trade decisions typically rely on relative values across assets. Trading strategies are often based on signals emitted by multiple factors, and we will see that **machine learning** (**ML**) models are particularly well suited to integrate the various signals efficiently to make more accurate predictions.

The design, evaluation, and combination of alpha factors are critical steps during the research phase of the algorithmic trading strategy workflow, as shown in the following diagram. We will focus on the research phase in this `Chapter 4`, *Strategy Evaluation*, and the execution phase in the next chapter. The remainder of this book will then focus on the use of ML to discover and combine alpha factors. Take a look at the following figure:



This chapter will use a simple mean-reversal factor to introduce the algorithmic trading simulator `zipline` that is written in Python and facilitates the testing of alpha factors for a given investment universe. We will also use `zipline` when we backtest trading strategies in a portfolio context in the next chapter. Next, we will discuss key metrics to evaluate the predictive performance of alpha factors, including the information coefficient and the information ratio, which leads to the fundamental law of active management.

In particular, this chapter will address the following topics:

- How to characterize, justify and measure key types of alpha factors
- How to create alpha factors using financial feature engineering
- How to use `zipline` offline to test individual alpha factors
- How to use `zipline` on Quantopian to combine alpha factors and identify more sophisticated signals
- How the **information coefficient** (**IC**) measures an alpha factor's predictive performance
- How to use `alphalens` to evaluate predictive performance and turnover

Engineering alpha factors

Alpha factors are transformations of market, fundamental, and alternative data that contain predictive signals. They are designed to capture risks that drive asset returns. One set of factors describes fundamental, economy-wide variables such as growth, inflation, volatility, productivity, and demographic risk. Another set consists of tradeable investment styles such as the market portfolio, value-growth investing, and momentum investing.

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, where the factors have high returns over the long run to compensate for their low returns during bad times, or 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 Bn, identified some 250 factors that had been published with empirical evidence in reputable journals by 2015 and 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 that makes it plausible that it reflects risks that the market would compensate.

The data transformations include simple arithmetic such as absolute or relative changes of a variable over time, ratios between data series, or aggregations over a time window such as a simple or exponential moving average. They also include calculations that have emerged from the technical analysis of price patterns such as the relative strength index of demand versus supply and numerous metrics familiar from the fundamental analysis of securities.

Important factor categories

In an idealized world, categories of risk factors should be independent of each other (orthogonal), 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 will hold only approximately. We will address how to derive synthetic, data-driven risk factors using unsupervised learning, in particular principal and independent component analysis in `Chapter 12`, *Unsupervised Learning*.

We will review the key categories for factors derived from market, fundamental, and alternative data, and typical metrics used to capture them. We will also demonstrate how to implement these factors for algorithms tested on the Quantopian platform using built-in factors, custom computations using `numpy` and `pandas`, or the `talib` library for technical analysis.

Momentum and sentiment factors

Momentum investing follows the adage: the trend is your friend or let your winners run. Momentum risk factors are designed to go long assets that have performed well while going short assets with poor performance over a certain period.

The premise of strategies relying on this factor is that asset prices exhibit a trend, reflected in positive serial correlations. Such price momentum would defy 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.

Rationale

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 reasons reflect biases of under-reaction and over-reaction 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.

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, FX, 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, for example, when commodity production takes significant amounts of time to adjust to demand trends. Oil production may lag increased demand from a booming economy for years, and persistent supply shortages can trigger and support upward price momentum.

Market microstructure effects can also create price momentum related to behavioral patterns that motivate investors to buy products and 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 proportion 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 had negative performance. The automatic rebalancing of portfolios using these strategies tends to reinforce price momentum.

Key metrics

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:

Factor	Description
Relative Strength Indicator (RSI)	The RSI compares recent price changes across stocks to identify stocks as overbought or oversold. A high RSI (example, above 70) indicates overbought, and a low RSI (example below 30) indicates oversold. It uses the average price change for a given number of prior trading days with positive price changes $\bar{\Delta p}^{\text{up}}$ and negative prices changes $\bar{\Delta p}^{\text{down}}$ to compute: $\text{RSI} = 100 - \frac{100}{1 + \bar{\Delta p}^{\text{up}} / \bar{\Delta p}^{\text{down}}}$
Price momentum	This factor computes the total return for a given number of prior trading days. In the academic literature, it is common to use the last 12 months but exclude the most recent month because of a short-term reversal effect frequently observed in most recent price movements, but shorter periods have also been widely used.
12-month price momentum Vol Adj	The 12-month price momentum adjusted for volatility factor 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 trend (adjusted for volatility) using a linear regression on daily prices for a longer and a shorter period, e.g., a one year and three 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:

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 and 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.

Value factors

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 provide signals to buy undervalued assets, that is, those that are relatively cheap and sell those that are overvalued and expensive. For this reason, at the core of any value strategy is a valuation model that estimates or proxies 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 (for example, two standard deviations).

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

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

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, for example, levels and changes in the capital account balance, economic activity, inflation, or fund flows. In equities and corporate credit, value factors go back to Graham and Dodd's Security Analysis in the 1930s, since made famous by Warren Buffet. 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 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 the next chapter). Over longer time horizons, market value trades also leverage seasonal effects in equities and commodities.

Cross-asset relative value strategies focus on the relative mispricing of different assets. For example, convertible bond arbitrage involves trades on the relative value between the stock, credit, and volatility of a single company. Relative value also includes trades between credit and equity volatility, using credit signals to trade equities or relative value trades between commodities and equities.

Rationale

There are both rational and behavioral explanations for the existence of the value effect. We will cite a few prominent examples from a wealth of research with further references listed in the GitHub repository.

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.

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

Key metrics

There is a large number of valuation proxies computed from fundamental data. These factors can be combined as inputs into a machine learning valuation model to predict prices. We will see examples of how some of these factors are used in practice 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 similar as for 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.
EBITDA to enterprise value	This ratio measures a company's EBITDA (Earnings before interest, taxes, depreciation, and amortization), which is a proxy for cash flow relative to its enterprise value.

Earnings yield (1 Yr trailing)	This ratio divides the sum of earnings for the past 12 months by the last market (close) price.
Earnings yield (1 Yr forward)	Instead of actual historical earnings, this ratio divides a rolling 12 month forward consensus analyst earnings estimate by the last price, where consensus consists in a (possibly weighted) average of forecasts.
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 Yr Forward Relative to sector	Forecast P/E ratio relative to the corresponding sector P/E. It aims to alleviate the sector bias of the 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 equal, stocks with higher historical sales to price ratios are expected to outperform.
Sales yield FY1	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.

Volatility and size factors

The low volatility factor captures excess returns on stocks with volatility, beta or idiosyncratic risk below average. 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 basic principles of finance. The **Capital Asset Pricing Model** (**CAPM**) and other asset pricing models assert that higher risk should earn higher returns, but in numerous markets and over extended periods, the opposite has been true with less risky assets outperforming their riskier peers.

Rationale

The low volatility anomaly contradicts the hypothesis of efficient markets and the CAPM assumptions. Instead, several behavioral explanations have been advanced.

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 the risk-return profile of a low price, volatile stock as similar to 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 in bull markets and during 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.

Key metrics

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.

Quality factors

The quality factor aims to capture the excess return on companies that are highly profitable, operationally efficient, safe, stable and well-governed, in short, high quality, versus the market. The markets also appear to reward relative earnings certainty and penalize stocks with high earnings volatility. A portfolio tilt towards businesses with high quality has been long advocated by stock pickers that rely on fundamental analysis but 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.

Rationale

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.

Key metrics

Quality factors rely on metrics computed from the balance sheet and income statement that 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*), return on invested capital, low earnings volatility, or a combination of various profitability, 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; 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 EBIT (Earnings before interest and taxes) 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 amount of earnings paid out in dividends to shareholders. Stocks with higher payout ratios were allocated to the top decile while those with lower payout ratios to the bottom decile.
Return on equity (**ROE**)	Ranks stocks based on their historical return on equity and allocates those with the highest ROE to the top decile.

How to transform data into factors

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.

Useful pandas and NumPy methods

NumPy and pandas are the key tools for custom factor computations. The Notebook `00-data-prep.ipynb` in the data directory contains examples of how to create various factors. The notebook uses data generated by the `get_data.py` script in the data folder in the root directory of the GitHub repo and stored in `HDF5` format for faster access. See the notebook `storage_benchmarks.ipynb` in the directory for `Chapter 2`, *Market and Fundamental Data*, on the GitHub repo for a comparison of `parquet`, `HDF5`, and `csv` storage formats for pandas `DataFrames`.

The following illustrates some key steps in computing selected factors from raw stock data. See the Notebook for additional detail and visualizations that we have omitted here to save some space.

Loading the data

We load the Quandl stock price datasets covering the US equity markets 2000-18 using `pd.IndexSlice` to perform a slice operation on the `pd.MultiIndex`, select the adjusted close price and unpivot the column to convert the DataFrame to 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 to month-end frequency using the available adjusted close price:

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

Computing momentum factors

To capture time series dynamics that capture, for example, momentum patterns, we compute historical returns using the `pct_change(n_periods)`, that is, returns over various monthly periods as identified by `lags`. We then convert the wide result back to 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 compounded monthly returns for six 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)
```

Compute factor betas

We will introduce the Fama—French data to estimate the exposure of assets to common risk factors using linear regression in `Chapter 8`, *Time Series Models*. The five Fama—French factors, namely market risk, size, value, operating profitability, and investment have been shown empirically to explain asset returns and are commonly used to assess the risk/return profile of portfolios. 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('return_1m', axis=1)).beta))
```

We will explore both the Fama—French factor model and linear regression in `Chapter 7`, *Linear Models* in more detail. See the notebook for additional examples.

Built-in Quantopian factors

The accompanying notebook `factor_library.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, the inverse of the sales yield introduce preceding, is available as part of the Morningstar fundamentals dataset. It can be used as part of a pipeline that is further described as we introduce the `zipline` library.

TA-Lib

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.

Seeking signals – how to use zipline

Historically, alpha factors used a single input and simple heuristics, thresholds or quantile cutoffs to identify buy or sell signals. ML has proven quite effective in extracting signals from a more diverse and much larger set of input data, including other alpha factors based on the analysis of historical patterns. As a result, algorithmic trading strategies today leverage a large number of alpha 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.

The open source `zipline` library is an event-driven backtesting system maintained and used in production by the crowd-sourced quantitative investment fund Quantopian (`https://www.quantopian.com/`) to facilitate algorithm-development and live-trading. It automates the algorithm's reaction to trade events and provides it with current and historical point-in-time data that avoids look-ahead bias.

You can use it 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.

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 that stores the data in your home folder under `~/.zipline/data/<bundle>`:

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

The architecture – event-driven trading simulation

A `zipline` algorithm will run for a specified period after an initial setup and executes its trading logic when specific events occur. These events are driven by the trading frequency and can also be scheduled by the algorithm, and result in `zipline` calling certain methods. The algorithm maintains state through a `context` dictionary and receives actionable information through a `data` variable containing **point-in-time** (**PIT**) current and historical data. The algorithm returns a `DataFrame` containing portfolio performance metrics if there were any trades, as well as user-defined metrics that can be used to record, for example, the factor values.

You can execute an algorithm from the command line, in a Jupyter Notebook, and by using the `run_algorithm()` function.

An algorithm requires an `initialize()` method that is called once when the simulation starts. This method can be used to add properties to the `context` dictionary that is available to all other algorithm methods or register `pipelines` that perform more complex data processing, such as filtering securities based, for example, on the logic of alpha factors.

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 serves primarily to identify a set of securities the algorithm may trade during the day. The method `handle_data()` is called every minute.

The `Pipeline API` facilitates the definition and computation of alpha factors for a cross-section of securities from historical data. A pipeline defines computations that produce columns in a table with PIT values for a set of securities. It needs to be registered with the `initialize()` method and can then be executed on an automatic or custom schedule. The library provides numerous built-in computations such as moving averages or Bollinger Bands that can be used to quickly compute standard factors but 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.

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 price increases are likely to mean-revert back down over horizons from less than a minute to one month. See the Notebook `single_factor_zipline.ipynby` 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.

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 could 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 would allow us to place long and short orders. We will see in the next chapter 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 the `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.

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 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` that 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 uses a naive method to combine the six individual factors by simply adding the ranks of assets for each of these factors. Instead of equal weights, we would like to take into account the relative importance and incremental information in predicting future returns. The ML algorithms of the next 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 for the evaluation of the predictive performance of alpha factors.

Separating signal and noise – how to use alphalens

Quantopian has open sourced the Python library, `alphalens`, for the performance analysis of predictive stock factors that integrates well with the backtesting library `zipline` and the portfolio performance and risk analysis library `pyfolio` that 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 repo to save some space.

Creating forward returns and factor quantiles

To utilize `alphalens`, we need to provide signals for a universe of assets like those returned by the ranks of the `MeanReversion` factor, and the forward returns earned by investing in an asset for a given holding period. See Notebook `03_performance_eval_alphalens.ipynb` for details.

We will recover the `prices` from the `single_factor.pickle` file as follows (`factor_data` accordingly):

```
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)
```

The GitHub repository's `alpha factor evaluation` Notebook has more detail on how to conduct the evaluation in a sector-specific way.

We can create the `alphalens` input data in the required format using the `get_clean_factor_and_forward_returns` utility function that also returns the signal quartiles 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 phase (set max_loss=0 to see potentially
suppressed Exceptions). max_loss is 35.0%, not exceeded: OK!
```

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	factor	factor_quantile
01/02/15	A	0.07%	-5.70%	-2.32%	4.09%	2618	4
	AAL	-3.51%	-7.61%	-11.89%	-10.23%	1088	2
	AAP	1.10%	-5.40%	-0.94%	-3.81%	791	1
	AAPL	2.45%	-3.05%	8.52%	15.62%	2917	5
	ABBV	-0.17%	-2.05%	-6.43%	-13.70%	2952	5

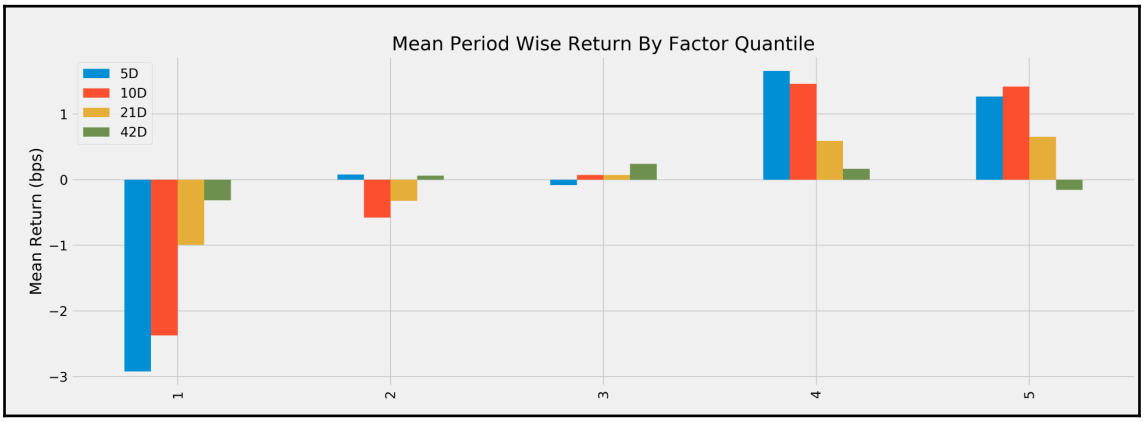
The forward returns and the signal quantiles are the basis for evaluating the predictive power of the signal. Typically, a factor should deliver markedly different returns for distinct quantiles, such as negative returns for the bottom quintile of the factor values and positive returns for the top quantile.

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` and `plot_quantile_returns_bar` from the `plotting` modules:

```
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);
```

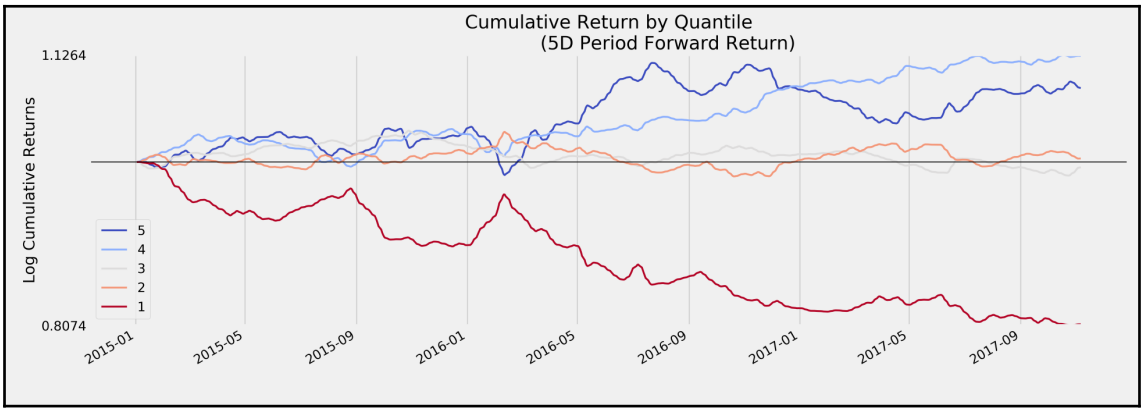
The result is a bar chart that breaks down the mean of the forward returns for the four different holding periods based on the quintile of the factor signal. As you can see, the bottom quintiles yielded markedly more negative results than the top quintiles, except for the longest holding period:



The 10D holding period provides slightly better results for the first and fourth quartiles. We would also like to see the performance over time of investments driven by each of the signal quintiles. We will calculate daily, as opposed to average returns for the 5D holding period, and `alphalens` will adjust the period returns to account for the mismatch between daily signals and a longer holding period (for details, see docs):

```
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 shows that, for most of this three-year period, the top two quintiles significantly outperformed the bottom two quintiles. However, as suggested by the previous plot, signals by the fourth quintile produced a better performance than those by the top quintile:

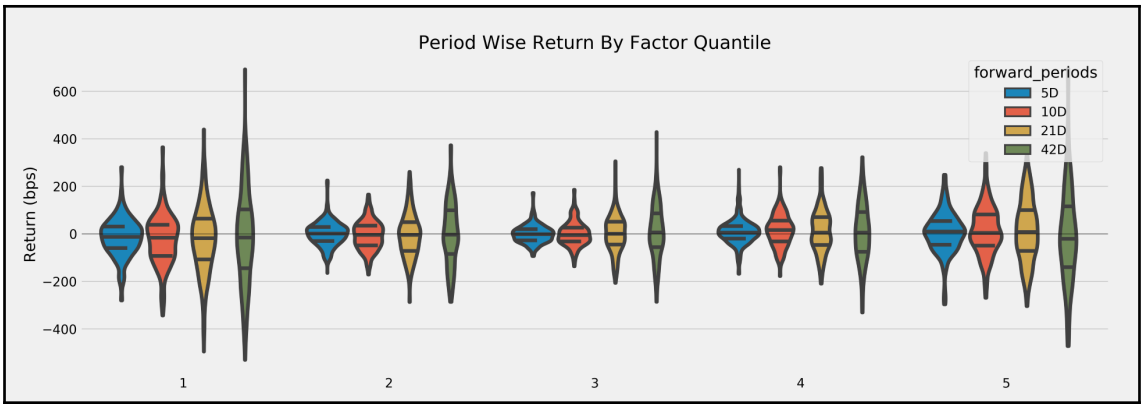


A factor that is useful for a trading strategy shows the preceding pattern where cumulative returns develop along clearly distinct paths because this allows for a long-short strategy with lower capital requirements and correspondingly lower exposure to the overall market.

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 highlights that the range of daily returns is fairly wide and, 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:



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, 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**) that 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 that breaks the IR down into a combination of forecasting skill and a strategy's ability to effectively leverage the forecasting skills.

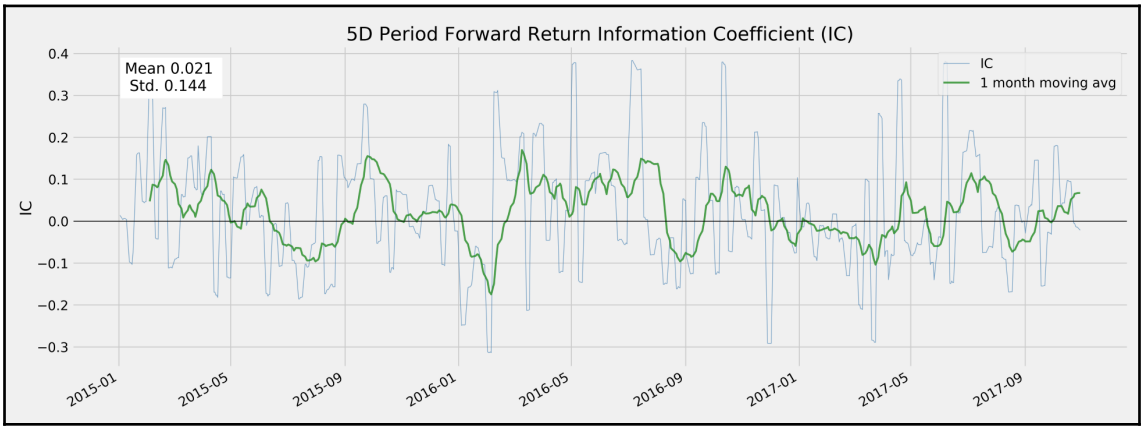
The goal of alpha factors is the accurate directional prediction of future returns. Hence, a natural performance measure is the correlation between an alpha factor's predictions and the forward returns of the target assets.

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

We can obtain the information coefficient using `alphalens`, which relies on `scipy.stats.spearmanr` under the hood (see the repo for an example on 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 one-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']])
```

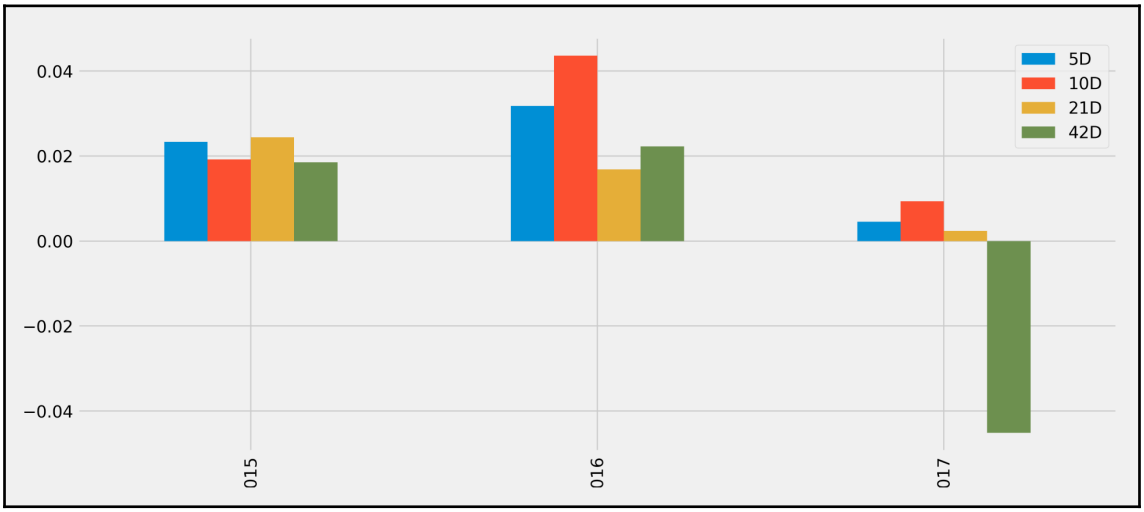
This time series plot shows extended periods with significantly positive moving-average IC. An IC of 0.05 or even 0.1 allows for significant outperformance if there are sufficient opportunities to apply this forecasting skill, as the fundamental law of active management will illustrate:



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 following chart:



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 `create_summary_tear_sheet(alphalens_data)` creates IC summary statistics, where 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	0.01	0.02	0.01	0.00
IC Std.	0.14	0.13	0.12	0.12
Risk-Adjusted IC	0.10	0.13	0.10	0.01
t-stat(IC)	2.68	3.53	2.53	0.14
p-value(IC)	0.01	0.00	0.01	0.89
IC Skew	0.41	0.22	0.19	0.21
IC Kurtosis	0.18	-0.33	-0.42	-0.27

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	59%	83%	83%	41%
Quantile 2	74%	80%	81%	65%
Quantile 3	76%	80%	81%	68%
Quantile 4	74%	81%	81%	64%
Quantile 5	57%	81%	81%	39%

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	0.711	0.452	-0.031	-0.013

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 (see links on GitHub):

- QuantConnect is a competitor to Quantopian
- WorldQuant offers online competition and recruits community contributors to a crowd-sourced hedge fund
- Alpha Trading Labs offers high-frequency focused testing infrastructure with a business model similar to Quantopian
- 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, analyzing 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 covered the use of the `zipline` library for the event-driven simulation of a trading algorithm, both offline and on the Quantopian online platform. We have illustrated the design and evaluation of individual alpha factors to derive signals for an algorithmic trading strategy from market, fundamental, and alternative data, and demonstrated a naive way of combining multiple factors. We also introduced the `alphalens` library that permits the comprehensive evaluation of the predictive performance and trading turnover of signals.

The portfolio construction process, in turn, takes a broader perspective and aims at the optimal sizing of positions from a risk and return perspective. We will now 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 and how to address these challenges.