5
Strategy Evaluation

Alpha factors drive an algorithmic strategy that translates into trades that, in turn, produce a portfolio. The returns and risk of the resulting portfolio determine the success of the strategy. Testing a strategy requires simulating the portfolios generated by an algorithm to verify its performance under market conditions. Strategy evaluation includes backtesting against historical data to optimize the strategy's parameters, and forward-testing to validate the in-sample performance against new, out-of-sample data and avoid false discoveries from tailoring a strategy to specific past circumstances.

In a portfolio context, positive asset returns can offset negative price movements in a non-linear way so that the overall variation of portfolio returns is less than the weighted average of the variation of the portfolio positions unless their returns are perfectly and positively correlated. Harry Markowitz developed the theory behind modern portfolio management based on diversification in 1952, which gave rise to mean-variance optimization: for a given set of assets, portfolio weights can be optimized to reduce risk, measured as the standard deviation of returns for a given expected level of returns.

The **capital asset pricing model** (**CAPM**) introduced a risk premium as an equilibrium reward for holding an asset that compensates for the exposure to a single risk factor—the market—that cannot be diversified away. Risk management has evolved to become much more sophisticated as additional risk factors and more granular choices for exposure have emerged. The Kelly Rule is a popular approach to dynamic portfolio optimization, which is the choice of a sequence of positions over time; it has been famously adapted from its original application in gambling to the stock market by Edward Thorp in 1968.

As a result, there are several approaches to optimize portfolios that include the application of **machine learning** (**ML**) to learn hierarchical relationships among assets and treat their holdings as complements or substitutes with respect to the portfolio risk profile.

In this chapter, we will cover the following topics:

- How to build and test a portfolio based on alpha factors using `zipline`
- How to measure portfolio risk and return
- How to evaluate portfolio performance using `pyfolio`
- How to manage portfolio weights using mean-variance optimization and alternatives
- How to use machine learning to optimize asset allocation in a portfolio context

The code examples for this chapter are in the `05_strategy_evaluation_and_portfolio_management` directory of the companion GitHub repository.

How to build and test a portfolio with zipline

In the last chapter, we introduced `zipline` to simulate the computation of alpha factors from trailing cross-sectional market, fundamental, and alternative data. Now we will exploit the alpha factors to derive and act on buy and sell signals. We will postpone optimizing the portfolio weights until later in this chapter, and for now, just assign positions of equal value to each holding. The code for this section is in the `01_trading_zipline` subdirectory.

Scheduled trading and portfolio rebalancing

We will use the custom `MeanReversion` factor developed in the last chapter—see the implementation in `alpha_factor_zipline_with_trades.py`.

The `Pipeline` created by the `compute_factors()` method returns a table with a long and a short column for the 25 stocks with the largest negative and positive deviations of their last monthly return from its annual average, normalized by the standard deviation. It also limited the universe to the 500 stocks with the highest average trading volume over the last 30 trading days. `before_trading_start()` ensures the daily execution of the pipeline and the recording of the results, including the current prices.

The new `rebalance()` method submits trade orders to the `exec_trades()` method for the assets flagged for long and short positions by the pipeline with equal positive and negative weights. It also divests any current holdings that are no longer included in the factor signals:

```
def exec_trades(data, assets, target_percent):
    """Place orders for assets using target portfolio percentage"""
    for asset in assets:
        if data.can_trade(asset) and not get_open_orders(asset):
            order_target_percent(asset, target_percent)

def rebalance(context, data):
    """Compute long, short and obsolete holdings; place trade orders"""
    factor_data = context.factor_data
    assets = factor_data.index

    longs = assets[factor_data.longs]
    shorts = assets[factor_data.shorts]
    divest = context.portfolio.positions.keys() - longs.union(shorts)

    exec_trades(data, assets=divest, target_percent=0)
    exec_trades(data, assets=longs, target_percent=1 / N_LONGS)
    exec_trades(data, assets=shorts, target_percent=-1 / N_SHORTS)
```

The `rebalance()` method runs according to `date_rules` and `time_rules` set by the `schedule_function()` utility at the beginning of the week, right after `market_open` as stipulated by the built-in `US_EQUITIES` calendar (see docs for details on rules). You can also specify a trade commission both in relative terms and as a minimum amount. There is also an option to define slippage, which is the cost of an adverse change in price between trade decision and execution:

```
def initialize(context):
    """Setup: register pipeline, schedule rebalancing,
        and set trading params"""
    attach_pipeline(compute_factors(), 'factor_pipeline')
    schedule_function(rebalance,
                      date_rules.week_start(),
                      time_rules.market_open(),
                      calendar=calendars.US_EQUITIES)

    set_commission(us_equities=commission.PerShare(cost=0.00075,
min_trade_cost=.01))
set_slippage(us_equities=slippage.VolumeShareSlippage(volume_limit=0.0025,
price_impact=0.01))
```

The algorithm continues to execute after calling the `run_algorithm()` function and returns the same backtest performance `DataFrame`. We will now turn to common measures of portfolio return and risk, and how to compute them using the `pyfolio` library.

How to measure performance with pyfolio

ML is about optimizing objective functions. In algorithmic trading, the objectives are the return and the risk of the overall investment portfolio, typically relative to a benchmark (which may be cash or the risk-free interest rate).

There are several metrics to evaluate these objectives. We will briefly review the most commonly-used metrics and how to compute them using the `pyfolio` library, which is also used by `zipline` and Quantopian. We will also review how to apply these metrics on Quantopian when testing an algorithmic trading strategy.

We'll use some simple notations: let R be the time series of one-period simple portfolio returns, $R=(r_1, ..., r_T)$, from dates 1 to T, and $R^f=(r^f_1, ..., r^f_T)$ be the matching time series of risk-free rates, so that $R^e=R-R^f=(r_1-r^f_1,..., r_T-r^f_T)$ is the excess return.

The Sharpe ratio

The ex-ante **Sharpe ratio** (**SR**) compares the portfolio's expected excess portfolio to the volatility of this excess return, measured by its standard deviation. It measures the compensation as the average excess return per unit of risk taken:

$$\begin{aligned}\mu &\equiv E(R_t)\\ \sigma^2_{R^e} &\equiv \mathrm{Var}(R - R^f)\\ \mathrm{SR} &\equiv \frac{\mu - R_f}{\sigma_{R^e}}\end{aligned}$$

Expected returns and volatilities are not observable, but can be estimated as follows using historical data:

$$\hat{\mu}_{R^e} = \frac{1}{T}\sum_{t=1}^{T} r_t^e$$
$$\hat{\sigma}^2_{R^e} = \frac{1}{T}\sum_{t=1}^{T} (r_t^e - \hat{\mu}_{R^e})^2$$
$$\hat{\mathrm{SR}} = \frac{\hat{\mu}_{R^e}}{\hat{\sigma}_{R_e}}$$

Unless the risk-free rate is volatile (as in emerging markets), the standard deviation of excess and raw returns will be similar. When the SR is used with a benchmark other than the risk-free rate, for example, the S&P 500, it is called the **information ratio**. In this case, it measures the excess return of the portfolio, also called **alpha**, relative to the tracking error, which is the deviation of the portfolio returns from the benchmark returns.

For **independently and identically-distributed** (**iid**) returns, the derivation of the distribution of the estimator of the SR for tests of statistical significance follows from the application of the Central Limit Theorem, according to large-sample statistical theory, to $\hat{\mu}$ and $\hat{\sigma}^2$.

However, financial returns often violate the iid assumptions. Andrew Lo has derived the necessary adjustments to the distribution and the time aggregation for returns that are stationary but autocorrelated returns. This is important because the time-series properties of investment strategies (for example, mean reversion, momentum, and other forms of serial correlation) can have a non-trivial impact on the SR estimator itself, especially when annualizing the SR from higher-frequency data (Lo 2002).

The fundamental law of active management

A high **Information Ratio** (**IR**) implies attractive out-performance relative to the additional risk taken. The Fundamental Law of Active Management breaks the IR down into the **information coefficient** (**IC**) as a measure of forecasting skill, and the ability to apply this skill through independent bets. It summarizes the importance to play both often (high breadth) and to play well (high IC):

$$\mathrm{IR} = \mathrm{IC} * \sqrt{\mathrm{breadth}}$$

The IC measures the correlation between an alpha factor and the forward returns resulting from its signals and captures the accuracy of a manager's forecasting skills. The breadth of the strategy is measured by the independent number of bets an investor makes in a given time period, and the product of both values is proportional to the IR, also known as **appraisal risk** (Treynor and Black).

This framework has been extended to include the **transfer coefficient** (**TC**) to reflect portfolio constraints (for example, on short-selling) that may limit the information ratio below a level otherwise achievable given IC or strategy breadth. The TC proxies the efficiency with which the manager translates insights into portfolio bets (Clarke et al. 2002).

The fundamental law is important because it highlights the key drivers of outperformance: both accurate predictions and the ability to make independent forecasts and act on these forecasts matter. In practice, managers with a broad set of investment decisions can achieve significant risk-adjusted excess returns with information coefficients between 0.05 and 0.15 (if there is space possibly include simulation chart).

In practice, estimating the breadth of a strategy is difficult given the cross-sectional and time-series correlation among forecasts.

In and out-of-sample performance with pyfolio

Pyfolio facilitates the analysis of portfolio performance and risk in-sample and out-of-sample using many standard metrics. It produces tear sheets covering the analysis of returns, positions, and transactions, as well as event risk during periods of market stress using several built-in scenarios, and also includes Bayesian out-of-sample performance analysis.

It relies on portfolio returns and position data, and can also take into account the transaction costs and slippage losses of trading activity. The metrics are computed using the `empyrical` library that can also be used on a standalone basis.

The performance `DataFrame` produced by the `zipline` backtesting engine can be translated into the requisite `pyfolio` input.

Getting pyfolio input from alphalens

However, `pyfolio` also integrates with `alphalens` directly and permits the creation of `pyfolio` input data using `create_pyfolio_input`:

```
from alphalens.performance import create_pyfolio_input

qmin, qmax = factor_data.factor_quantile.min(),
             factor_data.factor_quantile.max()
input_data = create_pyfolio_input(alphalens_data,
                                  period='1D',
                                  capital=100000,
                                  long_short=False,
                                  equal_weight=False,
                                  quantiles=[1, 5],
                                  benchmark_period='1D')
returns, positions, benchmark = input_data
```

There are two options to specify how portfolio weights will be generated:

- `long_short`: If `False`, weights will correspond to factor values divided by their absolute value so that negative factor values generate short positions. If `True`, factor values are first demeaned so that long and short positions cancel each other out and the portfolio is market neutral.
- `equal_weight`: If `True`, and `long_short` is `True`, assets will be split into two equal-sized groups with the top/bottom half making up long/short positions.

Long-short portfolios can also be created for groups if `factor_data` includes, for example, sector info for each asset.

Getting pyfolio input from a zipline backtest

The result of a `zipline` backtest can be converted into the required `pyfolio` input using `extract_rets_pos_txn_from_zipline`:

```
returns, positions, transactions =
         extract_rets_pos_txn_from_zipline(backtest)
```

Walk-forward testing out-of-sample returns

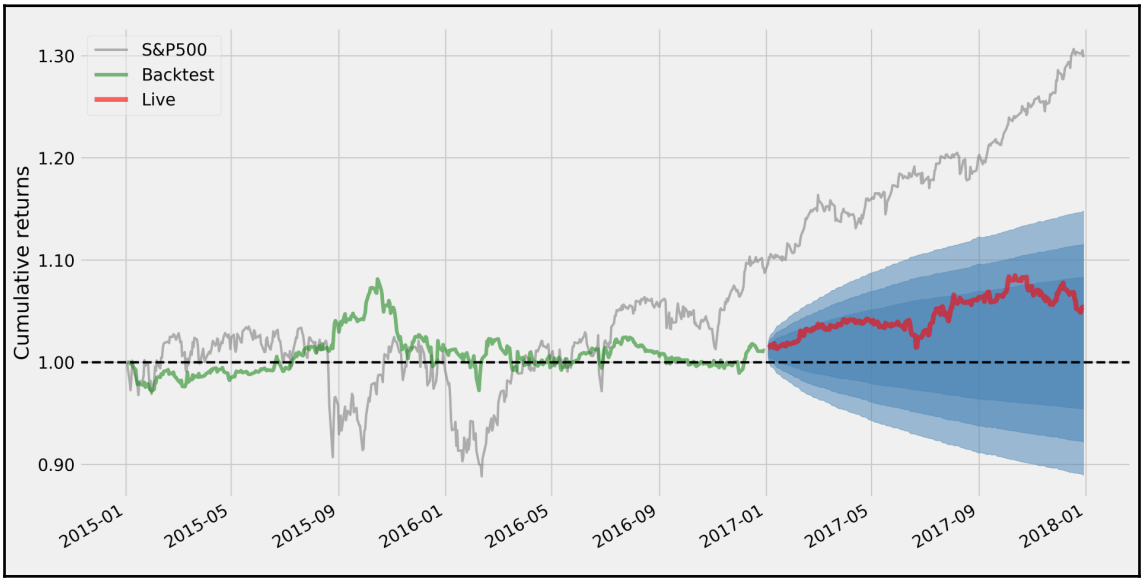
Testing a trading strategy involves backtesting against historical data to fine-tune alpha factor parameters, as well as forward-testing against new market data to validate that the strategy performs well out of sample or if the parameters are too closely tailored to specific historical circumstances.

Pyfolio allows for the designation of an out-of-sample period to simulate walk-forward testing. There are numerous aspects to take into account when testing a strategy to obtain statistically reliable results, which we will address here.

The `plot_rolling_returns` function displays cumulative in and out-of-sample returns against a user-defined benchmark (we are using the S&P 500):

```
from pyfolio.plotting import plot_rolling_returns
plot_rolling_returns(returns=returns,
                     factor_returns=benchmark_rets,
                     live_start_date='2017-01-01',
                     cone_std=(1.0, 1.5, 2.0))
```

The plot includes a cone that shows expanding confidence intervals to indicate when out-of-sample returns appear unlikely given random-walk assumptions. Here, our strategy did not perform well against the benchmark during the simulated 2017 out-of-sample period:



Summary performance statistics

`pyfolio` offers several analytic functions and plots. The `perf_stats` summary displays the annual and cumulative returns, volatility, skew, and kurtosis of returns and the SR. The following additional metrics (which can also be calculated individually) are most important:

- **Max drawdown**: Highest percentage loss from the previous peak
- **Calmar ratio**: Annual portfolio return relative to maximal drawdown
- **Omega ratio**: The probability-weighted ratio of gains versus losses for a return target, zero per default
- **Sortino ratio**: Excess return relative to downside standard deviation
- **Tail ratio**: Size of the right tail (gains, the absolute value of the 95th percentile) relative to the size of the left tail (losses, abs. value of the 5th percentile)
- **Daily value at risk (VaR)**: Loss corresponding to a return two standard deviations below the daily mean
- **Alpha**: Portfolio return unexplained by the benchmark return
- **Beta**: Exposure to the benchmark

```
from pyfolio.timeseries import perf_stats
perf_stats(returns=returns,
           factor_returns=benchmark_rets,
           positions=positions,
           transactions=transactions)
```

For the simulated long-short portfolio derived from the `MeanReversion` factor, we obtain the following performance statistics:

Metric	All	In-sample	Out-of-sample	Metric	All	In-sample	Out-of-sample
Annual return	1.80%	0.60%	4.20%	Skew	0.34	0.40	0.09
Cumulative returns	5.40%	1.10%	4.20%	Kurtosis	3.70	3.37	2.59
Annual volatility	5.80%	6.30%	4.60%	Tail ratio	0.91	0.88	1.03
Sharpe ratio	0.33	0.12	0.92	Daily value at risk	-0.7%	-0.8%	-0.6%
Calmar ratio	0.17	0.06	1.28	Gross leverage	0.38	0.37	0.42
Stability	0.49	0.04	0.75	Daily turnover	4.70%	4.40%	5.10%
Max drawdown	-10.10%	-10.10%	-3.30%	Alpha	0.01	0.00	0.04
Omega ratio	1.06	1.02	1.18	Beta	0.15	0.16	0.03
Sortino Ratio	0.48	0.18	1.37				

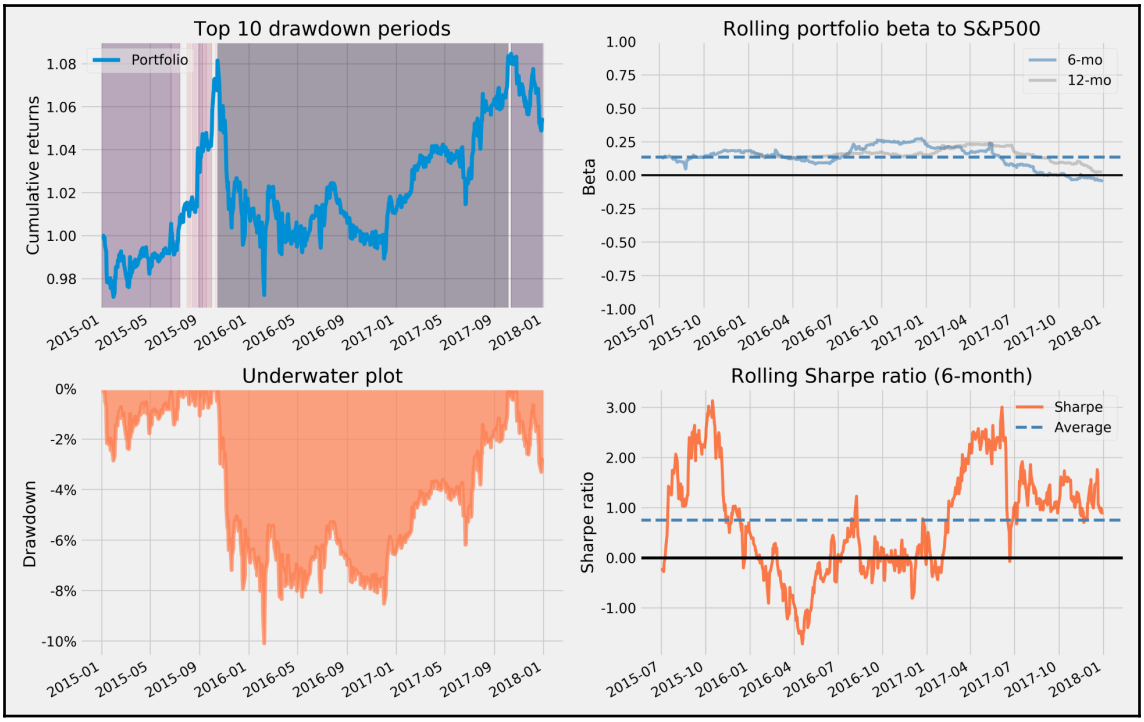
See the appendix for details on the calculation and interpretation of portfolio risk and return metrics.

Drawdown periods and factor exposure

The `plot_drawdown_periods(returns)` function plots the principal drawdown periods for the portfolio, and several other plotting functions show the rolling SR and rolling factor exposures to the market beta or the Fama French size, growth, and momentum factors:

```
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(16, 10))
axes = ax.flatten()

plot_drawdown_periods(returns=returns, ax=axes[0])
plot_rolling_beta(returns=returns, factor_returns=benchmark_rets,
                  ax=axes[1])
plot_drawdown_underwater(returns=returns, ax=axes[2])
plot_rolling_sharpe(returns=returns)
```

This plot, which highlights a subset of the visualization contained in the various tear sheets, illustrates how pyfolio allows us to drill down into the performance characteristics and exposure to fundamental drivers of risk and returns:

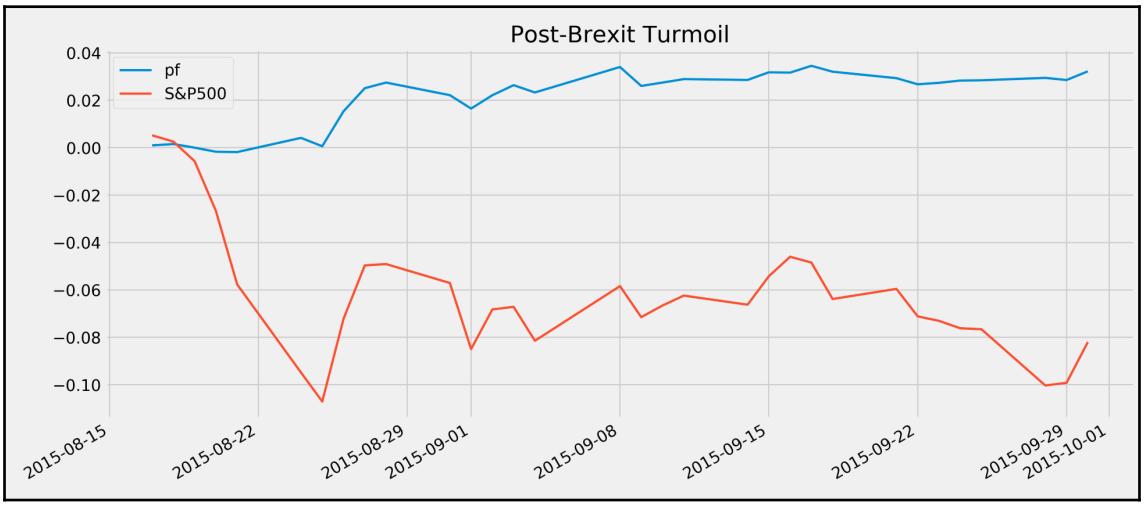


Modeling event risk

Pyfolio also includes timelines for various events that you can use to compare the performance of a portfolio to a benchmark during this period, for example, during the fall 2015 selloff following the Brexit vote:

```
interesting_times = extract_interesting_date_ranges(returns=returns)
interesting_times['Fall2015'].to_frame('pf') \
 .join(benchmark_rets) \
 .add(1).cumprod().sub(1) \
 .plot(lw=2, figsize=(14, 6), title='Post-Brexit Turmoil')
```

The resulting plot looks as follows:



How to avoid the pitfalls of backtesting

Backtesting simulates an algorithmic strategy using historical data with the goal of identifying patterns that generalize to new market conditions. In addition to the generic challenges of predicting an uncertain future in changing markets, numerous factors make mistaking positive in-sample performance for the discovery of true patterns very likely. These factors include aspects of the data, the implementation of the strategy simulation, and flaws with the statistical tests and their interpretation. The risks of false discoveries multiply with the use of more computing power, bigger datasets, and more complex algorithms that facilitate the identification of apparent patterns in the noise.

We will list the most serious and common methodological mistakes and refer to the literature on multiple testing for further detail. We will also introduce the deflated SR that illustrates how to adjust metrics that result from repeated trials when using the same set of financial data for your analysis.

Data challenges

Challenges to the backtest validity due to data issues include look-ahead bias, survivorship bias, and outlier control.

Look-ahead bias

Tests of trading rules derived from past data will yield biased results when the sample data used to develop the rules contains information that was not, in fact, available or known at the point in time the data refers to.

A typical source of this bias is the failure to account for the common ex-post corrections of reported financials. Stock splits or reverse splits can also generate look-ahead bias. When computing the earnings yield, earnings-per-share data comes from company financials with low frequency, while market prices are available at least daily. Hence, both EPS and price data need to be adjusted for splits at the same time.

The solution lies in the careful analysis of the timestamps associated with all data that enters a backtest to ensure that only point-in-time data is used. High-quality data providers, such as Compustat, ensure that these criteria are met. When point-in-time data is not available, assumptions about the lag in reporting needs to be made.

Survivorship bias

Survivorship bias emerges when a backtest is conducted on data that only contains currently active securities and omits assets that have disappeared over time, for example, due to bankruptcy, delisting, or acquisition. Securities that are no longer part of the investment universe often did not perform well, and including these cases can positively skew the backtest result.

The solution, naturally, is to verify that datasets include all securities available over time as opposed to only those that are still available when running the test.

Outlier control

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

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

Unrepresentative period

A backtest will not yield a representative result that generalizes to future periods if the time period used does not reflect the current environment well, lacks relevant market regime aspects, and does not include enough data points or captures extreme historical events that are unlikely to repeat.

The solution involves using sample periods that include important market phenomena, or generate synthetic data that reflect relevant market characteristics (see the *Resources* section for guidance on implementation).

Implementation issues

Practical issues related to the implementation of the historical simulation include failure to mark to market, i.e. accurately reflect underlying market prices and account for drawdowns, unrealistic assumptions about the availability, cost, or market impact of trades, or the timing of signals and trade execution.

Mark-to-market performance

This strategy may perform well over the course of the backtest but lead to unacceptable losses or volatility over time.

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

Trading costs

This strategy may assume short sales that require a counter-party, hold less liquid assets that may move the market when traded or underestimate the costs that arise due to broker fees or slippage, which is the difference between the market price at the decision to trade and subsequent execution.

The solution includes a limitation to a highly liquid universe and realistic parameter assumptions for trading and slippage costs (as illustrated in the preceding `zipline` example). This also safeguards against the inclusion of unstable factor signals with a high decay and, hence, turnover.

Timing of trades

The simulation could make unrealistic assumptions about the timing of the evaluation of the alpha factor signals and the resulting trades. For instance, signals may be evaluated at close prices when the next trade is only available at the often-quite-different open prices. As a consequence, the backtest will be significantly biased when the close price is used to evaluate trading performance.

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

Data-snooping and backtest-overfitting

The most prominent challenge to backtest validity, including to published results, relates to the discovery of spurious patterns due to multiple testing during the strategy-selection process. Selecting a strategy after testing different candidates on the same data will likely bias the choice because a positive outcome is more likely to be due to the stochastic nature of the performance measure itself. In other words, the strategy is overly tailored, or overfit, to the data at hand and produces deceptively positive results.

Hence, backtest performance is not informative unless the number of trials is reported to allow for an assessment of the risk of selection bias. This is rarely the case in practical or academic research, inviting doubts about the validity of many published claims.

The risk of overfitting a backtest to a particular dataset does not only arise from directly running numerous tests but includes strategies designed based on prior knowledge of what works and doesn't, that is, knowledge of different backtests run by others on the same data. As a result, backtest-overfitting is hard to avoid in practice.

Solutions include selecting tests to undertake based on investment or economic theory rather than broad data-mining efforts. It also implies testing in a variety of contexts and scenarios, including possibly on synthetic data.

The minimum backtest length and the deflated SR

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

Another result includes an estimate of the minimum length of the backtest that an investor should require given the number of trials attempted, to avoid selecting a strategy with a given in-sample SR during a given number of trials that has an expected out-of-sample SR of zero. This implies that, e.g., if only two years of daily backtest data is available no more than seven strategy variations should be tried, and if only five years of daily backtest data is available, no more than 45 strategy variations should be tried. See references for implementation details.

De Lopez Prado and Bailey (2014) also derive a deflated SR to compute the probability that the SR is statistically significant while controlling for the inflationary effect of multiple testing, non-normal returns, and shorter sample lengths (see the `03_multiple_testing` subdirectory for the Python implementation of `deflated_sharpe_ratio.py` and references for the derivation of the related formulas).

Optimal stopping for backtests

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

Based on the solution to the *secretary problem* from optimal stopping theory, the recommendation is to decide according to the following rule of thumb: test a random sample of 1/e (roughly 37%) of reasonable strategies and record their performance. Then, continue tests until a strategy outperforms those tested before.

This rule applies to tests of several alternatives with the goal to choose a near-best as soon as possible while minimizing the risk of a false positive.

How to manage portfolio risk and return

Portfolio management aims to take positions in financial instruments that achieve the desired risk-return trade-off regarding a benchmark. In each period, a manager selects positions that optimize diversification to reduce risks while achieving a target return. Across periods, the positions will be rebalanced to account for changes in weights resulting from price movements to achieve or maintain a target risk profile.

Diversification permits us to reduce risks for a given expected return by exploiting how price movements interact with each other as one asset's gains can make up for another asset's losses. Harry Markowitz invented **Modern Portfolio Theory** (**MPT**) in 1952 and provided the mathematical tools to optimize diversification by choosing appropriate portfolio weights. Markowitz showed how portfolio risk, measured as the standard deviation of portfolio returns, depends on the covariance among the returns of all assets and their relative weights. This relationship implies the existence of an efficient frontier of portfolios that maximize portfolio returns given a maximal level of portfolio risk.

However, mean-variance frontiers are highly sensitive to the estimates of the input required for their calculation, such as expected returns, volatilities, and correlations. In practice, mean-variance portfolios that constrain these input to reduce sampling errors have performed much better. These constrained special cases include equal-weighted, minimum-variance, and risk-parity portfolios.

The Capital Asset Pricing Model (CAPM) is an asset valuation model that builds on the MPT risk-return relationship. It introduces the concept of a risk premium that an investor can expect in market equilibrium for holding a risky asset; the premium compensates for the time value of money and the exposure to overall market risk that cannot be eliminated through diversification (as opposed to the idiosyncratic risk of specific assets). The economic rationale for non-diversifiable risk is, for example, macro drivers of the business risks affecting equity returns or bond defaults. Hence, an asset's expected return, $E[r_i]$, is the sum of the risk-free interest rate, r_f, and a risk premium proportional to the asset's exposure to the expected excess return of the market portfolio, r_m, over the risk-free rate:

$$E[r_i] = \alpha_i + r_f + \beta_i\,(E[r_m] - r_f)$$

In theory, the market portfolio contains all investable assets and will be held by all rational investors in equilibrium. In practice, a broad value-weighted index approximates the market, for example, the S&P 500 for US equity investments. β_i measures the exposure to the excess returns of the market portfolio. If the CAPM is valid, the intercept component, α_i, should be zero. In reality, the CAPM assumptions are often not met, and alpha captures the returns left unexplained by exposure to the broad market.

Over time, research uncovered non-traditional sources of risk premiums, such as the momentum or the equity value effects that explained some of the original alpha. Economic rationales, such as behavioral biases of under or overreaction by investors to new information justify risk premiums for exposure to these alternative risk factors. They evolved into investment styles designed to capture these alternative betas that also became tradable in the form of specialized index funds. After isolating contributions from these alternative risk premiums, true alpha becomes limited to idiosyncratic asset returns and the manager's ability to time risk exposures.

The EMH has been refined over the past several decades to rectify many of the original shortcomings of the CAPM, including imperfect information and the costs associated with transactions, financing, and agency. Many behavioral biases have the same effect, and some frictions are modeled as behavioral biases.

ML plays an important role in deriving new alpha factors using supervised and unsupervised learning techniques based on the market, fundamental, and alternative data sources discussed in the previous chapters. The inputs to a machine learning model consist of both raw data and features engineered to capture informative signals. ML models are also used to combine individual predictive signals and deliver higher-aggregate predictive power.

Modern portfolio theory and practice have evolved significantly over the last several decades. We will introduce:

- Mean-variance optimization, and its shortcomings
- Alternatives such as minimum-risk and 1/n allocation
- Risk parity approaches
- Risk factor approaches

Mean-variance optimization

MPT solves for the optimal portfolio weights to minimize volatility for a given expected return, or maximize returns for a given level of volatility. The key requisite input are expected asset returns, standard deviations, and the covariance matrix.

How it works

Diversification works because the variance of portfolio returns depends on the covariance of the assets and can be reduced below the weighted average of the asset variances by including assets with less than perfect correlation. In particular, given a vector, ω, of portfolio weights and the covariance matrix, Σ, the portfolio variance, σ_{PF}, is defined as:

$$\sigma_{\text{PF}} = \omega^T \Sigma \omega$$

Markowitz showed that the problem of maximizing the expected portfolio return subject to a target risk has an equivalent dual representation of minimizing portfolio risk subject to a target expected return level, $\mu_{PF.}$ Hence, the optimization problem becomes:

$$\begin{aligned} \min_{\omega} \quad & \sigma^2_{\text{PF}} = \omega^T \Sigma \omega \\ \text{s.t.} \quad & \mu_{\text{PF}} = \omega^T \mu \\ & \|\omega\| = 1 \end{aligned}$$

The efficient frontier in Python

We can calculate an efficient frontier using `scipy.optimize.minimize` and the historical estimates for asset returns, standard deviations, and the covariance matrix. The code can be found in the `efficient_frontier` subfolder of the repo for this chapter and implements the following sequence of steps:

1. The simulation generates random weights using the Dirichlet distribution, and computes the mean, standard deviation, and SR for each sample portfolio using the historical return data:

```
def simulate_portfolios(mean_ret, cov, rf_rate=rf_rate,
short=True):
    alpha = np.full(shape=n_assets, fill_value=.01)
    weights = dirichlet(alpha=alpha, size=NUM_PF)
    weights *= choice([-1, 1], size=weights.shape)

    returns = weights @ mean_ret.values + 1
    returns = returns ** periods_per_year - 1
    std = (weights @ monthly_returns.T).std(1)
    std *= np.sqrt(periods_per_year)
    sharpe = (returns - rf_rate) / std

    return pd.DataFrame({'Annualized Standard Deviation': std,
```

```
                                'Annualized Returns': returns,
                                'Sharpe Ratio': sharpe}), weights
```

2. Set up the quadratic optimization problem to solve for the minimum standard deviation for a given return or the maximum SR. To this end, define the functions that measure the key metrics:

```
def portfolio_std(wt, rt=None, cov=None):
    """Annualized PF standard deviation"""
    return np.sqrt(wt @ cov @ wt * periods_per_year)

def portfolio_returns(wt, rt=None, cov=None):
    """Annualized PF returns"""
    return (wt @ rt + 1) ** periods_per_year - 1

def portfolio_performance(wt, rt, cov):
    """Annualized PF returns & standard deviation"""
    r = portfolio_returns(wt, rt=rt)
    sd = portfolio_std(wt, cov=cov)
    return r, sd
```

3. Define a target function that represents the negative SR for `scipy`'s `minimize` function to optimize given the constraints that the weights are bounded by, [-1, 1], and sum to one in absolute terms:

```
def neg_sharpe_ratio(weights, mean_ret, cov):
    r, sd = portfolio_performance(weights, mean_ret, cov)
    return -(r - rf_rate) / sd

weight_constraint = {'type': 'eq',
                     'fun': lambda x: np.sum(np.abs(x)) - 1}

def max_sharpe_ratio(mean_ret, cov, short=True):
    return minimize(fun=neg_sharpe_ratio,
                    x0=x0,
                    args=(mean_ret, cov),
                    method='SLSQP',
                    bounds=((-1 if short else 0, 1),) * n_assets,
                    constraints=weight_constraint,
                    options={'tol':1e-10, 'maxiter':1e4})
```

4. Compute the efficient frontier by iterating over a range of target returns and solving for the corresponding minimum variance portfolios. The optimization problem and the constraints on portfolio risk and return as a function of the weights can be formulated as follows:

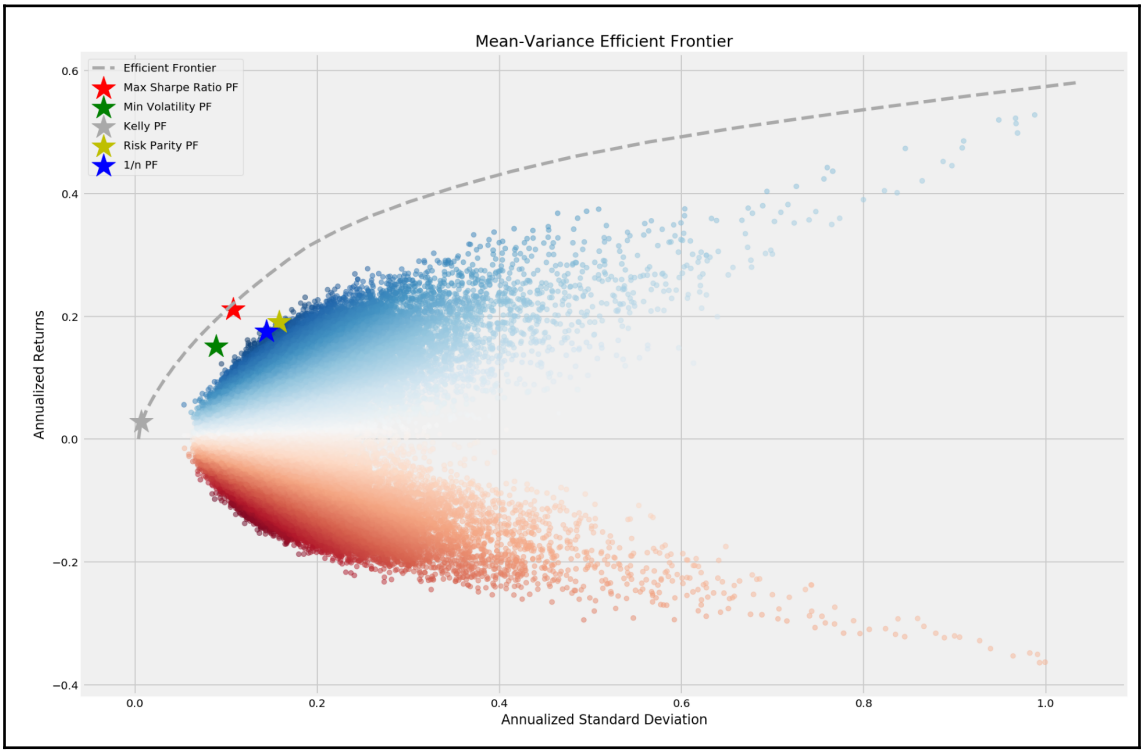
```
def neg_sharpe_ratio(weights, mean_ret, cov):
```

```
    r, sd = pf_performance(weights, mean_ret, cov)
    return -(r - RF_RATE) / sd

def pf_volatility(w, r, c):
    return pf_performance(w, r, c)[1]

def efficient_return(mean_ret, cov, target):
    args = (mean_ret, cov)
    def ret_(weights):
        return pf_ret(weights, mean_ret)

    constraints = [{'type': 'eq', 'fun': lambda x: ret_(x) -
                    target},
                   {'type': 'eq', 'fun': lambda x: np.sum(x) - 1}]
    bounds = ((0.0, 1.0),) * n_assets
    return minimize(pf_volatility,
                    x0=x0,
                    args=args, method='SLSQP',
                    bounds=bounds,
                    constraints=constraints)
```

5. The solution requires iterating over ranges of acceptable values to identify optimal risk-return combinations:

```
def min_vol_target(mean_ret, cov, target, short=True):

    def ret_(wt):
        return portfolio_returns(wt, mean_ret)

    constraints = [{'type': 'eq', 'fun': lambda x: ret_(x) -
target},
                    weight_constraint]

    bounds = ((-1 if short else 0, 1),) * n_assets
    return minimize(portfolio_std, x0=x0, args=(mean_ret, cov),
                    method='SLSQP', bounds=bounds,
                    constraints=constraints,
                    options={'tol': 1e-10, 'maxiter': 1e4})

def efficient_frontier(mean_ret, cov, ret_range):
    return [min_vol_target(mean_ret, cov, ret) for ret in
ret_range]
```

The simulation yields a subset of the feasible portfolios, and the efficient frontier identifies the optimal in-sample return-risk combinations that were achievable given historic data. The below figure shows the result including the minimum variance portfolio and the portfolio that maximizes the SR and several portfolios produce by alternative optimization strategies that we discuss in the following sections.



The portfolio optimization can be run at every evaluation step of the trading strategy to optimize the positions.

Challenges and shortcomings

The preceding mean-variance frontier example illustrates the in-sample, backward-looking optimization. In practice, portfolio optimization requires forward-looking input. Expected returns are notoriously difficult to estimate accurately.

The covariance matrix can be estimated somewhat more reliably, which has given rise to several alternative approaches. However, covariance matrices with correlated assets pose computational challenges since the optimization problem requires inverting the matrix. The high condition number induces numerical instability, which in turn gives rise to Markovitz curse: the more diversification is required (by correlated investment opportunities), the more unreliable the weights produced by the algorithm.

Many investors prefer to use portfolio-optimization techniques with less onerous input requirements. We now introduce several alternatives that aim to address these shortcomings, including more recent approaches based on machine learning.

Alternatives to mean-variance optimization

The challenges with accurate input for the mean-variance optimization problem have led to the adoption of several practical alternatives that constrain the mean, the variance, or both, or omit return estimates that are more challenging, such as the risk parity approach.

The 1/n portfolio

Simple portfolios provide useful benchmarks to gauge the added value of complex models that generate the risk of overfitting. The simplest strategy—an equally-weighted portfolio—has been shown to be one of the best performers.

Famously, de Miguel, Garlappi, and Uppal (2009) compared the out-of-sample performance of portfolios produced by various mean-variance optimizers, including robust Bayesian estimators, portfolio constraints, and optimal combinations of portfolios, to the simple 1/N rule. They found that the 1/N portfolio produced a higher Sharpe ratio than each asset class position, explained by the high cost of estimation errors that often outweighs the benefits of sophisticated optimization out-of-sample.

The 1/n portfolio is also included in the efficient frontier figure above.

The minimum-variance portfolio

Another alternative is the **global minimum variance** (**GMV**) portfolio, which prioritizes the minimization of risk. It is shown in the efficient frontier figure and can be calculated as follows by minimizing the portfolio standard deviation using the mean-variance framework:

```
def min_vol(mean_ret, cov, short=True):
    return minimize(fun=portfolio_std,
                    x0=x0,
                    args=(mean_ret, cov),
                    method='SLSQP',
                    bounds=bounds = ((-1 if short else 0, 1),) *
                          n_assets,
                          constraints=weight_constraint,
                          options={'tol': 1e-10, 'maxiter': 1e4})
```

The corresponding `min.` volatility portfolio lies on the efficient frontier as shown above.

Global Portfolio Optimization - The Black-Litterman approach

The Global Portfolio Optimization approach of Black and Litterman (1992) combines economic models with statistical learning and is popular because it generates estimates of expected returns that are plausible in many situations.

The technique departs from the assumption that the market is a mean-variance portfolio implied by the CAPM equilibrium model, and builds on the fact that the observed market capitalization can be considered as optimal weights assigned by the market. Market weights reflect market prices that, in turn, embody the market's expectations of future returns.

Hence, the approach can reverse-engineer the unobservable future expected returns from the assumption that the market is close enough to equilibrium, as defined by the CAPM, and allow investors to adjust these estimates to their own beliefs using a shrinkage estimator. The model can be interpreted as a Bayesian approach to portfolio optimization. We will introduce Bayesian methods in `Chapter 9`, *Bayesian Machine Learning*.

How to size your bets – the Kelly rule

The Kelly rule has a long history in gambling because it provides guidance on how much to stake on each of an (infinite) sequence of bets with varying (but favorable) odds to maximize terminal wealth. It was published as A New Interpretation of the Information Rate in 1956 by John Kelly who was a colleague of Claude Shannon's at Bell Labs. He was intrigued by bets placed on candidates at the new quiz show The $64,000 Question, where a viewer on the west coast used the three-hour delay to obtain insider information about the winners.

Kelly drew a connection to Shannon's information theory to solve for the bet that is optimal for long-term capital growth when the odds are favorable, but uncertainty remains. His rule maximizes logarithmic wealth as a function of the odds of success of each game, and includes implicit bankruptcy protection since log(0) is negative infinity so that a Kelly gambler would naturally avoid losing everything.

The optimal size of a bet

Kelly began by analyzing games with a binary win-lose outcome. The key variables are:

- **b**: The odds define the amount won for a $1 bet. Odds = 5/1 implies a $5 gain if the bet wins, plus recovery of the $1 capital.
- **p**: The probability defines the likelihood of a favorable outcome.
- **f**: The share of the current capital to bet.
- **V**: The value of the capital as a result of betting.

The Kelly rule aims to maximize the value's growth rate, *G*, of infinitely-repeated bets:

$$G = \lim_{N \to \infty} \frac{1}{N} \log \frac{V_N}{V_0}$$

When *W* and *L* are the numbers of wins and losses, then:

$$\begin{aligned} V_N &= (1 + b * f)^W (1 - f)^L V_0 && \Rightarrow \\ G &= \lim_{N \to \infty} \left[\frac{W}{N} \log(1 + \text{odds} * \text{share}) + \frac{L}{N} \log(1 - f) \right] && \Leftrightarrow \\ &= p \log(1 + b * f) + (1 - p) \log(1 - f) \end{aligned}$$

We can maximize the rate of growth *G* by maximizing *G* with respect to *f*, as illustrated using `sympy` as follows:

```
from sympy import symbols, solve, log, diff

share, odds, probability = symbols('share odds probability')
Value = probability * log(1 + odds * share) + (1 - probability) * log(1
        - share)
solve(diff(Value, share), share)

[(odds*probability + probability - 1)/odds]
```

We arrive at the optimal share of capital to bet:

$$\text{Kelly Criterion:} \quad f^* = \frac{b * p + p - 1}{b}$$

Optimal investment – single asset

In a financial market context, both outcomes and alternatives are more complex, but the Kelly rule logic does still apply. It was made popular by Ed Thorp, who first applied it profitably to gambling (described in Beat the Dealer) and later started the successful hedge fund Princeton/Newport Partners.

With continuous outcomes, the growth rate of capital is defined by an integrate over the probability distribution of the different returns that can be optimized numerically:

$$E[G] = \int \log(1 * fr) P(r) dr \Leftrightarrow$$
$$\frac{d}{df} E[G] = \int_{-\infty}^{+\infty} \frac{r}{1 * fr} P(r) dr = 0$$

We can solve this expression for the optimal f^* using the `scipy.optimize` module:

```
def norm_integral(f, m, st):
    val, er = quad(lambda s: np.log(1+f*s)*norm.pdf(s, m, st), m-3*st,
                   m+3*st)
    return -val


def norm_dev_integral(f, m, st):
    val, er = quad(lambda s: (s/(1+f*s))*norm.pdf(s, m, st), m-3*st,
                   m+3*st)
    return val
```

```
m = .058
s = .216
# Option 1: minimize the expectation integral
sol = minimize_scalar(norm_integral, args=(
                m, s), bounds=[0., 2.], method='bounded')
print('Optimal Kelly fraction: {:.4f}'.format(sol.x))
```

Optimal investment – multiple assets

We will use an example with various equities. E. Chan (2008) illustrates how to arrive at a multi-asset application of the Kelly Rule, and that the result is equivalent to the (potentially levered) maximum Sharpe ratio portfolio from the mean-variance optimization.

The computation involves the dot product of the precision matrix, which is the inverse of the covariance matrix, and the return matrix:

```
mean_returns = monthly_returns.mean()
cov_matrix = monthly_returns.cov()
precision_matrix = pd.DataFrame(inv(cov_matrix), index=stocks,
columns=stocks)
kelly_wt = precision_matrix.dot(mean_returns).values
```

The Kelly Portfolio is also shown in the efficient frontier diagram (after normalization so that the absolute weights sum to one). Many investors prefer to reduce the Kelly weights to reduce the strategy's volatility, and Half-Kelly has become particularly popular.

Risk parity

The fact that the previous 15 years have been characterized by two major crises in the global equity markets, a consistently upwardly-sloping yield curve, and a general decline in interest rates made risk parity look like a particularly compelling option. Many institutions carved out strategic allocations to risk parity to further diversify their portfolios.

A simple implementation of risk parity allocates assets according to the inverse of their variances, ignoring correlations and, in particular, return forecasts:

```
var = monthly_returns.var()
risk_parity_weights = var / var.sum()
```

The risk parity portfolio is also shown in the efficient frontier diagram at the beginning of this section.

Risk factor investment

An alternative framework for estimating input is to work down to the underlying determinants, or factors, that drive the risk and returns of assets. If we understand how the factors influence returns, and we understand the factors, we will be able to construct more robust portfolios.

The concept of factor investing looks beyond asset class labels to the underlying factor risks to maximize the benefits of diversification. Rather than distinguishing investment vehicles by labels such as hedge funds or private equity, factor investing aims to identify distinct risk-return profiles based on differences in exposure to fundamental risk factors. The naïve approach to mean-variance investing plugs (artificial) groupings as distinct asset classes into a mean-variance optimizer. Factor investing recognizes that such groupings share many of the same factor risks as traditional asset classes. Diversification benefits can be overstated, as investors discovered during the last crisis when correlations among risky asset classes increased due to exposure to the same underlying factor risks.

Hierarchical risk parity

Mean-variance optimization is very sensitive to the estimates of expected returns and the covariance of these returns. The covariance matrix inversion also becomes more challenging and less accurate when returns are highly correlated, as is often the case in practice. The result has been called the Markowitz curse: when diversification is more important because investments are correlated, conventional portfolio optimizers will likely produce an unstable solution. The benefits of diversification can be more than offset by mistaken estimates. As discussed, even naive, equally-weighted portfolios can beat mean-variance and risk-based optimization out of sample.

More robust approaches have incorporated additional constraints (Clarke et al., 2002), Bayesian priors (Black and Litterman, 1992), or used shrinkage estimators to make the precision matrix more numerically stable (Ledoit and Wolf [2003], available in scikit-learn (`http://scikit-learn.org/stable/modules/generated/sklearn.covariance.LedoitWolf.html`). **Hierarchical risk parity** (**HRP**), in contrast, leverages unsupervised machine learning to achieve superior out-of-sample portfolio allocations.

A recent innovation in portfolio optimization leverages graph theory and hierarchical clustering to construct a portfolio in three steps (Lopez de Prado, 2015):

1. Define a distance metric so that correlated assets are close to each other, and apply single-linkage clustering to identify hierarchical relationships

2. Use the hierarchical correlation structure to quasi-diagonalize the covariance matrix.
3. Apply top-down inverse-variance weighting using a recursive bisectional search to treat clustered assets as complements rather than substitutes in portfolio construction and to reduce the number of degrees of freedom.

A related method to construct **hierarchical clustering portfolios** (**HCP**) was presented by Raffinot (2016). Conceptually, complex systems such as financial markets tend to have a structure and are often organized in a hierarchical way, while the interaction among elements in the hierarchy shapes the dynamics of the system. Correlation matrices also lack the notion of hierarchy, which allows weights to vary freely and in potentially unintended ways.

Both HRP and HCP have been tested by JPM on various equity universes. The HRP, in particular, produced equal or superior risk-adjusted returns and Sharpe ratios compared to naive diversification, the maximum-diversified portfolios, or GMV portfolios.

We will present the Python implementation in `Chapter 12`, *Unsupervised Learning*.

Summary

In this chapter, we covered the important topic of portfolio management, which involves the combination of investment positions with the objective of managing risk-return trade-offs. We introduced `pyfolio` to compute and visualize key risk and return metrics and to compare the performance of various algorithms.

We saw how important accurate predictions are to optimize portfolio weights and maximize diversification benefits. We also explored how ML can facilitate more effective portfolio construction by learning hierarchical relationships from the asset-returns covariance matrix.

We will now move on to the second part of this book, which focuses on the use of ML models. These models will produce more accurate predictions by making more effective use of more diverse information to capture more complex patterns than the simpler alpha factors that were most prominent so far.

We will begin by training, testing, and tuning linear models for regression and classification using cross-validation to achieve robust out-of-sample performance. We will also embed these models within the framework of defining and backtesting algorithmic trading strategies, which we covered in the last two chapters.