O'REILLY

8

Evaluate Factor Risk and Performance with Alphalens Reloaded

Factor investing is a strategic approach where assets are chosen based on attributes or **factors** that are associated with higher returns. This method differs from traditional investment strategies which focus on asset classes like stocks, bonds, or sectors. Factor investing emphasizes the underlying drivers of risk and return in securities. The crux of factor investing lies in the systematic identification and harnessing of these and other factors. By understanding the sources of risk and return, we can aim for returns above traditional benchmarks. It's essential to note, however, that while factor investing can enhance portfolio diversification and potential returns, it does not eliminate risk. Market conditions, economic changes, and other externalities can influence the effectiveness of factor-based strategies at any given time.

In ***Chapter 5***, *Build Alpha Factors for Stock Portfolios*, we explored recipes to construct alpha factors. In this chapter, we explore how to analyze the risk and performance of these alpha factors using Alphalens Reloaded. Alphalens Reloaded is a library designed specifically for performance analysis of predictive alpha factors. It's useful for assessing the quality of signals generated from various factors, which allow us to evaluate how well these signals predict future returns. Alphalens Reloaded integrates with Zipline Reloaded and turns the output of backtests into statistics and visualizations. The library provides various utilities, including tear sheets that consolidate performance metrics and visualizations. These tear sheets can show cumulative returns, turnover analysis, and information coefficients, among other insights.

In this chapter, we present the following recipes:

- Preparing backtest results
- Evaluating the information coefficient
- Examining factor return performance
- Evaluating factor turnover

Preparing backtest results

Zipline Reloaded is a robust backtesting library that has an integrated ecosystem of tools designed to assess trading strategy performance. This ecosystem makes it easier for traders to transition from strategy development to evaluation. An example of an integrated tool is Alphalens Reloaded which is the focus of this chapter.

We learned in ***Chapter 7***, *Event-Based Backtesting Factor Portfolios with Zipline Reloaded* that the output DataFrame of a Zipline backtest provides a detailed analysis of a trading strategy's performance over a specified historical data period. The output includes metrics like cumulative returns, alpha, beta, Sharpe ratio, and maximum drawdown, among many others. We need to manipulate the output DataFrame to extract some of the data so it's suitable for use with Alphalens Reloaded.

This recipe will walk through the process of extracting the relevant information.

Getting ready…

To install Alphalens Reloaded in your virtual environment, use **pip**:

```
pip install alphalens-reloaded
```

We assume that a file **`mean_reversion.pickle`** exists in the same directory as the code for this recipe. The file is the cached output from the Zipline backtest we ran in ***Chapter 5***, *Build Alpha Factors for Stock Portfolios*.

How to do it…

This recipe focuses on manipulating the output of the Zipline backtest using pandas. There's one Alphalens Reloaded method we can use to create DataFrame used for most analysis in Alphalens Reloaded.

1. Import the libraries we need for the analysis:

```
import pandas as pd
from alphalens.utils import get_clean_factor_and_forward_returns
```

2. Read in the cached backtest output into a DataFrame:

```
mean_reversion = pd.read_pickle('mean_reversion.pickle')
```

3. Construct a DataFrame with symbols in the columns and dates in the rows:

```
prices = pd.concat(
    [df.to_frame(d) for d, df in mean_reversion.prices.dropna().items()],
    axis=1
).T
```

IMPORTANT

A Python list comprehension is a concise way to create lists by iterating over an iterable and applying an expression to each element. It condenses a loop and a list appending operation into a single line of code, making it more readable and elegant. The syntax includes brackets containing an expression followed by a for clause, as we see in Step 3, above.

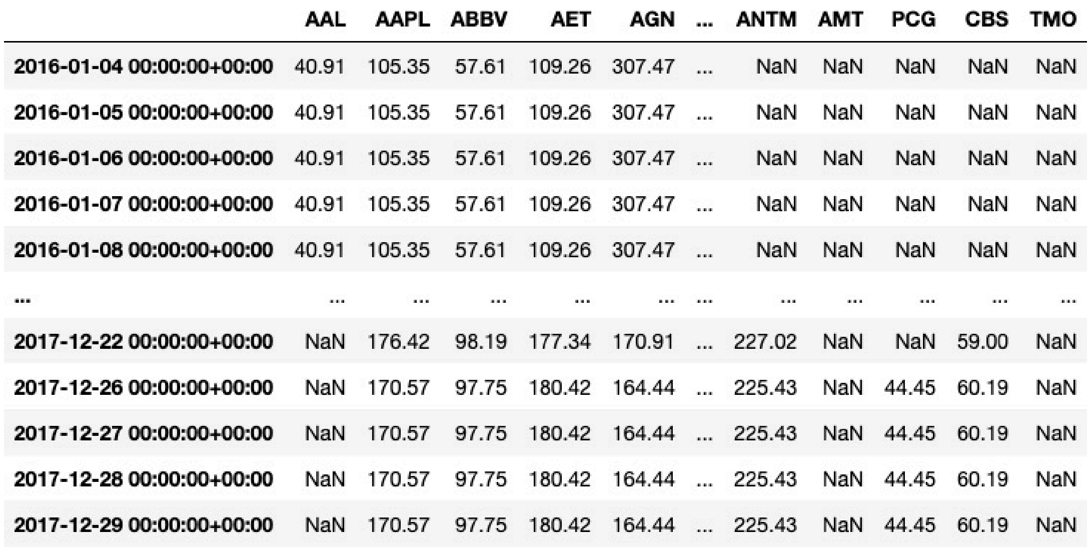
4. Convert column names to strings:

```
prices.columns = [col.symbol for col in prices.columns]
```

5. Normalize the timestamps to midnight, preserving time zone information:

```
prices.index = prices.index.normalize()
```

The result is the following DataFrame containing the prices of each asset in the backtest for each day:



	AAL	AAPL	ABBV	AET	AGN	...	ANTM	AMT	PCG	CBS	TMO
2016-01-04 00:00:00+00:00	40.91	105.35	57.61	109.26	307.47	...	NaN	NaN	NaN	NaN	NaN
2016-01-05 00:00:00+00:00	40.91	105.35	57.61	109.26	307.47	...	NaN	NaN	NaN	NaN	NaN
2016-01-06 00:00:00+00:00	40.91	105.35	57.61	109.26	307.47	...	NaN	NaN	NaN	NaN	NaN
2016-01-07 00:00:00+00:00	40.91	105.35	57.61	109.26	307.47	...	NaN	NaN	NaN	NaN	NaN
2016-01-08 00:00:00+00:00	40.91	105.35	57.61	109.26	307.47	...	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...
2017-12-22 00:00:00+00:00	NaN	176.42	98.19	177.34	170.91	...	227.02	NaN	NaN	59.00	NaN
2017-12-26 00:00:00+00:00	NaN	170.57	97.75	180.42	164.44	...	225.43	NaN	44.45	60.19	NaN
2017-12-27 00:00:00+00:00	NaN	170.57	97.75	180.42	164.44	...	225.43	NaN	44.45	60.19	NaN
2017-12-28 00:00:00+00:00	NaN	170.57	97.75	180.42	164.44	...	225.43	NaN	44.45	60.19	NaN
2017-12-29 00:00:00+00:00	NaN	170.57	97.75	180.42	164.44	...	225.43	NaN	44.45	60.19	NaN

Figure 8.1: Price data from the backtest output

6. We repeat a similar process for the factor data. Start by constructing a DataFrame with symbols in the columns and factor rank in the rows:

```
factor_data = pd.concat(
    [df.to_frame(d) for d,
        df in mean_reversion.factor_data.dropna().items()],
    axis=1
).T
```

7. Convert column names to strings:

```
factor_data.columns = [
    col.symbol for col in factor_data.columns]
```

8. Normalize the timestamps to midnight, preserving time zone information:

```
factor_data.index = factor_data.index.normalize()
```

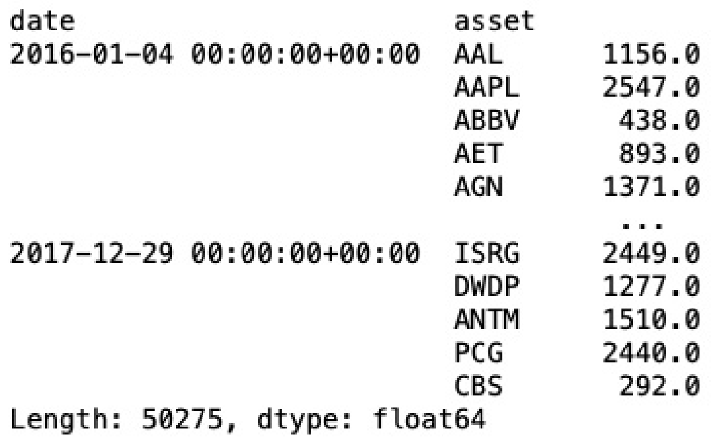
9. Create a MultiIndex with **`date`** in level **`0`** and **`symbol`** in level **`1`**:

```
factor_data = factor_data.stack()
```

10. Rename the MultiIndex:

```
factor_data.index.names = ["date", "asset"]
```

The result is the following a MultiIndex series with the data and asset in the indexes and the factor ranking in the column:



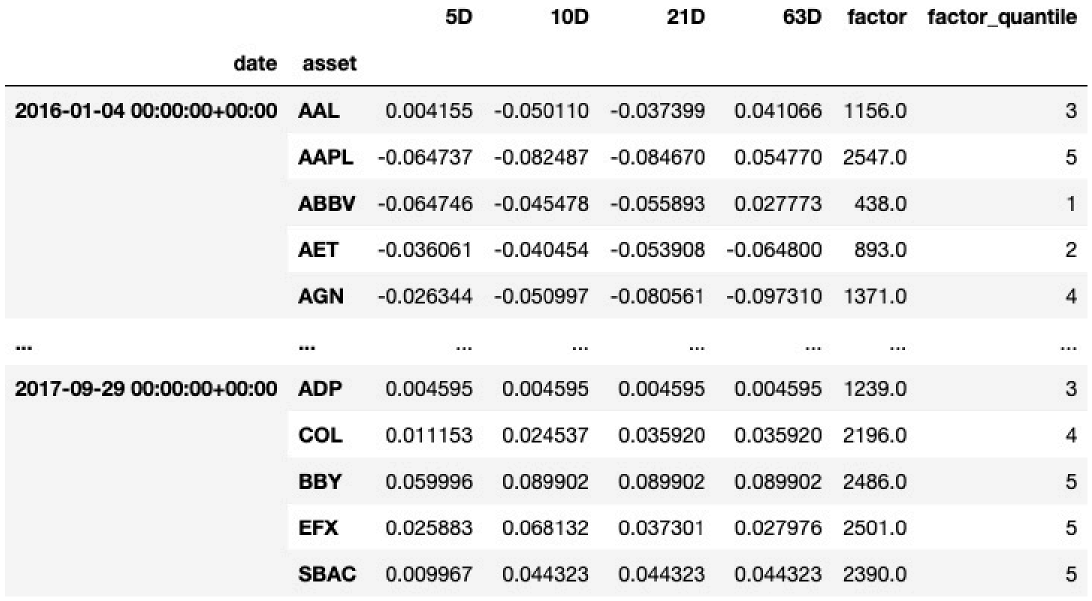
```
date                       asset
2016-01-04 00:00:00+00:00  AAL      1156.0
                           AAPL     2547.0
                           ABBV      438.0
                           AET       893.0
                           AGN      1371.0
                                      ...
2017-12-29 00:00:00+00:00  ISRG     2449.0
                           DWDP     1277.0
                           ANTM     1510.0
                           PCG      2440.0
                           CBS       292.0
Length: 50275, dtype: float64
```

Figure 8.2: MultiIndex Series containing factor ranks for each day and asset

11. The last step is to create a MultiIndex DataFrame with forward returns, factor values, and factor quantiles:

```
alphalens_data = get_clean_factor_and_forward_returns(
    factor=factor_data, prices=prices, periods=(5, 10, 21, 63)
)
```

The result is the following MultiIndex DataFrame containing the data we need to run all the factor analysis using Alphalens Reloaded:



		5D	10D	21D	63D	factor	factor_quantile
date	asset						
2016-01-04 00:00:00+00:00	AAL	0.004155	-0.050110	-0.037399	0.041066	1156.0	3
	AAPL	-0.064737	-0.082487	-0.084670	0.054770	2547.0	5
	ABBV	-0.064746	-0.045478	-0.055893	0.027773	438.0	1
	AET	-0.036061	-0.040454	-0.053908	-0.064800	893.0	2
	AGN	-0.026344	-0.050997	-0.080561	-0.097310	1371.0	4
...	...	...	...	...	...	...	...
2017-09-29 00:00:00+00:00	ADP	0.004595	0.004595	0.004595	0.004595	1239.0	3
	COL	0.011153	0.024537	0.035920	0.035920	2196.0	4
	BBY	0.059996	0.089902	0.089902	0.089902	2486.0	5
	EFX	0.025883	0.068132	0.037301	0.027976	2501.0	5
	SBAC	0.009967	0.044323	0.044323	0.044323	2390.0	5

Figure 8.3: MultiIndex DataFrame containing clean factor and forward returns for each date and asset

How it works…

After the imports, we need to manipulate the backtest output. First, we construct a DataFrame called **`prices`** where each column represents a symbol and each row represents a date. We do this by iterating through each row of **`prices`** using the **`items`** method which returns an iterator. Each key is the date and each value is a DataFrame representing price data. Each of these DataFrames is converted to a column with the date as the column's name using the **`to_frame()`** method. The **`concat()`** function then concatenates these columns horizontally using the **`axis=1`** argument.

Next, we modify the column names of the **`prices`** DataFrame to be the symbols since the original column names were Equity objects. Finally, we adjust the row indices of the **`prices`** DataFrame to normalize the dates to midnight, while retaining any time zone information.

To extract the factor data from the backtest output, we follow similar steps. First, we iterate through **`factor_data`** column dropping any null values, and reshaping the data so that each date corresponds to a unique column. Next, we convert the column names to strings and adjust the dates like before. Finally, the DataFrame is transformed from a two-dimensional table into a one-dimensional series with a hierarchical index. We do this with the **`stack`** method, resulting in a multi-indexed Series where the primary level of the index is the date and the secondary level is the symbol. For clarity, these levels are named **`date`** and **`asset`**.

Now that the DataFrames are prepared, we can call the Alphalens Reloaded **`get_clean_factor_and_forward_returns`** function. The purpose of this function is to associate factor values with future returns. The output, stored in the variable **`alphalens_data`**, is a DataFrame that combines the factor data with the subsequent returns based on the provided prices. This merged structure allows us to analyze how specific factor values might have influenced or related to future asset returns. The resulting DataFrame has a MultiIndex with dates and symbols. The columns in the DataFrame represent different forward returns (5D, 10D, 21D, and 63D for 5-day, 10-day, 21-day, and 63-day returns), the factor values and quantile rankings of the factor values. The quantile rankings segment the factor values into groups, with the shown values like “3” or “5” indicating the specific quantile bucket the factor value for an asset falls into on a given date.

This DataFrame will be used for all subsequent analysis.Top of Form

Bottom of Form

There’s more…

The **`get_clean_factor_and_forward_returns`** function from Alphalens Reloaded is designed to prepare factor data for subsequent analysis. It aligns factor values with forward returns, ensuring that comparisons between factors and future performance are valid. The function also handles data cleaning tasks such as handling missing values. The result is a structured DataFrame that is optimized for factor analysis. The **`get_clean_factor_and_forward_returns`** is flexible in how it aligns the factor data with return data. These are the additional arguments available for controlling this alignment:

- **groupby**: An optional parameter, which, if provided, groups assets based on common characteristics like industry or sector, facilitating sector-neutral analysis.
- **by_group**: A boolean that decides whether to perform computations (like quantile analysis) separately for each group in **groupby** or for the entire dataset.
- **bins**: The number of quantiles to form or specific bin edges. This can be an alternative to the **quantiles** argument.
- **quantiles**: An integer denoting how many quantiles to form. Quantiles segment the factor data, allowing for the examination of returns by factor quantile.
- **periods**: A list of time periods for which forward returns will be computed, e.g., [1, 5, 10] would yield forward returns for 1 day, 5 days, and 10 days.
- **filter_zscore**: Removes outliers based on the z-score. This ensures that extreme factor values, which could be errors or anomalies, do not skew the analysis.
- **groupby_labels**: An optional set of labels for the groups when using the **groupby** argument, useful for more meaningful group naming.
- **max_loss**: The maximum percentage loss that is tolerated in case of missing data. Determines how much missing data is acceptable.
- **zero_aware**: A **boolean** that, if **True**, makes quantile computation treat zeros distinctly, ensuring that they receive their own bin. This is especially useful when zero has a distinct meaning in the factor context.

See also

Factor investing is a broad topic. *Active Portfolio Management: A Quantitative Approach for Producing Superior Returns and Controlling Risk* by Richard C. Grinold and Ronald N. Khan is a staple in the world of quantitative finance and portfolio management. It dives deep into the mathematics and strategies behind managing portfolios actively, including factor-based investing. The book covers various factors, models, and tools, emphasizing the importance of forward returns in the evaluation and prediction of factor performance. For practitioners and students alike, this book provides a comprehensive and rigorous exploration of quantitative techniques in portfolio management. You can buy the book on Amazon here: https://amzn.to/3RtsovG

Finally, the Alphalens Reloaded documentation provides examples and details on how to use the code. The documentation can be found here: **https://alphalens.ml4trading.io**

Evaluating the information coefficient

The **Information Coefficient** (**IC**) is a fundamental metric in quantitative portfolio construction. It gauges the predictive power of a forecast relative to future returns. At its core, IC uses the Spearman rank correlation, a non-parametric measure that assesses how well the relationship between two variables can be described using a monotonic function. The IC's value ranges from -1 to 1 with a positive IC indicating a forecast's genuine predictive power, a value near zero indicating an absence of predictive capacity, and a negative value indicating an inverse relationship between the forecast and subsequent returns.

The origins of the IC can be traced back to the 1960s and 1970s. The preliminary idea of the metric was introduced by Jack L. Treynor in the early 1960s as he presented the topic of correlating investment decisions with corresponding outcomes in his writings on performance measurement. By the mid-1970s, Fischer Black, who is also co-credited for the renowned Black-Scholes option pricing model, further honed and expanded upon the concept in academic literature. Today, IC is a cornerstone metric to gauge the predictive power of alpha factors.

Alphalens Reloaded consolidates extensive quantitative research into a single tool tailored for factor evaluation. It simplifies complex methodologies, offering us an easy way to analyze the predictive capabilities of factors.

This recipe will show us how.

Getting ready…

For this recipe and subsequent recipes, we assume the imports from the last recipe are available and **`alphalens_data`** is defined.

How to do it…

Alphalens Reloaded provides tear sheets that provide consolidated summaries of the analyses presented in this recipe. We'll look at each independently to deepen our understanding.

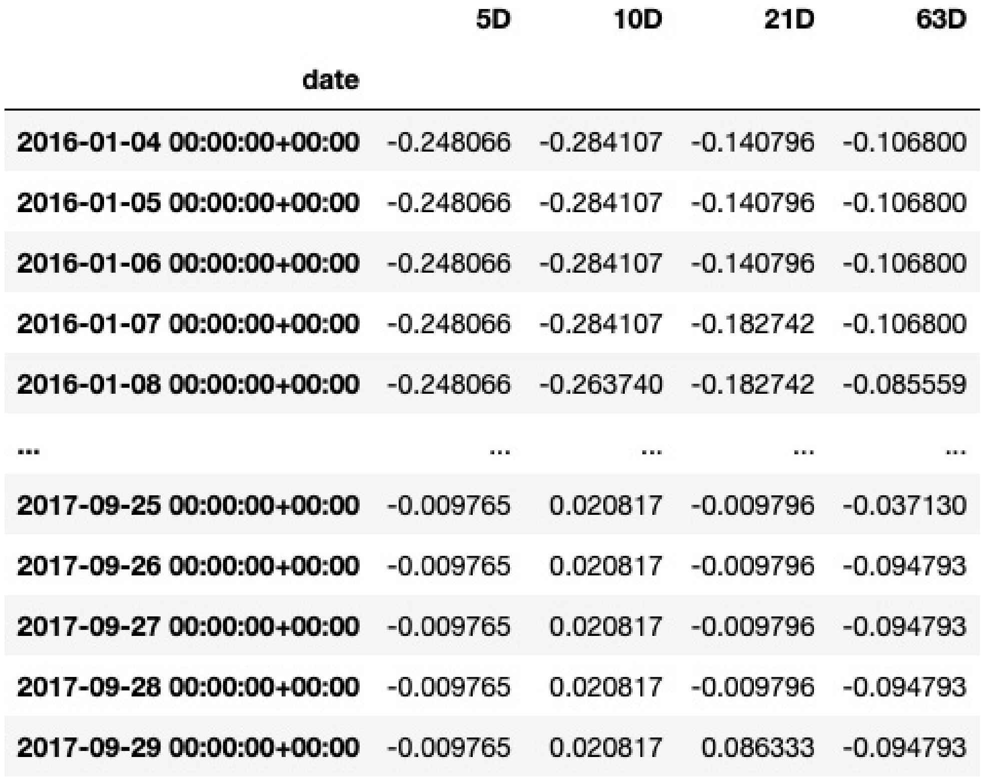
1. Import additional libraries from Alphalens Reloaded to evaluate the IC:

```
from alphalens.performance import (
    factor_information_coefficient,
    mean_information_coefficient,
)
from alphalens.plotting import (
    plot_ic_ts,
    plot_information_table,
)
```

2. Generate the information coefficient for each holding period on each date:

```
ic = factor_information_coefficient(alphalens_data)
```

The result is the following DataFrame with the IC for each day and forward return:



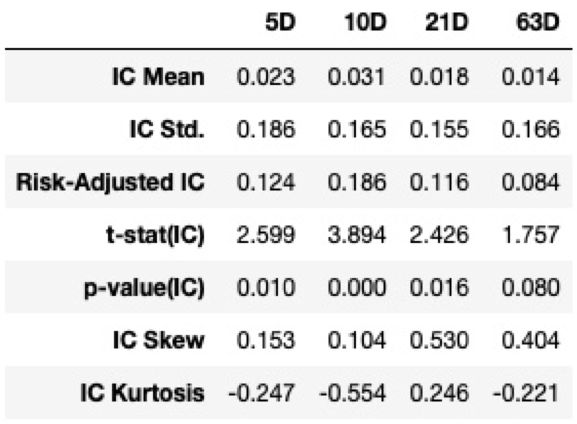
	5D	10D	21D	63D
date				
2016-01-04 00:00:00+00:00	-0.248066	-0.284107	-0.140796	-0.106800
2016-01-05 00:00:00+00:00	-0.248066	-0.284107	-0.140796	-0.106800
2016-01-06 00:00:00+00:00	-0.248066	-0.284107	-0.140796	-0.106800
2016-01-07 00:00:00+00:00	-0.248066	-0.284107	-0.182742	-0.106800
2016-01-08 00:00:00+00:00	-0.248066	-0.263740	-0.182742	-0.085559
...	...	...	...	...
2017-09-25 00:00:00+00:00	-0.009765	0.020817	-0.009796	-0.037130
2017-09-26 00:00:00+00:00	-0.009765	0.020817	-0.009796	-0.094793
2017-09-27 00:00:00+00:00	-0.009765	0.020817	-0.009796	-0.094793
2017-09-28 00:00:00+00:00	-0.009765	0.020817	-0.009796	-0.094793
2017-09-29 00:00:00+00:00	-0.009765	0.020817	0.086333	-0.094793

Figure 8.4: DataFrame containing the IC for each period and forward return

3. Inspect the statistical properties of the IC at each forward return:

   ```
   plot_information_table(ic)
   ```

   The result is the following table containing the mean, standard deviation, and other properties of the IC:



	5D	10D	21D	63D
IC Mean	0.023	0.031	0.018	0.014
IC Std.	0.186	0.165	0.155	0.166
Risk-Adjusted IC	0.124	0.186	0.116	0.084
t-stat(IC)	2.599	3.894	2.426	1.757
p-value(IC)	0.010	0.000	0.016	0.080
IC Skew	0.153	0.104	0.530	0.404
IC Kurtosis	-0.247	-0.554	0.246	-0.221

Figure 8.5: DataFrame containing statistical properties of the IC including mean, standard deviation, risk adjusted IC, t-stat, p-value, skew, and kurtosis

4. Plot the IC over the forward return period to inspect how the alpha decays over longer holding periods as follows:

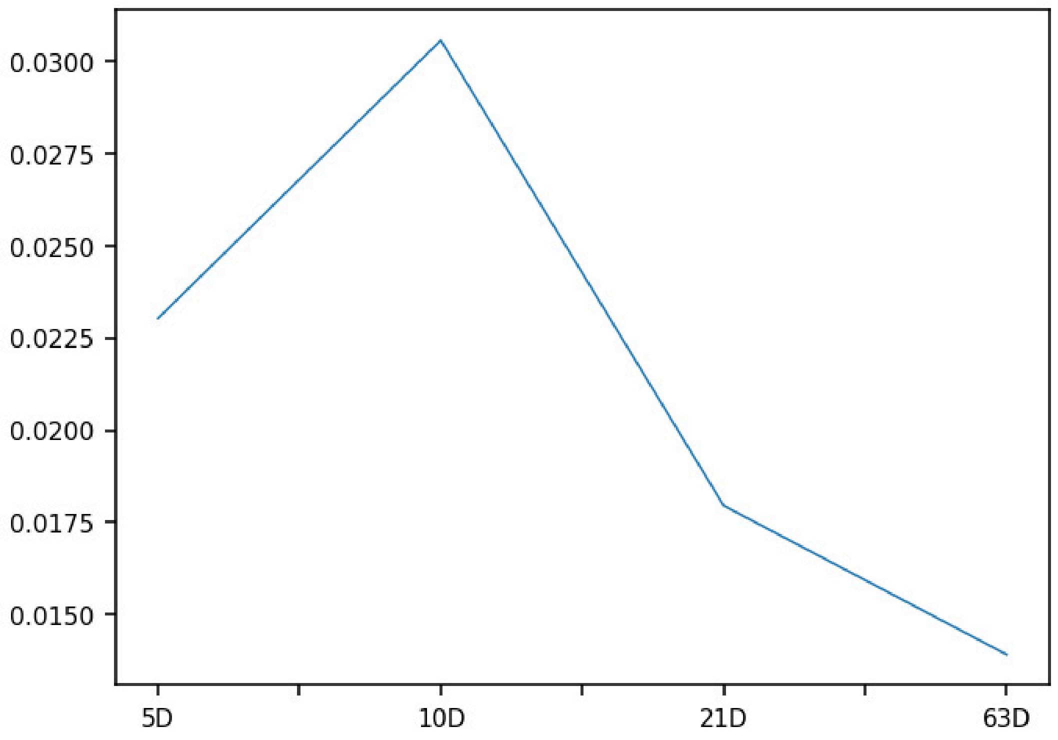


Figure 8.6: Plot demonstrating the decay of the IC as the forward return period lengthens

How it works…

The Spearman rank correlation, often denoted as ρ or Spearman's r, is a non-parametric measure used to assess the strength and direction of the monotonic relationship between two variables. Unlike the Pearson correlation coefficient, which measures linear relationships between continuous variables, Spearman's rank correlation focuses on the relative order of values rather than their absolute magnitudes.

To compute the Spearman rank correlation:

1. Rank each variable separately, assigning ranks from 1 to N for a dataset of N observations. For tied values, assign the average rank.
2. Calculate the difference between ranks for each observation, denoted as d.
3. Square these differences, d^2.

4. Use the following formula to compute ρ:

$$\rho = 1 - \frac{6\sum d^2}{N(N^2 - 1)}$$

Where:

- ρ is the Spearman rank correlation.
- d is the difference between paired ranks.
- N is the number of observations.

The resulting ρ will fall between `-1` and `1`. A value of `1` indicates perfect positive correlation in ranks (i.e., as one variable increases, the other does too), `-1` indicates perfect negative correlation in ranks, and a value close to `0` suggests no significant rank correlation.

In the context of the IC using the Spearman rank correlation:

- **Predicted or forward returns**: These are the returns that a factor model anticipates. They are typically derived from the factor scores, which are usually the output of some quantitative model or strategy.
- **Actual returns**: These are the realized returns of assets over a specific period. They serve as the ground truth against which predicted returns are evaluated.

When computing the Spearman rank correlation for the IC, the ranks of both predicted and actual returns are compared. The differences in ranks (*dd*) between these two sets of values are used in the Spearman formula to determine how well the predicted returns' order (or rank) matches the actual returns' order.

Alphalens Reloaded uses the Spearman rank correlation to compare the ranks of the factor values with the ranks of the forward returns. A high correlation would suggest that assets with higher factor scores tend to have higher (or lower, depending on the sign) future returns, affirming the predictive power of the factor.

In *Figure 8.6*, we observe the predictive power of the alpha factor decaying over time. This is referred to as IC decay. In algorithmic trading, it's important to measure how quickly the correlation between predicted and actual returns diminishes. A rapid IC decay indicates that the factor's predictive power is short-lived. Conversely, a slow IC decay suggests that the signal remains relevant for a more extended period, potentially allowing for longer investment horizons.

There's more…

Alphalens Reloaded ships with plotting functions to visualize how IC evolves through time. To see how the IC using 5-day forward returns evolves through time, use the **plot_ic_ts()** function:

```
plot_ic_ts(ic[["5D"]])
```

The result is a timeseries plot of the IC and it's one-month moving average:

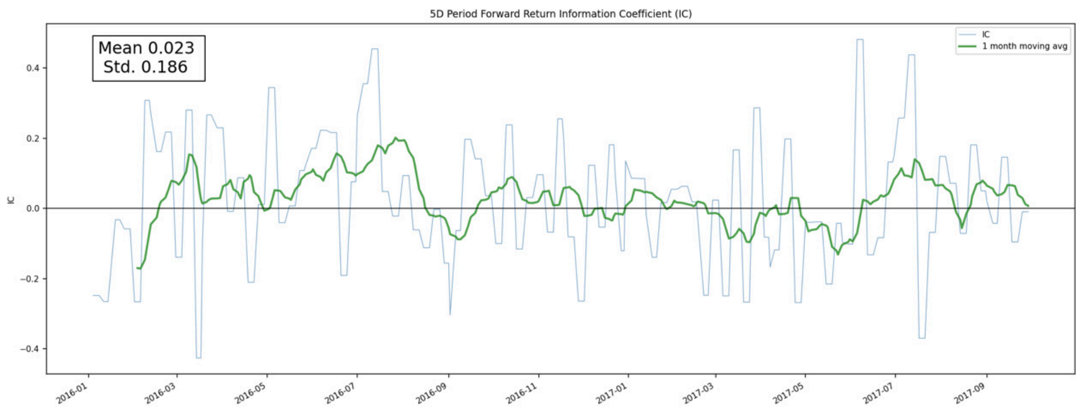


Figure 8.7: Evolution of the IC for 5-day forward returns

Since the return value of the **factor_information_coefficient()** function is a DataFrame, all the methods we learned previously can be used to further analyze the IC. This code resamples the daily IC into the quarterly mean and plots the value for each forward return:

```
ic_by_quarter = ic.resample("Q").mean()
ic_by_quarter.index = ic_by_quarter.index.to_period("Q")
ic_by_quarter.plot.bar(figsize=(14, 6))
```

The result is the following bar chart visualizing the mean IC over each quarter for each forward period :

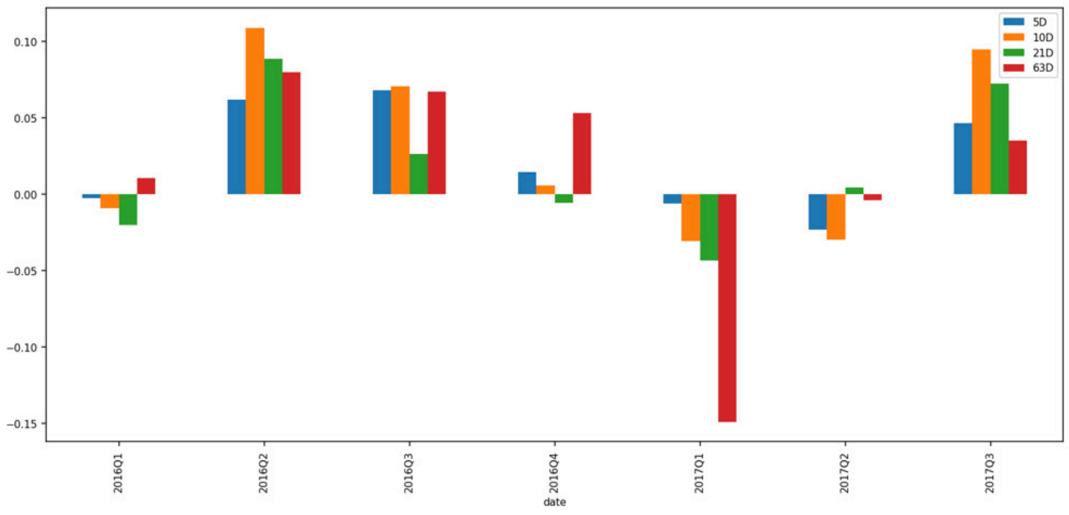


Figure 8.8: Mean IC value per quarter for each forward period

See also

Using the information coefficient is an advanced topic. Here are some resources to get you started:

- Investopedia article describing the information coefficient: https://www.investopedia.com/terms/i/information-coefficient.asp
- Details of the Spearman rank correlation coefficient: https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient

Examining factor return performance

Factor performance relates to the returns generated by a portfolio constructed explicitly based on specific factor values. These factors can encompass any measurable characteristic or set of characteristics about assets, such as value, momentum, size, or volatility, or in our case, mean reversion. In constructing a factor portfolio, assets with high factor values are held long, while those with low values are held short. This approach seeks to isolate the return attributable solely to the factor under consideration. The primary objective behind computing factor returns is to assess the performance of a factor and determine if it offers a premium over some benchmark return. To ensure that these returns are purely a result of the factor, various adjustments, such as demeaning or group adjustments, are applied.

On the other hand, portfolio returns represent the returns generated by a portfolio based on any chosen investment strategy, which might not necessarily be tied to specific factor values. Such portfolios can be constructed based on criteria like expected returns, risk, fundamental analysis, or technical indicators. The weightings of assets in these portfolios can vary, ranging from equal weighting to market-cap weighting or other schemes. The main aim of evaluating normal portfolio returns is to gauge the efficacy of the investment strategy, which can give us insights into the portfolio's performance relative to benchmarks or other investment alternatives. Unlike factor returns, normal portfolio returns provide a view of the portfolio's performance without the intent to isolate returns due to specific factors.

This recipe will demonstrate the available Alphalens Reloaded functionality to examine factor return performance.

How to do it…

We'll explore how to analyze the return data in the *How it works* section, next.

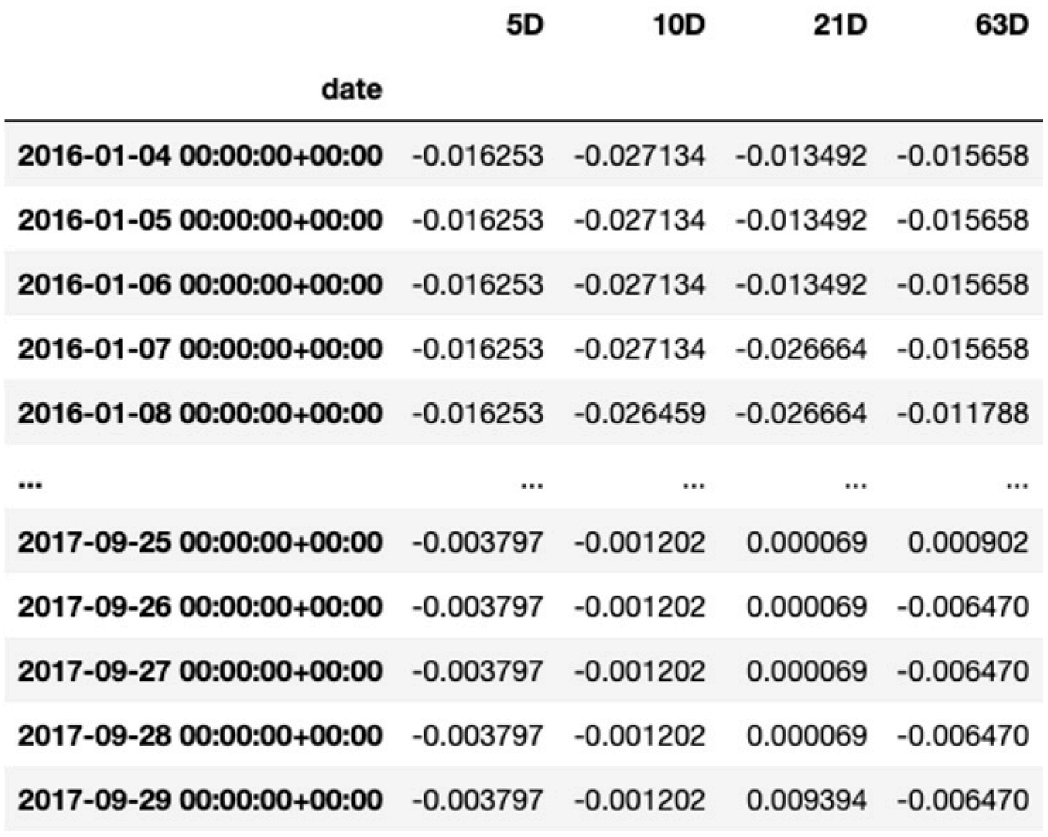
1. Import additional libraries from Alphalens Reloaded to assess the factor performance:

```
from alphalens.performance import (
    factor_returns,
    factor_cumulative_returns,
    mean_return_by_quantile,
    compute_mean_returns_spread,
    factor_alpha_beta,
)
```

2. Compute the period-wise, returns for the portfolio weighted by the factor values:

```
returns = factor_returns(alphalens_data)
```

The result is the following DataFrame with the portfolio returns for each forward return period:



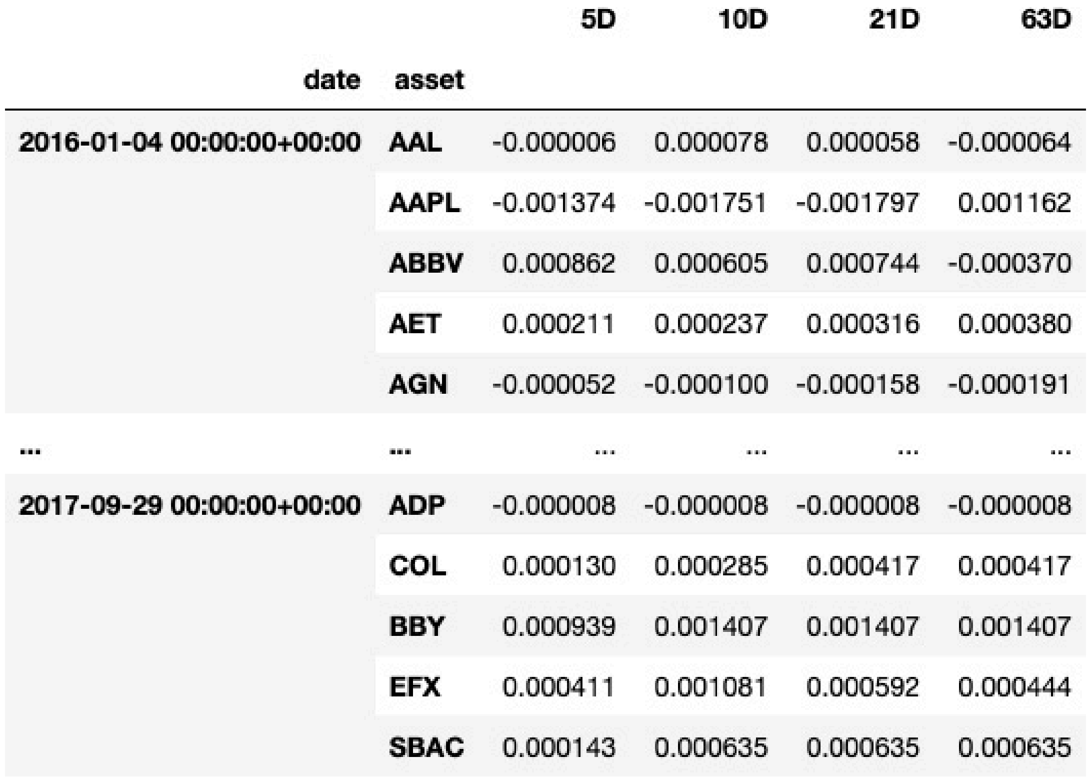
	5D	10D	21D	63D
date				
2016-01-04 00:00:00+00:00	-0.016253	-0.027134	-0.013492	-0.015658
2016-01-05 00:00:00+00:00	-0.016253	-0.027134	-0.013492	-0.015658
2016-01-06 00:00:00+00:00	-0.016253	-0.027134	-0.013492	-0.015658
2016-01-07 00:00:00+00:00	-0.016253	-0.027134	-0.026664	-0.015658
2016-01-08 00:00:00+00:00	-0.016253	-0.026459	-0.026664	-0.011788
...	...	...	...	...
2017-09-25 00:00:00+00:00	-0.003797	-0.001202	0.000069	0.000902
2017-09-26 00:00:00+00:00	-0.003797	-0.001202	0.000069	-0.006470
2017-09-27 00:00:00+00:00	-0.003797	-0.001202	0.000069	-0.006470
2017-09-28 00:00:00+00:00	-0.003797	-0.001202	0.000069	-0.006470
2017-09-29 00:00:00+00:00	-0.003797	-0.001202	0.009394	-0.006470

Figure 8.9: DataFrame with factor-weighted portfolio returns

3. Inspect the returns on a per-asset basis:

```
returns = factor_returns(alphalens_data, by_asset=True)
```

The result is the following MultiIndex DataFrame with each asset return weighted by the associated factor:



		5D	10D	21D	63D
date	**asset**				
2016-01-04 00:00:00+00:00	**AAL**	-0.000006	0.000078	0.000058	-0.000064
	AAPL	-0.001374	-0.001751	-0.001797	0.001162
	ABBV	0.000862	0.000605	0.000744	-0.000370
	AET	0.000211	0.000237	0.000316	0.000380
	AGN	-0.000052	-0.000100	-0.000158	-0.000191
...	**...**	...	...	...	...
2017-09-29 00:00:00+00:00	**ADP**	-0.000008	-0.000008	-0.000008	-0.000008
	COL	0.000130	0.000285	0.000417	0.000417
	BBY	0.000939	0.001407	0.001407	0.001407
	EFX	0.000411	0.001081	0.000592	0.000444
	SBAC	0.000143	0.000635	0.000635	0.000635

Figure 8.10: MultiIndex DataFrame with per-asset, factor-weighted return

4. Visualize the factor-weighted, cumulative portfolio returns for the 5-day forward return against the equal-weighted portfolio:

```
pd.concat(
    {
        "factor_weighted": factor_cumulative_returns(
            alphalens_data,
            period="5D"
        ),
        "equal_weighted": factor_cumulative_returns(
            alphalens_data,
            period="5D",
            equal_weight=True
        ),
    },
    axis=1,
).plot()
```

The result is the following plot demonstrating the cumulative return of the portfolios:

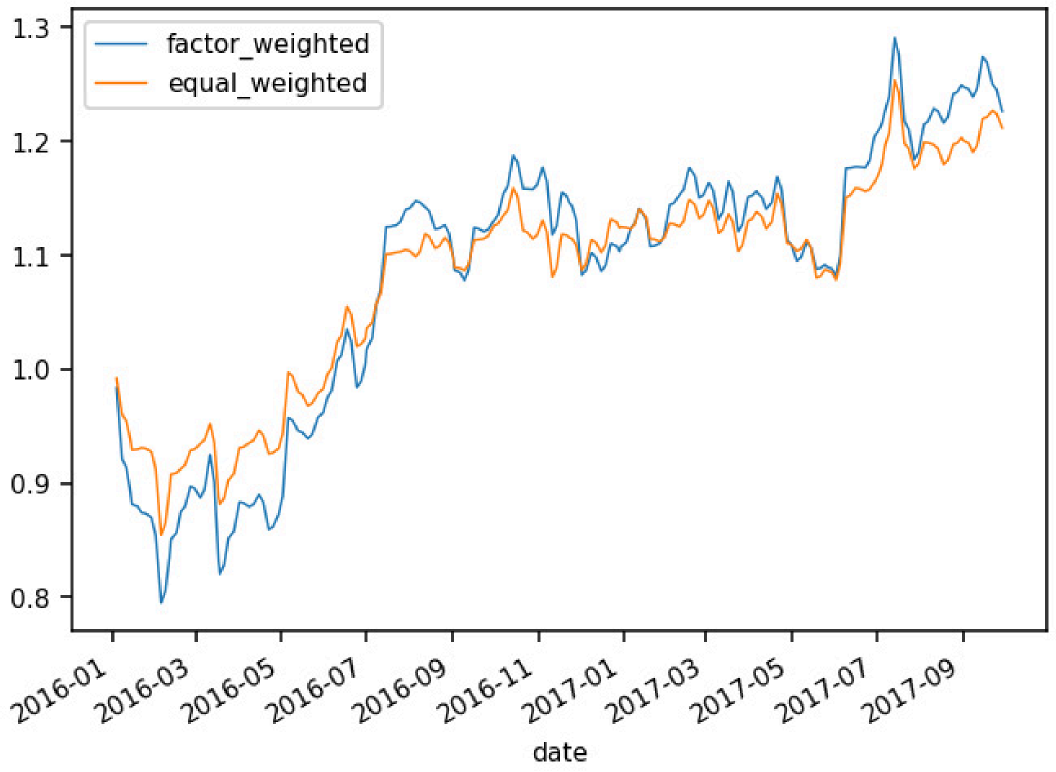


Figure 8.11: Plot with the cumulative return of the 5-day forward, factor-weighted portfolio against the 5-day forward, equal-weighted portfolio

5. Compute the mean return for factor quantiles across the forward returns:

```
mean_returns, _ = mean_return_by_quantile(alphalens_data)
mean_returns.plot.bar()
```

6. The result is the following bar chart visualizing the mean return for each forward return in each quantile:

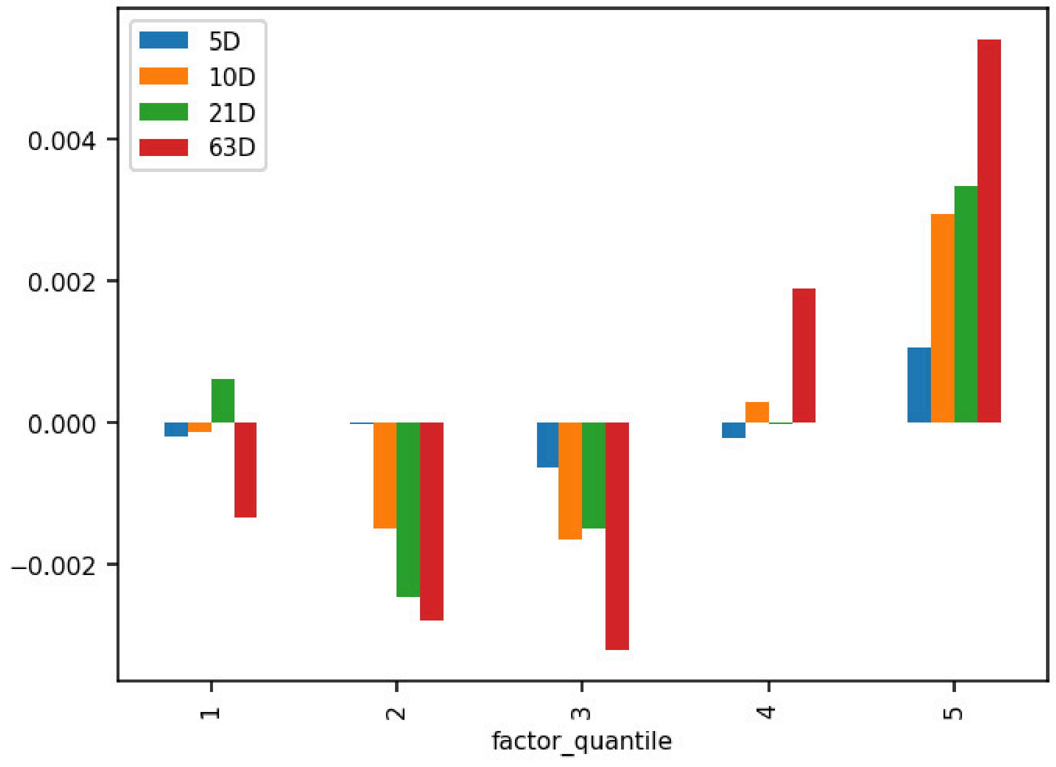


Figure 8.12: Bar chart with the mean factor-weighted portfolio return for each forward return in each quantile

7. Compute the difference in mean returns between the portfolios at the upper and lower quantiles:

```
mean_returns_by_date, _ = mean_return_by_quantile(
    alphalens_data,
    by_date=True
)
mean_return_difference, _ = compute_mean_returns_spread(
    mean_returns=mean_returns_by_date,
    upper_quant=1,
    lower_quant=5,
)
```

8. The result is a DataFrame with the difference between the mean return in the upper and lower quantiles. We can take it a step further and resample the daily values to monthly, take their mean, and plot them:

```
(
    mean_return_difference[["5D"]]
    .resample("M")
```

```
        .mean()
        .to_period("M")
        .plot
        .bar()
    )
```

The result is the following plot of the mean spread per month for the 5-day forward return:

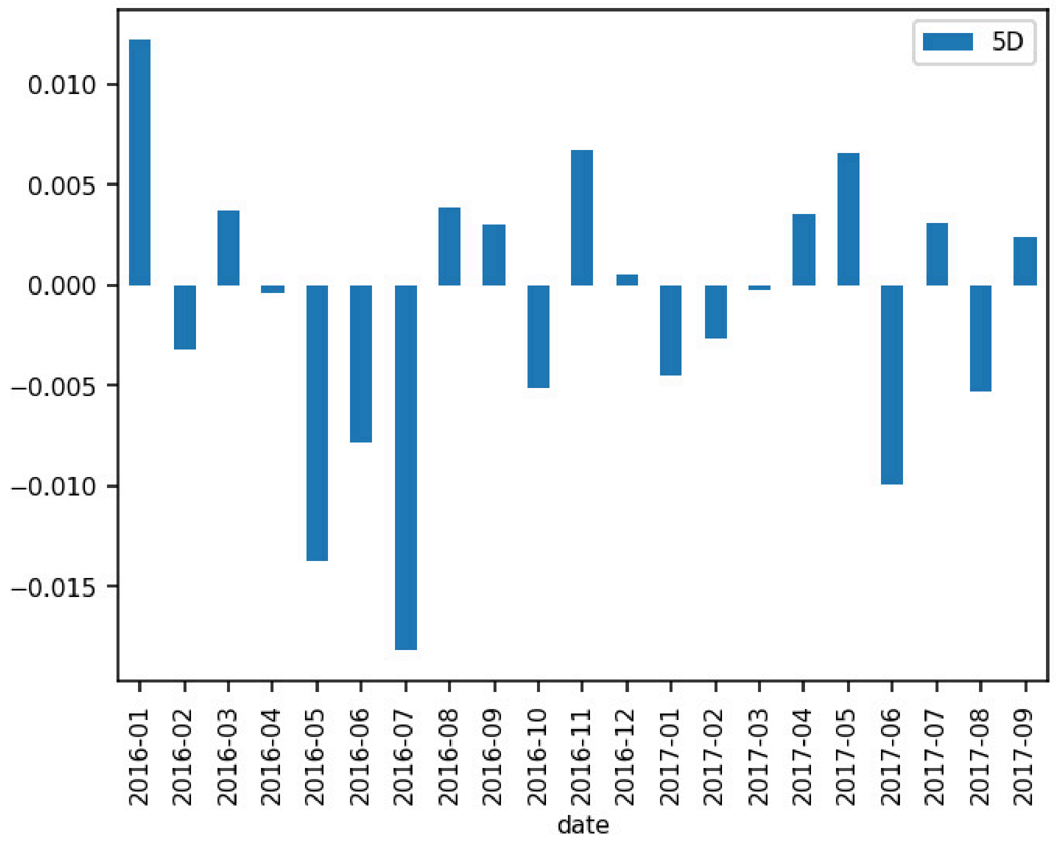


Figure 8.13: Bar chart demonstrating the mean, factor-weighted portfolio returns for the 5-day forward return

How it works…

The **`factor_returns()`** function is designed to compute the factor-weighted returns. It takes in two primary arguments: **`factor_data`**, which is a MultiIndex DataFrame that contains factor values, forward returns for each period, and quantiles and **`periods`**, which is a list of periods for which to compute the returns. For each specified period, it calculates the factor-weighted return by multiplying factor values with forward returns. We can optionally override the factor-weighting and assume an equal-weighted portfolio. This is helpful to compare performance.

The **factor_cumulative_returns()** function is designed to compute the cumulative returns of a factor. The **factor_data** argument is a multi-index DataFrame that contains factor values, forward returns, and group codes. The **period** argument specifies the period for which the cumulative returns are to be calculated, and the **demeaned** argument, when set to **True**, will demean the returns based on the group codes provided in the **alphalens_data** DataFrame. The function computes the mean returns for each date and then determines the cumulative product of these mean returns. The result is a time-series of cumulative returns indexed by date.

The **mean_return_by_quantile()** function computes the mean returns for factors, grouped by quantiles. This is useful to understand how different quantiles of a factor perform over time. In the context of factor portfolio management, the spread between quantiles, which we look at next, is a way to measure the predictive power of the factor. The function offers flexibility through parameters like **by_date**, **demeaned**, **group_adjust**, and **by_group**, which determine how returns are computed and grouped. The core computation groups data by quantiles, and depending on the flags set, by date or group codes, then calculates the mean returns for each group. The result is a MultiIndex DataFrame with the mean returns for each quantile, and optionally, dates and quantiles.

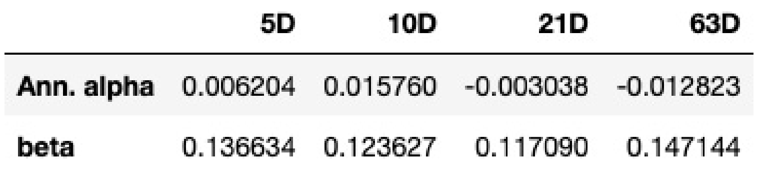
The **compute_mean_returns_spread()** function is designed to compute the mean returns spread for quantiles over a specified period. The spread between the mean return in different quantiles provides insights into how well an alpha factor can differentiate between high and low performing assets over time or across different groups.

There's more…

Using Alphalens Reloaded we can calculate the factor's alpha (representing excess returns) and its beta (indicating market exposure). To do it, we call **`factor_alpha_beta`** which performs a regression using the mean return of the factor universe for each period as the independent variable and the average return from a portfolio, weighted by factor values, for each period as the dependent variable.

```
factor_alpha_beta(alphalens_data)
```

The result is a DataFrame with the annualized alpha and beta across each forward return:



	5D	10D	21D	63D
Ann. alpha	0.006204	0.015760	-0.003038	-0.012823
beta	0.136634	0.123627	0.117090	0.147144

Figure 8.14: DataFrame with the results of a regression on the mean return of the factor universe and average portfolio return.

As we learned in *Chapter 5*, *Build Alpha Factors for Stock Portfolios*, alpha provides a measure of the factor's performance. A positive alpha indicates that the factor has outperformed the benchmark on a risk-adjusted basis, while a negative alpha suggests underperformance. Beta gives insight into the factor's risk relative to the benchmark. A beta greater than 1 indicates that the factor is more volatile than the benchmark, while a beta less than 1 suggests it is less volatile. Over the forward returns of 5- and 10-days, our factor seems to generate positive alpha.

See also

Applied **Quantitative Research** (**AQR**) is a global investment management firm that employs a systematic, research-driven approach to capture investment opportunities across various asset classes. Founded in 1998 by Cliff Asness, David Kabiller, John Liew, and Robert Krail, AQR has become known for its focus on quantitative, data-driven strategies, combining academic research with practical market experience. AQR writes extensively about factor investing. Here's a great paper about measuring factor exposures:

https://www.aqr.com/-/media/AQR/Documents/Insights/Trade-Publications/Measuring-Portfolio-Factor-Exposures-A-Practical-Guide.pdf

Evaluating factor turnover

In algorithmic trading, factor portfolios form the bedrock of many strategies. As we learned in the last recipe, assets are systematically ranked using the Spearman rank correlation from highest to lowest. Following this ranking, assets in the top quartile are bought and those from the bottom quartile are sold. In this context, turnover quantifies the frequency with which assets are bought or sold to rebalance the designated quartiles.

If a factor exhibits high turnover, it might suggest that the factor's signals are not persistent and could lead to higher trading costs if one were to trade based on this factor. Conversely, a factor with low turnover might indicate more stable signals. By analyzing turnover at the quantile level, we can gain insights into the stability of the factor's rankings across different segments of the asset universe. This can be particularly useful when assessing the viability of a factor for portfolio construction, as it provides a lens into the potential transaction costs and the reliability of the factor's signals.

This recipe will demonstrate the available Alphalens Reloaded functionality to examine turnover.

How to do it…

We'll use Alphalens Reloaded to compute the proportion of assets in a factor quantile that were not in that quantile in the previous period. This is a good way to measure aggregate changes in quartiles across the portfolio.

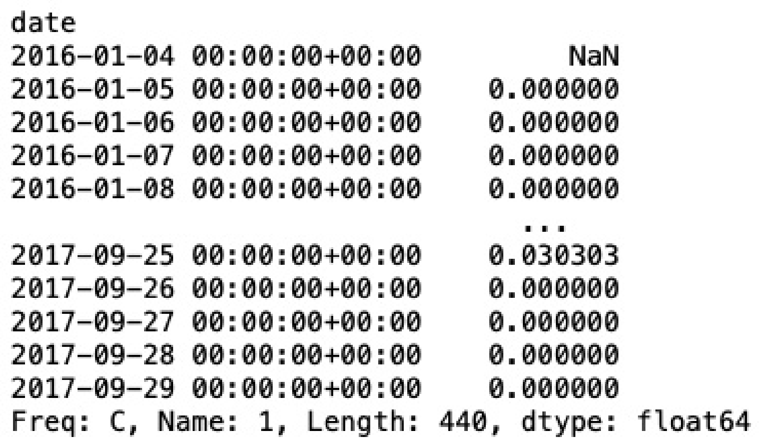
1. Import additional libraries from Alphalens Reloaded to assess the turnover:

```
from alphalens.performance import (
    quantile_turnover,
    factor_rank_autocorrelation
)
from alphalens.plotting import plot_factor_rank_auto_correlation
```

2. Compute the portfolio turnover for the first quantile:

```
turnover = quantile_turnover(alphalens_data, quantile=1)
```

The result is the following Series containing the proportion of assets in the first quantile that were not in that quantile in the previous period:



```
date
2016-01-04 00:00:00+00:00         NaN
2016-01-05 00:00:00+00:00    0.000000
2016-01-06 00:00:00+00:00    0.000000
2016-01-07 00:00:00+00:00    0.000000
2016-01-08 00:00:00+00:00    0.000000
                                ...
2017-09-25 00:00:00+00:00    0.030303
2017-09-26 00:00:00+00:00    0.000000
2017-09-27 00:00:00+00:00    0.000000
2017-09-28 00:00:00+00:00    0.000000
2017-09-29 00:00:00+00:00    0.000000
Freq: C, Name: 1, Length: 440, dtype: float64
```

Figure 8.15: Series containing the proportion of assets in the first quantile that were not in that quantile in the previous period

3. Resample the daily values to monthly and plot the mean turnover for the first quantile:

```
turnover.resample("M").mean().to_period("M").plot.bar()
```

The result is the following bar chart with the mean monthly turnover for the first quantile:

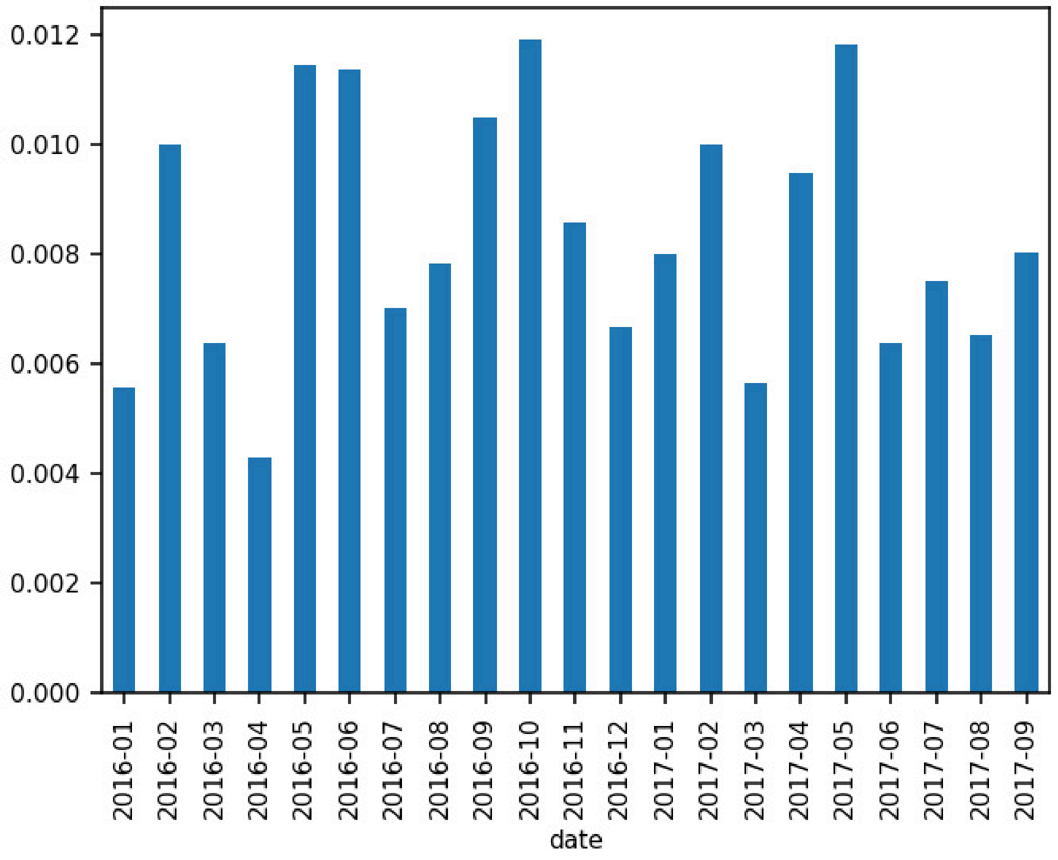


Figure 8.16: Bar chart of the mean monthly turnover for the first quantile

How it works…

The **`quantile_turnover()`** function calculates the turnover for factor quantiles over a specified period. Turnover, in the context of factor portfolio analysis, refers to the rate at which assets within a portfolio are replaced by new assets. High turnover can indicate frequent trading, which might lead to higher transaction costs, while low turnover suggests a more buy-and-hold strategy. By analyzing turnover in the context of factor quantiles, one can assess how stable the factor rankings are over time.

The function takes in the following parameters:

- **`alphalens_data`**: A multi-index DataFrame that contains factor values, forward returns for each period, the factor quantile/bin that factor value belongs to, and (optionally) the group the asset belongs to.

- **`quantiles`**: An integer or sequence of integers. If an integer, it will select the top and bottom **`quantiles`** from the factor values. If a sequence, it will define the exact quantiles to compute.
- **`period`**: The period over which to calculate the turnover. It defaults to '1D', meaning daily turnover.

The function begins by filtering the factor data for the specified quantiles. It then groups the data by date and quantile, and for each group, it calculates the set difference between the current and previous day's assets. This set difference represents the assets that have entered or left the quantile on that day. The function then computes the average turnover for each quantile over the specified period.

There's more…

Another common turnover analysis technique is using the autocorrelation of the factor's daily Spearman rank correlations. Factor rank autocorrelation is a crucial metric in factor portfolio analysis for several reasons:

- **Stability of Factor Ranks**: The autocorrelation of factor ranks provides insights into the stability of the factor's rankings over time. A high autocorrelation indicates that the factor's rankings are relatively stable from one day to the next. This stability can be beneficial for strategies that rely on the persistence of factor rankings.
- **Turnover Implications**: Factors with high rank autocorrelation tend to have lower portfolio turnover. This is because if the ranks of assets based on a factor do not change significantly from one period to the next, the positions in a portfolio constructed based on those ranks would also remain relatively stable. Lower turnover can lead to reduced transaction costs, which can enhance the net returns of a strategy.
- **Factor Consistency**: A consistent factor, one that maintains its rank across assets over time, is generally preferred in quantitative strategies. High rank autocorrelation can be an indicator of such consis-

tency. It suggests that the factor is not frequently changing its opinion about the relative attractiveness of different assets.

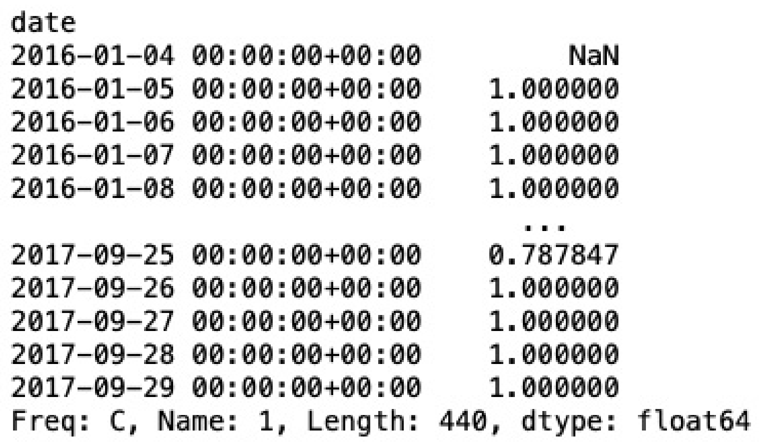
- **Risk Management**: Understanding the autocorrelation of factor ranks can also aid in risk management. If a factor's ranks are highly volatile (low autocorrelation), it might introduce additional risks in a portfolio strategy. Being aware of this can help in designing appropriate risk controls.

We can measure this with the `factor_rank_autocorrelation()` function. The `factor_rank_autocorrelation()` function computes the autocorrelation of the provided factor's daily ranks. Specifically, it calculates the Spearman rank correlation between the factor values of a particular day and those of the subsequent day. The function takes in the following parameters:

- `alphalens_data`: A multi-index DataFrame that contains factor values, forward returns for each period, and the factor quantile/bin that factor values belong to. The indices are date and asset.
- `period`: The period over which the autocorrelation is computed. It's the lag between the factor values being compared.

  ```
  factor_ac = factor_rank_autocorrelation(alphalens_data)
  ```

  The result is a Series with the daily autocorrelation.



```
date
2016-01-04 00:00:00+00:00          NaN
2016-01-05 00:00:00+00:00     1.000000
2016-01-06 00:00:00+00:00     1.000000
2016-01-07 00:00:00+00:00     1.000000
2016-01-08 00:00:00+00:00     1.000000
                                ...
2017-09-25 00:00:00+00:00     0.787847
2017-09-26 00:00:00+00:00     1.000000
2017-09-27 00:00:00+00:00     1.000000
2017-09-28 00:00:00+00:00     1.000000
2017-09-29 00:00:00+00:00     1.000000
Freq: C, Name: 1, Length: 440, dtype: float64
```

Figure 8.17: Series with the autocorrelation of the factor's Spearman rank

IMPORTANT

> *Note in our strategy rebalances on a weekly basis. This means that since there is no opportunity for rebalancing, there will be no turnover except on the weekly rebalance days. We see this reflected in the autocorrelation of 1.*

We can use the **`plot_factor_rank_auto_correlation()`** function to visualize how the autocorrelations change through time.

```
plot_factor_rank_auto_correlation(factor_ac)
```

The result is a time series plot of the autocorrelation.

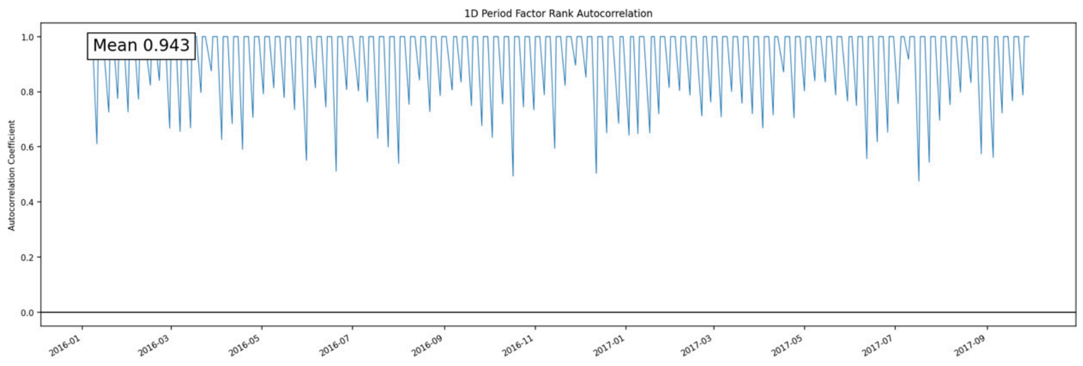


Figure 8.18: Plot visualizing the daily autocorrelation

See also

In case you want to deep dive into the code that computes these metrics, you can do so by visiting the following links:

- Source code for the **`quantile_turnover`** function: https://github.com/stefan-jansen/alphalens-reloaded/blob/d4490ba1290f1f135ed398d1b3601569e0e7996b/src/alphalens/performance.py#L575
- Source code for the **`factor_rank_autocorrelation`** function: https://github.com/stefan-jansen/alphalens-reloaded/blob/d4490ba1290f1f135ed398d1b3601569e0e7996b/src/alphalens/performance.py#L620