5
Portfolio Optimization and Performance Evaluation

Alpha factors generate signals that an algorithmic strategy translates into trades, which, in turn, produce long and short positions. The returns and risk of the resulting portfolio determine the success of the strategy.

To test a strategy prior to implementation under market conditions, we need to simulate the trades that the algorithm would make and verify their performance. 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. The goal is to avoid false discoveries from tailoring a strategy to specific past circumstances.

In a portfolio context, positive asset returns can offset negative price movements. Positive price changes for one asset are more likely to offset losses on another, the lower the correlation between the two positions is. Based on how portfolio risk depends on the positions' covariance, Harry Markowitz developed the theory behind modern portfolio management based on diversification in 1952. The result is mean-variance optimization, which selects weights for a given set of assets to minimize risk, measured as the standard deviation of returns for a given expected return.

The **capital asset pricing model** (**CAPM**) introduces a risk premium, measured as the expected return in excess of a risk-free investment, as an equilibrium reward for holding an asset. This reward compensates for the exposure to a single risk factor—the market—that is systematic as opposed to idiosyncratic to the asset and thus cannot be diversified away.

Risk management has evolved to become more sophisticated as additional risk factors and more granular choices for exposure have emerged. The Kelly criterion is a popular approach to dynamic portfolio optimization, which is the choice of a sequence of positions over time; it was 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, including the application of **machine learning** (**ML**) to learn hierarchical relationships among assets, and to 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 measure portfolio risk and return
- Managing portfolio weights using mean-variance optimization and alternatives
- Using machine learning to optimize asset allocation in a portfolio context
- Simulating trades and create a portfolio based on alpha factors using Zipline
- How to evaluate portfolio performance using pyfolio

> You can find the code samples for this chapter and links to additional resources in the corresponding directory of the GitHub repository. The notebooks include color versions of the images..

How to measure portfolio performance

To evaluate and compare different strategies or to improve an existing strategy, we need metrics that reflect their performance with respect to our objectives. In investment and trading, the most common objectives are the return and the risk of the investment portfolio.

Typically, these metrics are compared to a benchmark that represents alternative investment opportunities, such as a summary of the investment universe like the S&P 500 for US equities or the risk-free interest rate for fixed income assets.

There are several metrics to evaluate these objectives. In this section, we will review the most common measures for comparing portfolio results. These measures will be useful when we look at different approaches to optimize portfolio performance, simulate the interaction of a strategy with the market using Zipline, and compute relevant performance metrics using the pyfolio library in later sections.

We'll use some simple notation: 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.

Capturing risk-return trade-offs in a single number

The return and risk objectives imply a trade-off: taking more risk may yield higher returns in some circumstances, but also implies greater downside. To compare how different strategies navigate this trade-off, ratios that compute a measure of return per unit of risk are very popular. We'll discuss the Sharpe ratio and the information ratio in turn.

The Sharpe ratio

The ex ante **Sharpe ratio** (**SR**) compares the portfolio's expected excess return 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:

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

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

$$\hat{\mu}_{R^e} = \frac{1}{T}\sum_{t=1}^{T} r_t^e$$

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

$$\text{SR} \equiv \frac{\hat{\mu}_{R^e} - R_f}{\hat{\sigma}^2_{R^e}}$$

Unless the risk-free rate is volatile (as in emerging markets), the standard deviation of excess and raw returns will be similar.

For **independently and identically distributed (IID)** returns, the distribution of the SR estimator for tests of statistical significance follows from the application of the **central limit theorem (CLT)**, according to large-sample statistical theory, to $\hat{\mu}$ and $\hat{\sigma}^2$. The CLT implies that sums of IID random variables like $\hat{\mu}$ and $\hat{\sigma}^2$ converge to the normal distribution.

When you need to compare SR for different frequencies, say for monthly and annual data, you can multiply the higher frequency SR by the square root of the number of the corresponding period contained in the lower frequency. To convert a monthly SR into an annual SR, multiply by $\sqrt{12}$, and from daily to monthly multiply by $\sqrt{12}$.

However, financial returns often violate the IID assumption. Andrew Lo has derived the necessary adjustments to the distribution and the time aggregation for returns that are stationary but autocorrelated. 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 information ratio

The **information ratio (IR)** is similar to the Sharpe ratio but uses a benchmark rather than the risk-free rate. The benchmark is usually cho-

sen to represent the available investment universe such as the S&P 500 for a portfolio on large-cap US equities.

Hence, the IR 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, that is:

$$\text{IR} = \frac{\text{Alpha}}{\text{Tracking Error}}$$

The IR has also been used to explain how excess returns depend on a manager's skill and the nature of her strategy, as we will see next.

The fundamental law of active management

> *"Diversification is protection against ignorance. It makes little sense if you know what you are doing."*
>
> *– Warren Buffet*

It's a curious fact that **Renaissance Technologies** (**RenTec**), the top-performing quant fund founded by Jim Simons, which we mentioned in *Chapter 1*, *Machine Learning for Trading – From Idea to Execution*, has produced similar returns as Warren Buffet, despite extremely different approaches. Warren Buffet's investment firm Berkshire Hathaway holds some 100-150 stocks for fairly long periods, whereas RenTec may execute 100,000 trades per day. How can we compare these distinct strategies?

A high IR reflects an attractive out-performance of the benchmark relative to the additional risk taken. The **Fundamental Law of Active Management** explains how such a result can be achieved: it approximates the IR as the product of the **information coefficient** (**IC**) and the breadth of the strategy.

As discussed in the previous chapter, the IC measures the rank correlation between return forecasts, like those implied by an alpha factor, and the actual forward returns. Hence, it is a measure of the forecasting skill of the manager. The breadth of the strategy is measured by the independent number of bets (that is, trades) an investor makes in a given time period, and thus represents the ability to apply the forecasting skills.

The Fundamental Law states that the IR, also known as the **appraisal risk** (Treynor and Black), is the product of both values. In other words, it

summarizes the importance to play both often (high breadth) and to play well (high IC):

$$IR \sim IC * \sqrt{breadth}$$

This framework has been extended to include the **transfer coefficient (TC)** to reflect portfolio constraints as an additional factor (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: if there are no constraints, the TC would simply equal one; but if the manager does not short stocks even though forecasts suggests they should, the TC will be less than one and reduce the IC (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, as illustrated by the following simulation:

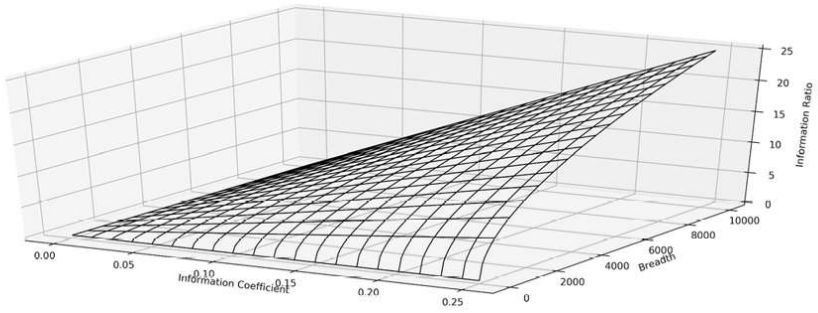

Figure 5.1: Information ratios for different values of breadth and information coefficient

In practice, estimating the breadth of a strategy is difficult, given the cross-sectional and time-series correlation among forecasts. You should view the Fundamental Law and its extensions as a useful analytical framework for thinking about how to improve your risk-adjusted portfolio performance. We'll look at techniques for doing so in practice next.

How to manage portfolio risk and return

Portfolio management aims to pick and size positions in financial instruments that achieve the desired risk-return trade-off regarding a benchmark. As a portfolio manager, in each period, you select positions that optimize diversification to reduce risks while achieving a target return. Across periods, these positions may require rebalancing to account for changes in weights resulting from price movements to achieve or maintain a target risk profile.

The evolution of modern portfolio management

Diversification permits us to reduce risks for a given expected return by exploiting how imperfect correlation allows for one asset's gains to 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 maximizes portfolio returns given a maximal level of portfolio risk.

However, mean-variance frontiers are highly sensitive to the estimates of the inputs required for their calculation, namely expected returns, volatilities, and correlations. In practice, mean-variance portfolios that constrain these inputs 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** includes, for example, macro drivers of the business risks affecting all 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:

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

In theory, the **market portfolio** contains all investable assets and, in equilibrium, will be held by all rational investors. 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 of asset, *i*, 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.

As discussed in the previous chapter, 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.

These factors evolved into investment styles designed to capture these **alternative betas** that became tradable in the form of specialized index funds. Similarly, risk management now aims to control the exposure of numerous sources of risk beyond the market portfolio.

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 **efficient market hypothesis** (**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.

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

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

Mean-variance optimization

Modern portfolio theory 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 inputs 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. It 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, μ_{PF}. Hence, the optimization problem becomes:

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

Finding 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. SciPy 's `minimize` function implements a range of constrained and unconstrained optimization algorithms for scalar functions that output a single number from one or more input variables (see the SciPy documentation for more details). The code can be found in the `strategy_evaluation` subfolder of the repository for this chapter and implements the following sequence of steps:

First, the simulation generates random weights using the Dirichlet distribution and computes the mean, standard deviation, and SR for each sam-

ple 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=.05)
    weights = dirichlet(alpha=alpha, size=NUM_PF)
    if short:
        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
```

Next, we set up the quadratic optimization problem to solve for the minimum standard deviation for a given return or the maximum SR. To this end, we define the functions that measure the key performance 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
```

Next, we 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, [0, 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=False):
    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})
```

Then, we compute the efficient frontier by iterating over a range of target returns and solving for the corresponding minimum variance portfolios. To this end, we formulate the optimization problem using the constraints on portfolio risk and return as a function of the weights, as follows:

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

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

```
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.

Figure 5.2 shows the result, including the minimum variance portfolio, the portfolio that maximizes the SR, and several portfolios produced by alternative optimization strategies that we'll discuss in the following sections:

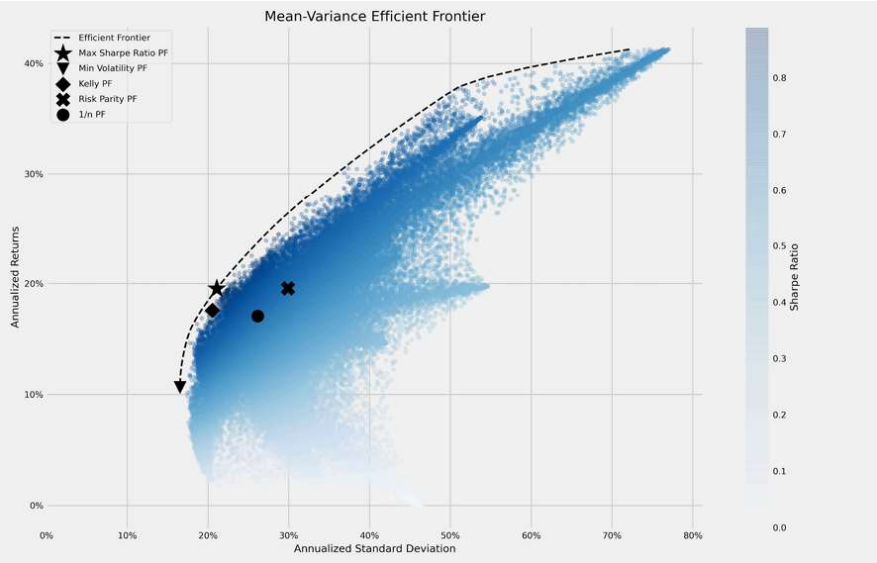

Figure 5.2: The efficient frontier and different optimized portfolios

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 estimation illustrates **in-sample**, that is, **backward-looking** optimization. In practice, portfolio optimization requires forward-looking inputs and outputs. However, expected returns are notoriously difficult to estimate accurately. It is best viewed as a starting point and benchmark for numerous improvements.

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 the **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 will now introduce several alternatives that aim to address these shortcomings, including a more recent approach based on machine learning.

Alternatives to mean-variance optimization

The challenges with accurate inputs 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, which we'll discuss later in this section.

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 the alternatives on various datasets, explained by the high cost of estimation errors that often outweighs the benefits of sophisticated optimization out of sample.

More specifically, they found that the estimation window required for the sample-based mean-variance strategy and its extensions to outperform the 1/N benchmark is around 3,000 months for a portfolio with 25 assets and about 6,000 months for a portfolio with 50 assets.

The 1/N portfolio is also included in *Figure 5.2* in the previous section.

The minimum-variance portfolio

Another alternative is the **global minimum-variance** (**GMV**) portfolio, which prioritizes the minimization of risk. It is shown in *Figure 5.2* and can be calculated, as follows, by minimizing the portfolio standard deviation using the mean-variance framework:

```
def min_vol(mean_ret, cov, short=False):
    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 minimum volatility portfolio lies on the efficient frontier, as shown previously in *Figure 5.2*.

Global Portfolio Optimization – the Black-Litterman approach

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

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

The approach can thus reverse-engineer the unobservable future expected returns from the assumption that the market is close enough to equilibrium, as defined by the CAPM. Investors can 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 10*, *Bayesian ML – Dynamic Sharpe Ratios and Pairs Trading Strategies*.

How to size your bets – the Kelly criterion

The **Kelly criterion** has a long history in gambling because it provides guidance on how much to stake on each bet in an (infinite) sequence of bets with varying (but favorable) odds to maximize terminal wealth. It was published in a 1956 paper, *A New Interpretation of the Information Rate*, 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 defining 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 defining 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 criterion 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 (you can find this in the `kelly_rule` notebook):

```
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 criterion logic does still apply. It was made popular by Ed Thorp, who first applied it profitably to gambling (described in the

book *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:

$$E[G] = \int \log(1 * fr) P(r) dr \quad \Leftrightarrow$$

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

We can solve this expression for the optimal *f** using the `scipy.optimize` module. The `quad` function computes the value of a definite integral between two values *a* and *b* using FORTRAN's QUADPACK library (hence its name). It returns the value of the integral and an error estimate:

```
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 Kelly fraction: 1.1974
```

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 criterion, 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 previous efficient frontier diagram (after normalization so that the sum of the absolute weights equals 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. It looks to the underlying factor risks that we discussed in the previous chapter on alpha factors 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 (Ang 2014).

The naive approach to mean-variance investing plugs (artificial) groupings as distinct asset classes into a mean-variance optimizer. Factor in-

vesting 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 2008 crisis when correlations among risky asset classes increased due to exposure to the same underlying factor risks.

In *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*, we will show how to measure the exposure of a portfolio to various risk factors so that you can either adjust the positions to tune your factor exposure, or hedge accordingly.

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) or 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 JP Morgan (2012) 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 13*, *Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*.

Trading and managing portfolios with Zipline

In the previous chapter, we introduced Zipline to simulate the computation of alpha factors from trailing market, fundamental, and alternative data for a cross-section of stocks. In this section, we will start acting on the signals emitted by alpha factors. We'll do this by submitting buy and sell orders so we can enter long and short positions or rebalance the portfolio to adjust our holdings to the most recent trade 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. As mentioned in the previous chapter, an in-depth introduction to the testing and evaluation of strategies that include ML models will follow in *Chapter 6*, *The Machine Learning Process*.

Scheduling signal generation and trade execution

We will use the custom `MeanReversion` factor developed in the previous chapter (see the implementation in `01_backtest_with_trades.ipynb`).

The `Pipeline` created by the `compute_factors()` method returns a table with columns containing the 50 longs and shorts. It selects the equities according to the largest negative and positive deviations, respectively, of

their last monthly return from the annual average, normalized by the standard deviation:

```
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))
```

It also limited the universe to the 1,000 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:

```
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'))
```

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

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 the Zipline documentation 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_impa
```

The algorithm continues to execute after calling the `run_algorithm()` function and returns the same backtest performance `DataFrame` that we saw in the previous chapter.

Implementing mean-variance portfolio optimization

We demonstrated in the previous section how to find the efficient frontier using `scipy.optimize`. In this section, we will leverage the PyPortfolioOpt library, which offers portfolio optimization (using SciPy under the hood), including efficient frontier techniques and more recent shrinkage approaches that regularize the covariance matrix (see *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*, on shrinkage for linear regression). The code example lives in `02_backtest_with_pf_optimization.ipynb`.

We'll use the same setup with 50 long and short positions derived from the `MeanReversion` factor ranking. The `rebalance()` function receives the suggested long and short positions and passes each subset on to a new `optimize_weights()` function to obtain dictionaries with `asset: target_percent` pairs:

```
def rebalance(context, data):
    """Compute long, short and obsolete holdings; place 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, positions={asset: 0 for asset in divest})
    # get price history
    prices = data.history(assets, fields='price',
                          bar_count=252+1, # 1 yr of returns
                          frequency='1d')
    if len(longs) > 0:
        long_weights = optimize_weights(prices.loc[:, longs])
        exec_trades(data, positions=long_weights)
    if len(shorts) > 0:
        short_weights = optimize_weights(prices.loc[:, shorts], short=True)
        exec_trades(data, positions=short_weights)
```

The `optimize_weights()` function uses the `EfficientFrontier` object, provided by PyPortfolioOpt, to find the weights that maximize the Sharpe ratio based on the last year of returns and the covariance matrix, both of which the library also computes:

```
def optimize_weights(prices, short=False):
    returns = expected_returns.mean_historical_return(prices=prices,
                                                      frequency=252)
    cov = risk_models.sample_cov(prices=prices, frequency=252)
    # get weights that maximize the Sharpe ratio
    ef = EfficientFrontier(expected_returns=returns,
                           cov_matrix=cov,
                           weight_bounds=(0, 1),
                           gamma=0)

    weights = ef.max_sharpe()
    if short:
        return {asset: -weight for asset, weight in ef.clean_weights().items()}
    else:
        return ef.clean_weights()
```

It returns normalized weights that sum to 1, set to negative values for the short positions.

Figure 5.3 shows that, for this particular set of strategies and time frame, the mean-variance optimized portfolio performs significantly better:

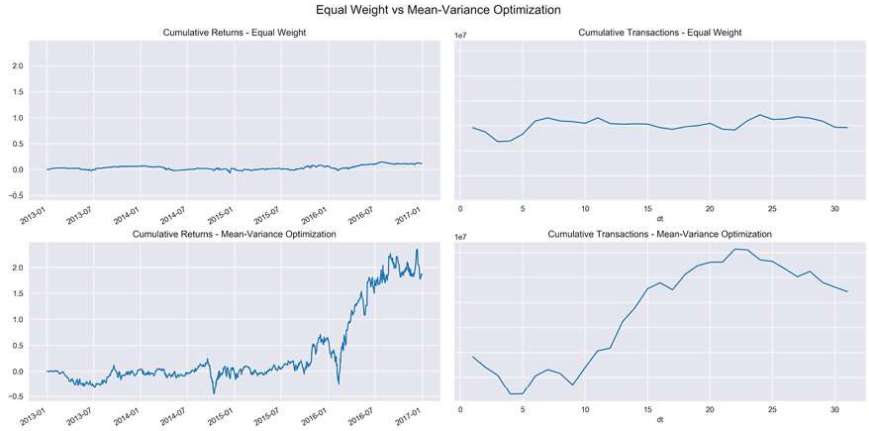

Figure 5.3: Mean-variance vs equal-weighted portfolio performance

PyPortfolioOpt also finds the minimum volatility portfolio. More generally speaking, this example illustrates how you can add logic to tweak portfolio weights using the methods presented in the previous section, or any other of your choosing.

We will now turn to common measures of portfolio return and risk, and how to compute them using the pyfolio library.

Measuring backtest performance with pyfolio

Pyfolio facilitates the analysis of portfolio performance, both in and out of sample using a rich set of metrics and visualizations. It produces tear sheets that cover the analysis of returns, positions, and transactions, as well as event risk during periods of market stress using several built-in scenarios. It also includes Bayesian out-of-sample performance analysis.

Pyfolio relies on portfolio returns and position data and can also take into account the transaction costs and slippage losses of trading activity. It uses the empyrical library, which can also be used on a standalone basis to compute performance metrics.

Creating the returns and benchmark inputs

The library is part of the Quantopian ecosystem and is compatible with Zipline and Alphalens. We will first demonstrate how to generate the requisite inputs from Alphalens and then show how to extract them from a Zipline backtest performance `DataFrame`. The code samples for this section are in the notebook `03_pyfolio_demo.ipynb`.

Getting pyfolio input from Alphalens

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 information for each asset.

Getting pyfolio input from a Zipline backtest

The result of a Zipline backtest can also 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 back- and forward testing. The former involves historical data and often refers to the sample period used to fine-tune alpha factor parameters. Forward-testing simulates the strategy on new market data to validate that it performs well out of sample and is not 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. We will address this in more detail in *Chapter 8*, *The ML4T Workflow – From Model to Strategy Backtesting*.

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). Pyfolio computes cumulative returns as the product of simple returns after adding 1 to each:

```
from pyfolio.plotting import plot_rolling_returns
plot_rolling_returns(returns=returns,
                     factor_returns=benchmark_rets,
                     live_start_date='2016-01-01',
                     cone_std=(1.0, 1.5, 2.0))
```

The plot in *Figure 5.4* includes a cone that shows expanding confidence intervals to indicate when out-of-sample returns appear unlikely, given random-walk assumptions. Here, our toy strategy did not perform particularly well against the S&P 500 benchmark during the simulated 2016 out-of-sample period:

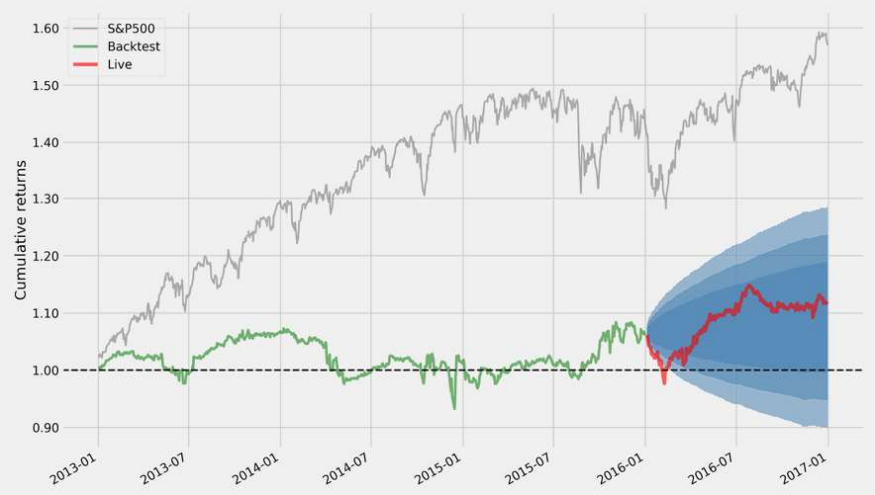

Figure 5.4: Pyfolio cumulative performance plot

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**: 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, absolute 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

The `plot_perf_stats` function bootstraps estimates of parameter variability and displays the result as a box plot:

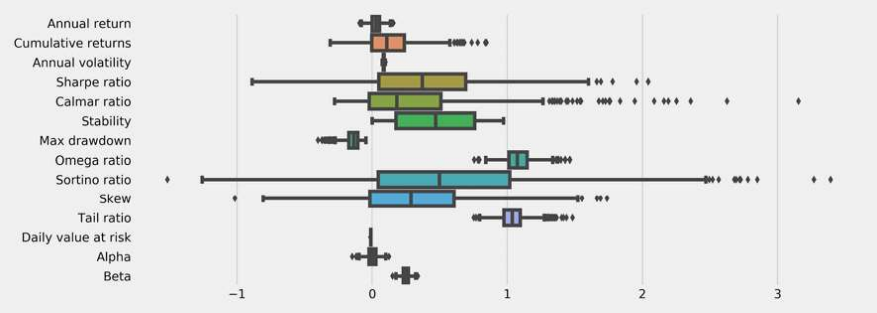

Figure 5.5: Pyfolio performance statistic plot

The `show_perf_stats` function computes numerous metrics for the entire period, as well as separately, for in- and out-of-sample periods:

```
from pyfolio.timeseries import show_perf_stats
show_perf_stats(returns=returns,
                factor_returns=benchmark_rets,
                positions=positions,
                transactions=transactions,
                live_start_date=oos_date)
```

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

Metric	All	In-sample	Out-of-sample

Annual return	2.80%	2.10%	4.70%
Cumulative returns	11.60%	6.60%	4.70%
Annual volatility	8.50%	8.80%	7.60%
Sharpe ratio	0.37	0.29	0.64
Calmar ratio	0.21	0.16	0.57
Stability	0.26	0.01	0.67
Max drawdown	-13.10%	-13.10%	-8.30%
Omega ratio	1.07	1.06	1.11
Sortino ratio	0.54	0.42	0.96
Skew	0.33	0.35	0.25
Kurtosis	7.2	8.04	2
Tail ratio	1.04	1.06	1.01
Daily value at risk	-1.10%	-1.10%	-0.90%
Gross leverage	0.69	0.68	0.72
Daily turnover	8.10%	8.00%	8.40%
Alpha	0	-0.01	0.03
Beta	0.25	0.27	0.17

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)
```

The plots in *Figure 5.6*, which highlights a subset of the visualization contained in the various tear sheets, illustrate how pyfolio allows us to drill down into the performance characteristics and gives us exposure to fundamental drivers of risk and returns:

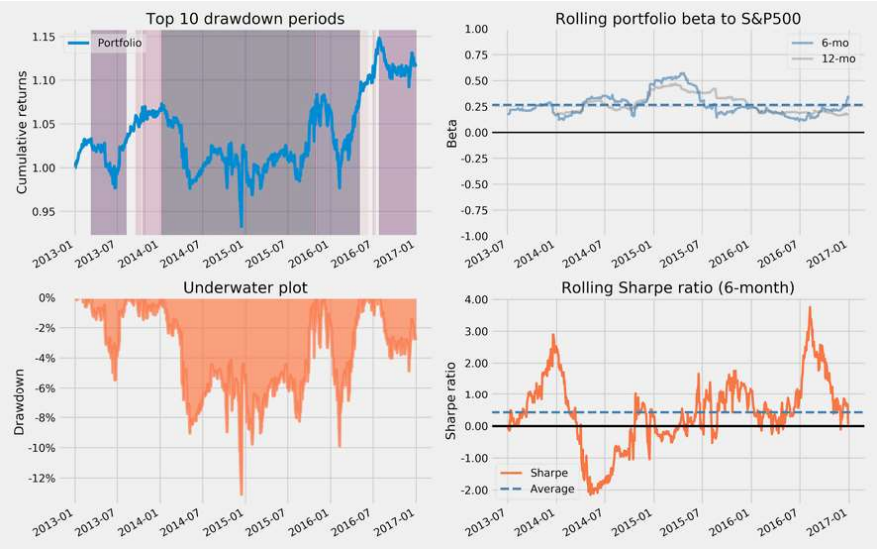

Figure 5.6: Various pyfolio plots of performance over time

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. Pyfolio uses the S&P 500 by default, but you can also provide benchmark returns of your choice. The following example compares the performance to the S&P 500 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')
```

Figure 5.7 shows the resulting plot:

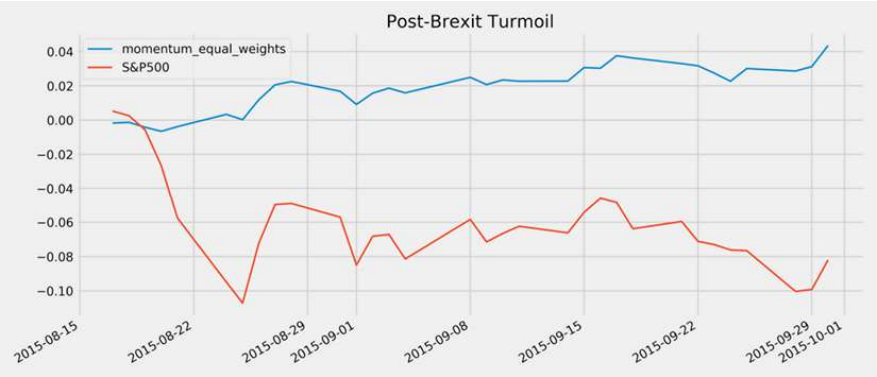

Figure 5.7: Pyfolio event risk analysis

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, as well as to compare the performance of various algorithms.

We saw how important accurate predictions are for optimizing portfolio weights and maximizing diversification benefits. We also explored how machine learning 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 machine learning models. These models will produce more accurate predictions by making more effective use of more diverse information. They do this 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 for defining and backtesting algorithmic trading strategies, which we covered in the previous two chapters.