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

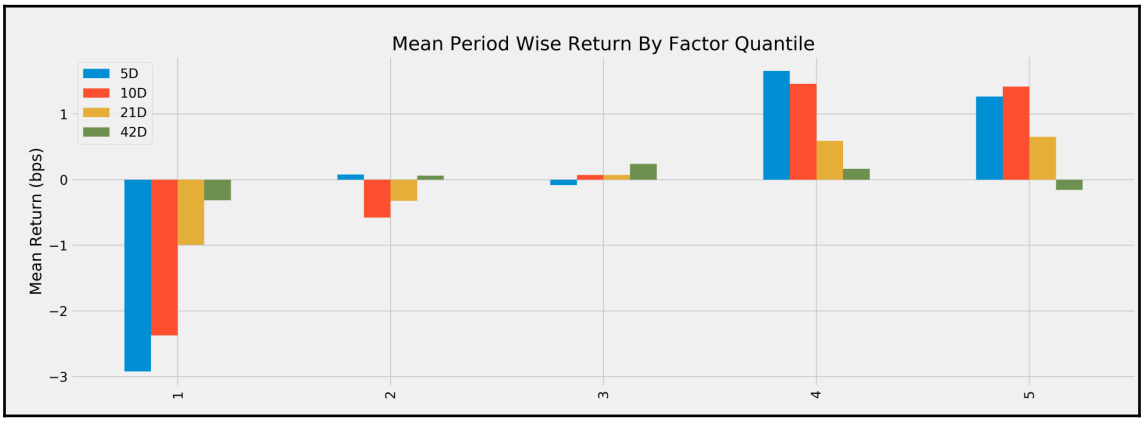
The forward returns and the signal quantiles are the basis for evaluating the predictive power of the signal. Typically, a factor should deliver markedly different returns for distinct quantiles, such as negative returns for the bottom quintile of the factor values and positive returns for the top quantile.

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);
```

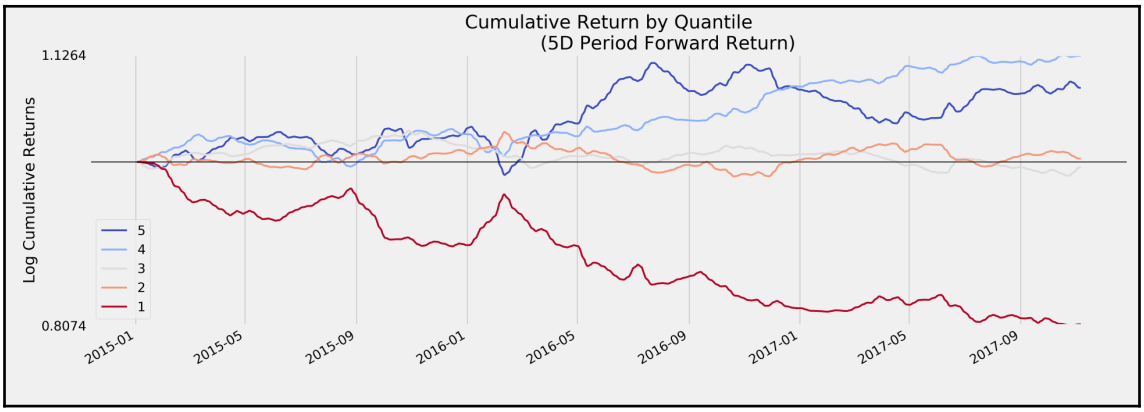
The result is a bar chart that breaks down the mean of the forward returns for the four different holding periods based on the quintile of the factor signal. As you can see, the bottom quintiles yielded markedly more negative results than the top quintiles, except for the longest holding period:



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:

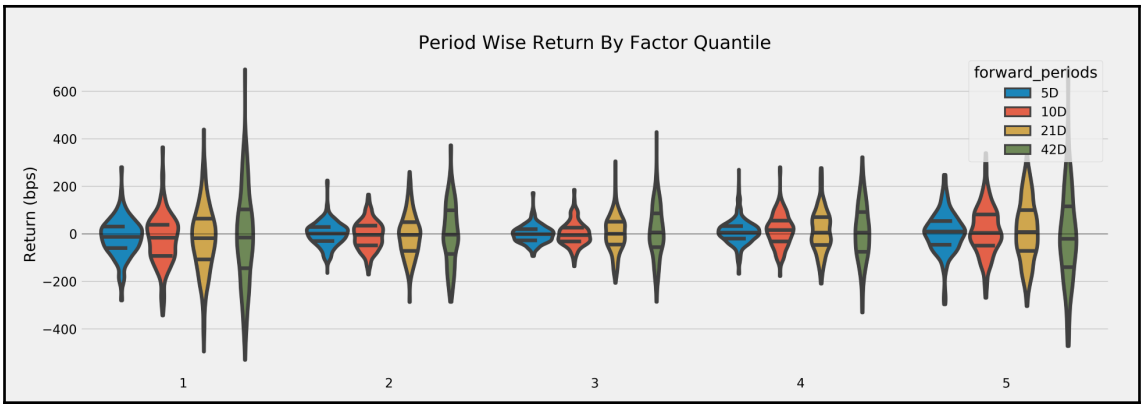


A factor that is useful for a trading strategy shows the preceding pattern where cumulative returns develop along clearly distinct paths because this allows for a long-short strategy with lower capital requirements and correspondingly lower exposure to the overall market.

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.

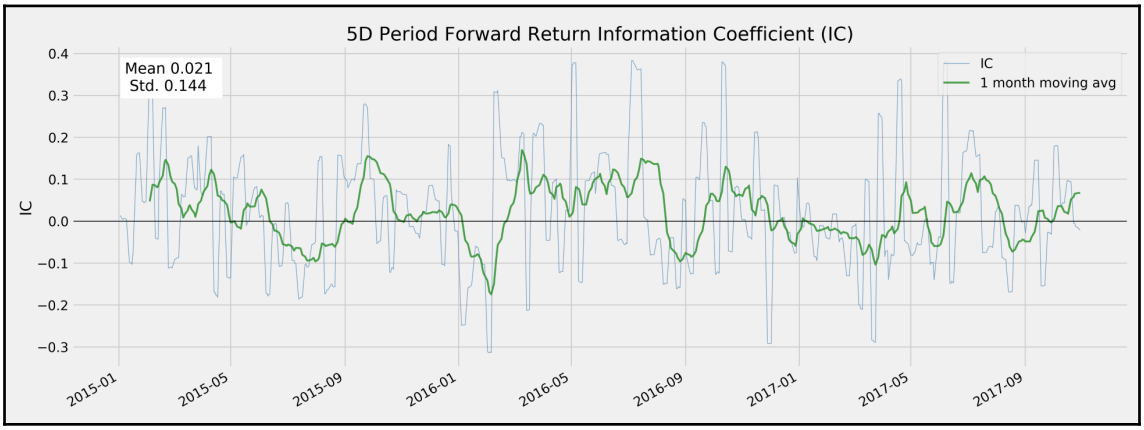
The goal of alpha factors is the accurate directional prediction of future returns. Hence, a natural performance measure is the correlation between an alpha factor's predictions and the forward returns of the target assets.

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']])
```

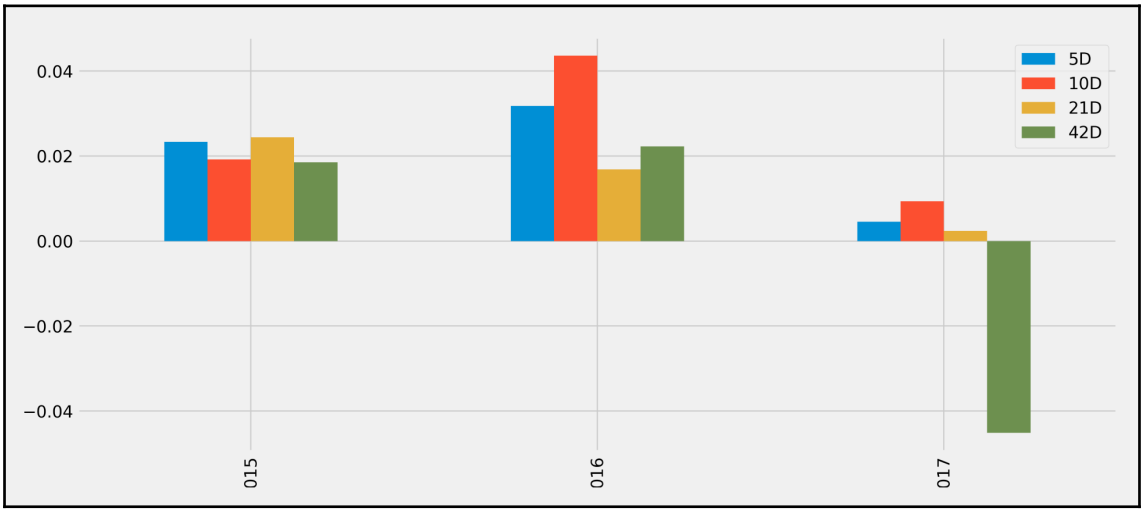
This time series plot shows extended periods with significantly positive moving-average IC. An IC of 0.05 or even 0.1 allows for significant outperformance if there are sufficient opportunities to apply this forecasting skill, as the fundamental law of active management will illustrate:



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.