9
Time-Series Models for Volatility Forecasts and Statistical Arbitrage

In *Chapter 7, Linear Models – From Risk Factors to Asset Return Forecasts*, we introduced linear models for inference and prediction, starting with static models for a contemporaneous relationship with cross-sectional inputs that have an immediate effect on the output. We presented the **ordinary least squares** (**OLS**) learning algorithm, and saw that it produces unbiased coefficients for a correctly specified model with residuals that are not correlated with the input variables. Adding the assumption that the residuals have constant variance guarantees that OLS produces the smallest mean squared prediction error among unbiased estimators.

We also encountered panel data that had both cross-sectional and time-series dimensions, when we learned how the Fama-Macbeth regressions estimate the value of risk factors over time and across assets. However, the relationship between returns across time is typically fairly low, so this procedure could largely ignore the time dimension.

Furthermore, we covered the regularized ridge and lasso regression models, which produce biased coefficient estimates but can reduce the mean squared prediction error. These predictive models took a more dynamic perspective and combined historical returns with other inputs to predict forward returns.

In this chapter, we will build dynamic linear models to explicitly represent time and include variables observed at specific intervals or lags. A key characteristic of time-series data is their sequential order: rather than random samples of individual observations, as in the case of cross-sectional data, our data is a single realization of a stochastic process that we cannot repeat.

Our goal is to identify systematic patterns in time series that help us predict how the time series will behave in the future. More specifically, we will focus on models that extract signals from a historical sequence of the output and, optionally, other contemporaneous or lagged input variables to predict future values of the output. For example, we might try to predict future returns for a stock using past returns, combined with historical returns of a benchmark or macroeconomic variables. We will focus on linear time-series models before turning to nonlinear models like recurrent or convolutional neural networks in Part 4.

Time-series models are very popular given the time dimension inherent to trading. Key applications include the prediction of asset returns and volatility, as well as the identification of the co-movements of asset price series. Time-series data is likely to become more prevalent as an ever-

broader array of connected devices collects regular measurements with potential signal content.

We will first introduce the tools we can use to diagnose time-series characteristics and to extract features that capture potential patterns. Then, we will cover how to diagnose and achieve time-series stationarity. Next, we will introduce univariate and multivariate time-series models and apply them in order to forecast macro data and volatility patterns. We will conclude with the concept of cointegration and how to apply it to develop a pairs trading strategy.

In particular, we will cover the following topics:

- How to use time-series analysis to prepare and inform the modeling process
- Estimating and diagnosing univariate autoregressive and moving-average models
- Building **autoregressive conditional heteroskedasticity (ARCH)** models to predict volatility
- How to build multivariate vector autoregressive models
- Using cointegration to develop a pairs trading strategy

> 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. For a thorough introduction to the topics of this chapter from an investment perspective, see Tsay (2005) and Fabozzi, Focardi, and Kolm (2010).

Tools for diagnostics and feature extraction

A time series is a sequence of values separated by discrete intervals that are typically even spaced (except for missing values). A time series is often modeled as a stochastic process consisting of a collection of random variables, $y(t_1), \ldots, y(t_T)$, with one variable for each point in time, $t_i, i = 1, \ldots, T$. A univariate time series consists of a single value, y, at each point in time, whereas a multivariate time series consists of several observations that can be represented by a vector.

The number of periods, $\Delta t = t_i - t_j$, between distinct points in time, t_i, t_j, is called **lag**, with T-1 distinct lags for each time series. Just as relationships between different variables at a given point in time is key for cross-sectional models, relationships between data points separated by a given lag are fundamental to analyzing and exploiting patterns in time series.

For cross-sectional models, we distinguished between input and output variables, or target and predictors, with the labels y and x, respectively. In a time-series context, some or all of the lagged values $y_{t-1}, y_{t-2}, \cdots, y_{t_T}$ of the outcome y play the role of the input or x values in the cross-section context.

A time series is called **white noise** if it is a sequence of **independent and identically distributed** (**IID**) random variables, ϵ_t, with finite mean and variance. In particular, the series is called a **Gaussian white noise** if the random variables are normally distributed with a mean of zero and a constant variance of σ.

A time series is linear if it can be written as a weighted sum of past disturbances, ϵ_t, that are also called innovations and are here assumed to represent white noise, and the mean of the series, μ:

$$y_t = \mu + \sum_{i=0}^{\infty} a_i \epsilon_{t-i}, \quad a_0 = 1, \epsilon \sim \text{i. i. d}$$

A key goal of time-series analysis is to understand the dynamic behavior that is driven by the coefficients, a_i. The analysis of time series offers methods tailored to this type of data with the goal of extracting useful patterns that, in turn, help us build predictive models.

We will introduce the most important tools for this purpose, including the decomposition into key systematic elements, the analysis of autocorrelation, and rolling window statistics such as moving averages.

For most of the examples in this chapter, we will work with data provided by the Federal Reserve that you can access using pandas-datareader, which we introduced in *Chapter 2, Market and Fundamental Data – Sources and Techniques*. The code examples for this section are available in the notebook `tsa_and_stationarity`.

How to decompose time-series patterns

Time-series data typically contains a mix of patterns that can be decomposed into several components. In particular, a time series often combines systematic components like trend, seasonality, and cycles with unsystematic noise. These components can be modeled as a linear combination (for example, when fluctuations do not depend on the level of the series) or in a nonlinear, multiplicative form.

Based on the model assumptions, they can also be split up automatically. Statsmodels includes a simple method to split the time series into separate trend, seasonal, and residual components using moving averages. We can apply it to monthly data on industrial manufacturing that contain both a strong trend component and a seasonality component, as follows:

```
import statsmodels.tsa.api as tsa
industrial_production = web.DataReader('IPGMFN', 'fred', '1988', '2017-12').squeeze()
components = tsa.seasonal_decompose(industrial_production, model='additive')
ts = (industrial_production.to_frame('Original')
      .assign(Trend=components.trend)
      .assign(Seasonality=components.seasonal)
```

```
        .assign(Residual=components.resid))
ts.plot(subplots=True, figsize=(14, 8));
```

Figure 9.1 shows the resulting charts that display the additive components. The residual component would be the focus of subsequent modeling efforts, assuming that the trend and seasonality components are more deterministic and amenable to simple extrapolation:

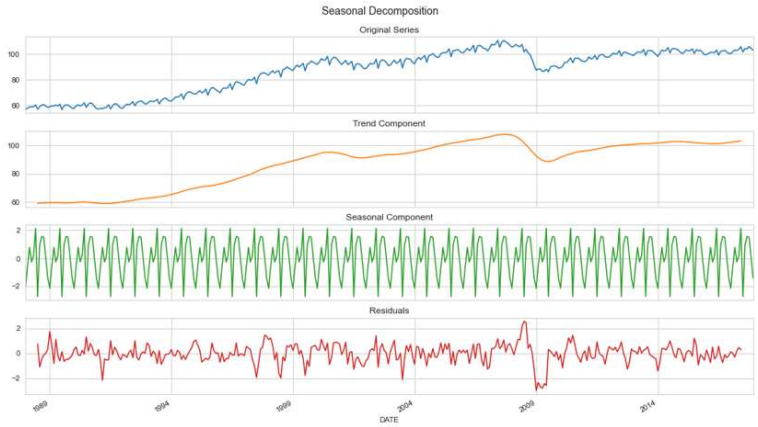

Figure 9.1: Time-series decomposition into trend, seasonality, and residuals

There are more sophisticated model-based approaches—see, for example, *Chapter 6*, *The Machine Learning Process*, in Hyndman and Athanasopoulos (2018).

Rolling window statistics and moving averages

Given the sequential ordering of time-series data, it is natural to compute familiar descriptive statistics for periods of a given length. The goal is to detect whether the series is stable or changes over time and obtain a smoothed representation that captures systematic aspects while filtering out the noise.

Rolling window statistics serve this process: they produce a new time series where each data point represents a summary statistic computed for a certain period of the original data. Moving averages are the most familiar example. The original data points can enter the computation with weights that are equal or, for example, emphasize more recent data points. Exponential moving averages recursively compute weights that decay for data points further in the past. The new data points are typically a summary of all preceding data points, but they can also be computed from a surrounding window.

The pandas library includes rolling or expanding windows and allows for various weight distributions. In a second step, you can apply computations to each set of data captured by a window. These computations include built-in functions for individual series, such as the mean or the sum, and the correlation or covariance for several series, as well as user-defined functions.

We used this functionality to engineer features in *Chapter 4*, *Financial Feature Engineering – How to Research Alpha Factors*, and *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*, for example. The moving average and exponential smoothing examples in the following section will also apply these tools.

Early forecasting models included **moving-average models** with exponential weights called **exponential smoothing models**. We will encounter moving averages again as key building blocks for linear time series. Forecasts that rely on exponential smoothing methods use weighted averages of past observations, where the weights decay exponentially as the observations get older. Hence, a more recent observation receives a higher associated weight. These methods are popular for time series that do not have very complicated or abrupt patterns.

How to measure autocorrelation

Autocorrelation (also called *serial correlation*) adapts the concept of correlation to the time-series context: just as the correlation coefficient measures the strength of a linear relationship between two variables, the **autocorrelation coefficient**, ρ_k, measures the extent of a linear relationship between time-series values separated by a given lag, k:

$$\rho_k = \frac{\sum_{t=k+1}^{T}(y_t - \bar{y})(y_{t-k} - \bar{y})}{\sum_{t=1}^{T}(y_t - \bar{y})^2}$$

Hence, we can calculate one autocorrelation coefficient for each of the T-1 lags in a time series of length T. The **autocorrelation function** (**ACF**) computes the correlation coefficients as a function of the lag.

The autocorrelation for a lag larger than 1 (that is, between observations more than one timestep apart) reflects both the direct correlation between these observations and the indirect influence of the intervening data points. The **partial autocorrelation** removes this influence and only measures the linear dependence between data points at the given lag distance, T. Removing means using the residuals of a linear regression with the outcome x_t and the lagged values x_{t-1}, x_{t-2}, …, x_{T-1} as features (also known as an $AR(T\text{-}1)$ model, which we'll discuss in the next section on univariate time-series models). The **partial autocorrelation function** (**PACF**) provides all the correlations that result once the effects of a correlation at shorter lags have been removed, as described previously.

There are also algorithms that estimate the partial autocorrelation from the sample autocorrelation based on the exact theoretical relationship between the PACF and the ACF.

A **correlogram** is simply a plot of the ACF or PACF for sequential lags, k=0,1,...,n. It allows us to inspect the correlation structure across lags at one glance (see *Figure 9.3* for an example). The main usage of correlograms is to detect any autocorrelation after the removal of a determinis-

tic trend or seasonality. Both the ACF and the PACF are key diagnostic tools for the design of linear time-series models, and we will review examples of ACF and PACF plots in the following section on time-series transformations.

How to diagnose and achieve stationarity

The statistical properties, such as the mean, variance, or autocorrelation, of a **stationary time series** are independent of the period—that is, they don't change over time. Thus, **stationarity** implies that a time series does not have a trend or seasonal effects. Furthermore, it requires that descriptive statistics, such as the mean or the standard deviation, when computed for different rolling windows, are constant or do not change significantly over time. A stationary time series reverts to its mean, and the deviations have a constant amplitude, while short-term movements are always alike in a statistical sense.

More formally, **strict stationarity** requires the joint distribution of any subset of time-series observations to be independent of time with respect to all moments. So, in addition to the mean and variance, higher moments such as skew and kurtosis also need to be constant, irrespective of the lag between different observations. In most applications, such as most time-series models in this chapter that we can use to model asset returns, we limit stationarity to first and second moments so that the time series is covariance stationary with constant mean, variance, and autocorrelation. However, we abandon this assumption when building modeling volatility and explicitly assume the variance to change over time in predictable ways.

Note that we specifically allow for **dependence between output values at different lags**, just like we want the input data for linear regression to be correlated with the outcome. Stationarity implies that these relationships are stable. Stationarity is a key assumption of classical statistical models. The following two subsections introduce transformations that can help make a time series stationary, as well as how to address the special case of a stochastic trend caused by a unit root.

Transforming a time series to achieve stationarity

To satisfy the stationarity assumption of many time-series models, we need to transform the original series, often in several steps. Common transformations include the (natural) **logarithm** to convert an exponential growth pattern into a linear trend and stabilize the variance. **Deflation** implies dividing a time series by another series that causes trending behavior, for example, dividing a nominal series by a price index to convert it into a real measure.

A series is **trend-stationary** if it reverts to a stable long-run linear trend. It can often be made stationary by fitting a trend line using linear regression and using the residuals. This implies including the time index as an

independent variable in a linear regression model, possibly combined with logging or deflating.

In many cases, detrending is not sufficient to make the series stationary. Instead, we need to transform the original data into a series of **period-to-period and/or season-to-season differences**. In other words, we use the result of subtracting neighboring data points or values at seasonal lags from each other. Note that when such differencing is applied to a log-transformed series, the results represent instantaneous growth rates or returns in a financial context.

If a univariate series becomes stationary after differencing d times, it is said to be integrated of the order of d, or simply integrated if d=1. This behavior is due to unit roots, which we will explain next.

Handling instead of how to handle

Unit roots pose a particular problem for determining the transformation that will render a time series stationary. We will first explain the concept of a unit root before discussing diagnostics tests and solutions.

On unit roots and random walks

Time series are often modeled as stochastic processes of the following autoregressive form so that the current value is a weighted sum of past values, plus a random disturbance:

$$y_t = a_1 y_{t-1} + a_2 y_{t-2} + \ldots + a_p y_{t-p} + \epsilon_t$$

We will explore these models in more detail as the AR building block for ARIMA models in the next section on univariate time-series models. Such a process has a characteristic equation of the following form:

$$m^p - m^{p-1} a_1 - m^{p-2} a_2 - \ldots - a_p = 0$$

If one of the (up to) p roots of this polynomial equals 1, then the process is said to have a **unit root**. It will be non-stationary but will not necessarily have a trend. If the remaining roots of the characteristic equation are less than 1 in absolute terms, the first difference of the process will be stationary, and the **process is integrated of order 1 or I(1)**. With additional roots larger than 1 in absolute terms, the order of integration is higher and additional differencing will be required.

In practice, time series of interest rates or asset prices are often not stationary because there isn't a price level to which the series reverts. The most prominent example of a non-stationary series is the random walk. Given a time series of prices p_t with starting price p_0(for example, a stock's IPO price) and a white-noise disturbance ϵ_t, then a random walk satisfies the following autoregressive relationship:

$$p_t = p_{t-1} + \epsilon_t = \sum_{s-0}^{t} \epsilon_s + p_0$$

Repeated substitution shows that the current value, p_t, is the sum of all prior disturbances or innovations, ϵ_t, and the initial price, p_0. If the equation includes a constant term, then the random walk is said to have **drift**.

The random walk is thus an **autoregressive stochastic process** of the following form:

$$y_t = a_1 y_{t-1} + \epsilon_t, \;\; a_1 = 1$$

It has the characteristic equation $m - a_1 = 0$ with a unit root and is both non-stationary and integrated of order 1. On the one hand, given the IID nature of ε, the variance of the time series equals $t\sigma^2$, which is **not second-order stationary**, and implies that, in principle, the series could assume any value over time. On the other hand, taking the **first difference**, $\Delta p_t = p_t - p_{t-1}$, leaves $\Delta p_t = \varepsilon_t$, which is **stationary**, given the statistical assumption about ε.

The defining characteristic of a non-stationary series with a unit-root is **long memory**: since current values are the sum of past disturbances, large innovations persist for much longer than for a mean-reverting, stationary series.

How to diagnose a unit root

Statistical unit root tests are a common way to determine objectively whether (additional) differencing is necessary. These are statistical hypothesis tests of stationarity that are designed to determine whether differencing is required.

The **augmented Dickey-Fuller test** (**ADF test**) evaluates the null hypothesis that a time-series sample has a unit root against the alternative of stationarity. It regresses the differenced time series on a time trend, the first lag, and all lagged differences, and computes a test statistic from the value of the coefficient on the lagged time-series value. `statsmodels` makes it easy to implement (see the notebook `tsa_and_stationarity`).

Formally, the ADF test for a time series, y_t, runs the linear regression where α is a constant, β is a coefficient on a time trend, and p refers to the number of lags used in the model:

$$\Delta y_t = \alpha + \beta t + \gamma y_{t-1} + \delta_1 \Delta y_{t-1} + \ldots + \delta_{p-1} \Delta y_{t-p+1} + \epsilon_t$$

The constraint $\alpha = \beta = 0$ implies a random walk, whereas only $\beta = 0$ implies a random walk with drift. The lag order is usually decided using the

Akaike information criterion (**AIC**) and **Bayesian information criterion** (**BIC**) information criteria introduced in *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*.

The ADF test statistic uses the sample coefficient γ, which, under the null hypothesis of unit-root non-stationarity, equals zero and is negative otherwise. It intends to demonstrate that, for an integrated series, the lagged series value should not provide useful information in predicting the first difference above and beyond lagged differences.

How to remove unit roots and work with the resulting series

In addition to using the difference between neighboring data points to remove a constant pattern of change, we can apply **seasonal differencing** to remove patterns of seasonal change. This involves taking the difference of values at a lag distance that represents the length of the seasonal pattern. For monthly data, this usually involves differences at lag 12, and for quarterly data, it involves differences at lag 4 to remove both seasonality and linear trend.

Identifying the correct transformation and, in particular, the appropriate number and lags for differencing is not always clear-cut. Some **heuristics** have been suggested, which can be summarized as follows:

- Lag-1 autocorrelation close to zero or negative, or autocorrelation generally small and patternless: there is no need for higher-order differencing
- Positive autocorrelations up to 10+ lags: the series probably needs higher-order differencing
- Lag-1 autocorrelation < -0.5: the series may be over-differenced
- Slightly over- or under-differencing can be corrected with AR or MA terms (see the next section on univariate time-series models)

Some authors recommend fractional differencing as a more flexible approach to rendering an integrated series stationary, and may be able to keep more information or signal than simple or seasonal differences at discrete intervals. See, for example, *Chapter 5*, *Portfolio Optimization and Performance Evaluation*, in Marcos Lopez de Prado (2018).

Time-series transformations in practice

The charts in *Figure 9.2* shows time series for the NASDAQ stock index and industrial production for the 30 years through 2017 in their original form, as well as the transformed versions after applying the logarithm and subsequently applying the first and seasonal differences (at lag 12), respectively.

The charts also display the ADF p-value, which allows us to reject the hypothesis of unit-root non-stationarity after all transformations in both cases:

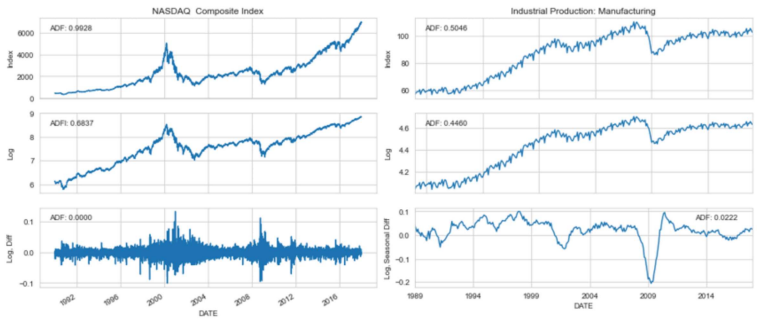

Figure 9.2: Time-series transformations and unit-root test results

We can further analyze the relevant time-series characteristics for the transformed series using a Q-Q plot that compares the quantiles of the distribution of the time-series observation to the quantiles of the normal distribution and the correlograms based on the ACF and PACF.

For the NASDAQ plots in *Figure 9.3*, we can see that while there is no trend, the variance is not constant but rather shows clustered spikes around periods of market turmoil in the late 1980s, 2001, and 2008. The Q-Q plot highlights the fat tails of the distribution with extreme values that are more frequent than the normal distribution would suggest.

The ACF and the PACF show similar patterns, with autocorrelation at several lags appearing to be significant:

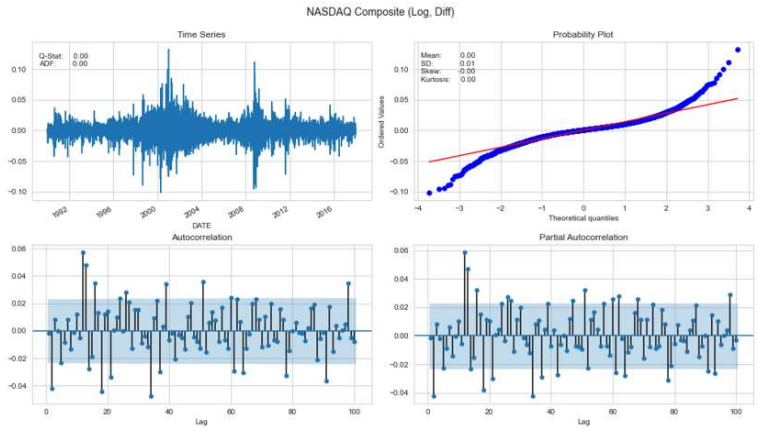

Figure 9.3: Descriptive statistics for transformed NASDAQ Composite index

For the monthly time series on industrial manufacturing production, we can see a large negative outlier following the 2008 crisis, as well as the corresponding skew in the Q-Q plot (see *Figure 9.4*). The autocorrelation is much higher than for the NASDAQ returns and declines smoothly. The PACF shows distinct positive autocorrelation patterns at lags 1 and 13 and significant negative coefficients at lags 3 and 4:

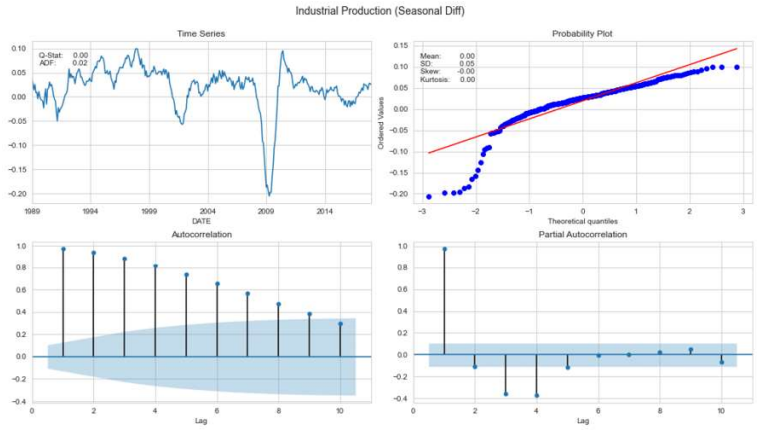

Figure 9.4: Descriptive statistics for transformed industrial production data

Univariate time-series models

Multiple linear-regression models expressed the variable of interest as a linear combination of the inputs, plus a random disturbance. In contrast, univariate time-series models relate the current value of the time series to a linear combination of lagged values of the series, current noise, and possibly past noise terms.

While exponential smoothing models are based on a description of the trend and seasonality in the data, **ARIMA models aim to describe the autocorrelations in the data**. ARIMA(*p*, *d*, *q*) models require stationarity and leverage two building blocks:

- **Autoregressive** (**AR**) terms consisting of *p* lagged values of the time series
- **Moving average** (**MA**) terms that contain *q* lagged disturbances

The **I** stands for *integrated* because the model can account for unit-root non-stationarity by differentiating the series *d* times. The term autoregression underlines that ARIMA models imply a regression of the time series on its own values.

We will introduce the ARIMA building blocks, AR and MA models, and explain how to combine them in **autoregressive moving-average** (**ARMA**) models that may account for series integration as ARIMA models or include exogenous variables as **AR(I)MAX** models. Furthermore, we will illustrate how to include seasonal AR and MA terms to extend the toolbox so that it also includes **SARMAX** models.

How to build autoregressive models

An AR model of order *p* aims to capture the linear dependence between time-series values at different lags and can be written as follows:

$$\text{AR}(p):\ y_t = \emptyset_0 + \emptyset_1 y_{t-1} + \emptyset_2 y_{t-2} + \ldots + \emptyset_p y_{t-p} + \epsilon_t,\ \ \epsilon \sim \text{i.i.d}$$

This closely resembles a multiple linear regression on lagged values of y_t. This model has the following characteristic equation:

$$1 - \emptyset_1 x - \emptyset_2 x^2 - \ldots - \emptyset_p x^p = 0$$

The inverses of the solution to this polynomial of degree p in x are the characteristic roots, and the AR(p) process is stationary if all roots are less than 1 in absolute terms, and unstable otherwise. For a stationary series, multistep forecasts will converge to the mean of the series.

We can estimate the model parameters with the familiar least squares method using the p+1, ..., T observations to ensure there is data for each lagged term and the outcome.

How to identify the number of lags

In practice, the challenge consists of deciding on the appropriate order p of lagged terms. The time-series analysis tools for serial correlation, which we discussed in the *How to measure autocorrelation* section, play a key role in making this decision.

More specifically, a visual inspection of the correlogram often provides helpful clues:

- The **ACF** estimates the autocorrelation between observations at different lags, which, in turn, results from both direct and indirect linear dependence. Hence, if an AR model of order k is the correct model, the ACF will show a significant serial correlation up to lag k and, due to the inertia caused by the indirect effects of the linear relationship, will extend to subsequent lags until it eventually trails off as the effect weakens.
- The **PACF**, in turn, only measures the direct linear relationship between observations a given lag apart so that it will not reflect correlation for lags beyond k.

How to diagnose model fit

If the model properly captures the linear dependence across lags, then the residuals should resemble white noise, and the ACF should highlight the absence of significant autocorrelation coefficients.

In addition to a residual plot, the **Ljung-Box Q-statistic** allows us to test the hypothesis that the residual series follows white noise. The null hypothesis is that all m serial correlation coefficients are zero against the alternative that some coefficients are not. The test statistic is computed from the sample autocorrelation coefficients $\rho_k$for different lags k and follows a X^2 distribution:

$$Q(m) = T(T+2)\sum_{t=1}^{m}\frac{\rho_l^2}{T-l}$$

As we will see, statsmodels provides information about the significance of coefficients for different lags, and insignificant coefficients should be removed. If the Q-statistic rejects the null hypothesis of no autocorrelation, you should consider additional AR terms.

How to build moving-average models

An MA(q) model uses q past disturbances rather than lagged values of the time series in a regression-like model, as follows:

$$\text{MA}(q):\ y_t = c + \epsilon_t + \theta_1\epsilon_{t-1} + \theta_2\epsilon_{t-2} + \ldots + \theta_p\epsilon_{t-p},\ \ \epsilon \sim \text{i.i.d}$$

Since we do not observe the white-noise disturbance values, ϵ_t, MA(q) is not a regression model like the ones we have seen so far. Rather than using least squares, MA(q) models are estimated using **maximum likelihood** (**MLE**), alternatively initializing or estimating the disturbances at the beginning of the series and then recursively and iteratively computing the remainder.

The MA(q) model gets its name from representing each value of y_t as a weighted moving average of the past q innovations. In other words, current estimates represent a correction relative to past errors made by the model. The use of moving averages in MA(q) models differs from that of exponential smoothing, or the estimation of seasonal time-series components, because an MA(q) model aims to forecast future values, as opposed to denoising or estimating the trend cycle of past values.

MA(q) processes are always stationary because they are the weighted sum of white noise variables that are, themselves, stationary.

How to identify the number of lags

A time series generated by an MA(q) process is driven by the residuals of the prior q model predictions. Hence, the ACF for the MA(q) process will show significant coefficients for values up to lag q and then decline sharply because this is how the model assumes the series values have been generated.

Note how this differs from the AR case we just described, where the PACF would show a similar pattern.

The relationship between the AR and MA models

An AR(p) model can always be expressed as an MA(∞) process using repeated substitution, as in the random walk example in the *How to handle stochastic trends caused by unit roots* section.

When the coefficients of the MA(q) process meet certain size constraints, it also becomes invertible and can be expressed as an AR(∞) process (see Tsay, 2005, for details).

How to build ARIMA models and extensions

Autoregressive integrated moving-average—ARIMA(p, d, q)—models combine AR(p) and MA(q) processes to leverage the complementarity of these building blocks and simplify model development. They do this using a more compact form and reducing the number of parameters, in turn reducing the risk of overfitting.

The models also take care of eliminating unit-root non-stationarity by using the d^{th} difference of the time-series values. An ARIMA(p, 1, q) model is the same as using an ARMA(p, q) model with the first differences of the series. Using y' to denote the original series after non-seasonal differencing d times, the ARIMA(p, d, q) model is simply:

$$\begin{aligned}\text{ARIMA}(p,d,q): \quad y'_t &= \text{AR}(p) + \text{MA}(q) \\ &= \phi_0 + \phi_1 y'_{t-1} + \ldots + \phi_p y'_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \ldots + \theta_q \epsilon_{t-q}, \;\; \epsilon \sim \text{i.i.d.}\end{aligned}$$

ARIMA models are also estimated using MLE. Depending on the implementation, higher-order models may generally subsume lower-order models.

For example, up to version 0.11, statsmodels includes all lower-order p and q terms and does not permit removing coefficients for lags below the highest value. In this case, higher-order models will always fit better. Be careful not to overfit your model to the data by using too many terms. The most recent version, which is 0.11 at the time of writing, added an experimental new ARIMA model with more flexible configuration options.

How to model differenced series

There are also guidelines for designing the univariate times-series models when using data:

- A model without differencing assumes that the original series is stationary, including mean-reverting. It normally includes a constant term to allow for a non-zero mean.
- A model with one order of differencing assumes that the original series has a constant trend and should thus include a constant term.
- A model with two orders of differencing assumes that the original series has a time-varying trend and should not include a constant.

How to identify the number of AR and MA terms

Since AR(p) and MA(q) terms interact, the information provided by the ACF and PACF is no longer reliable and can only be used as a starting point.

Traditionally, the AIC and BIC information criteria have been used to rely on in-sample fit when selecting the model design. Alternatively, we can

rely on out-of-sample tests to cross-validate multiple parameter choices.

The following summary provides some guidance on how to choose the model order in the case of considering AR and MA models in isolation:

- The lag beyond which the PACF cuts off is the indicated number of AR terms. If the PACF of the differenced series cuts off sharply and/or the lag-1 autocorrelation is positive, add one or more AR terms.
- The lag beyond which the ACF cuts off is the indicated number of MA terms. If the ACF of the differenced series displays a sharp cutoff and/or the lag-1 autocorrelation is negative, consider adding an MA term to the model.
- AR and MA terms may cancel out each other's effects, so always try to reduce the number of AR and MA terms by 1 if your model contains both to avoid overfitting, especially if the more complex model requires more than 10 iterations to converge.
- If the AR coefficients sum to nearly one and suggest a unit root in the AR part of the model, eliminate one AR term and difference the model once (more).
- If the MA coefficients sum to nearly one and suggest a unit root in the MA part of the model, eliminate one MA term and reduce the order of differencing by one.
- Unstable long-term forecasts suggest there may be a unit root in the AR or MA part of the model.

Adding features – ARMAX

An **autoregressive moving-average model with exogenous inputs (ARMAX)** model adds input variables or covariate on the right-hand side of the ARMA time-series model (assuming the series is stationary, so we can skip differencing):

$$\begin{aligned}\text{ARIMA}(p,d,q): \quad y_t &= \beta x_t + \text{AR}(p) + \text{MA}(q) \\ &= \beta x_t + \phi_0 + \phi_1 y_{t-1} + \cdots + \phi_p y_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \cdots + \theta_q \epsilon_{t-q}, \;\; \epsilon \sim \text{i.i.d.}\end{aligned}$$

This resembles a linear regression model but is quite difficult to interpret. This is because the effect of β on y_t is not the effect of an increase in x_t by one unit as in linear regression. Instead, the presence of lagged values of y_t on the right-hand side of the equation implies that the coefficient can only be interpreted, given the lagged values of the response variable, which is hardly intuitive.

Adding seasonal differencing – SARIMAX

For time series with seasonal effects, we can include AR and MA terms that capture the seasonality's periodicity. For instance, when using monthly data and the seasonal effect length is 1 year, the seasonal AR and MA terms would reflect this particular lag length.

The ARIMAX(p, d, q) model then becomes a SARIMAX(p, d, q) × (P, D, Q) model, which is a bit more complicated to write out, but the statsmodels documentation (see link on GitHub) provides this information in detail.

We will now build a seasonal ARMA model using macro-data to illustrate its implementation.

How to forecast macro fundamentals

We will build a SARIMAX model for monthly data on an industrial production time series for the 1988-2017 period. As illustrated in the first section on analytical tools, the data has been log-transformed, and we are using seasonal (lag-12) differences. We estimate the model for a range of both ordinary and conventional AR and MA parameters using a rolling window of 10 years of training data, and evaluate the **root mean square error** (**RMSE**) of the 1-step-ahead forecast, as shown in the following simplified code (see the notebook `arima_models` for details):

```
for p1 in range(4):                 # AR order
    for q1 in range(4):             # MA order
        for p2 in range(3):         # seasonal AR order
            for q2 in range(3):     # seasonal MA order
                y_pred = []
                for i, T in enumerate(range(train_size, len(data))):
                    train_set = data.iloc[T - train_size:T]
                    model = tsa.SARIMAX(endog=train_set, # model specification
                                        order=(p1, 0, q1),
                                        seasonal_order=(p2, 0, q2, 12)).fit()
                    preds.iloc[i, 1] = model.forecast(steps=1)[0]
                mse = mean_squared_error(preds.y_true, preds.y_pred)
                results[(p1, q1, p2, q2)] = [np.sqrt(mse),
                    preds.y_true.sub(preds.y_pred).std(),
                    np.mean(aic)]
```

We also collect the AIC and BIC criteria, which show a very high rank correlation coefficient of 0.94, with BIC favoring models with slightly fewer parameters than AIC. The best five models by RMSE are:

```
                  RMSE         AIC         BIC
p1 q1 p2 q2
2  3  1  0   0.009323 -772.247023 -752.734581
3  2  1  0   0.009467 -768.844028 -749.331586
2  2  1  0   0.009540 -770.904835 -754.179884
   3  0  0   0.009773 -760.248885 -743.523935
   2  0  0   0.009986 -758.775827 -744.838368
```

We reestimate a SARIMA(2, 0 ,3) × (1, 0, 0) model, as follows:

```
best_model = tsa.SARIMAX(endog=industrial_production_log_diff, order=(2, 0, 3),
                         seasonal_order=(1, 0, 0, 12)).fit()
print(best_model.summary())
```

We obtain the following summary:

Statespace Model Results

Dep. Variable:	IPGMFN	No. Observations:	348
Model:	SARIMAX(2, 0, 3)x(1, 0, 0, 12)	Log Likelihood	1139.719
Date:	Sat, 22 Sep 2018	AIC	-2265.438
Time:	17:48:17	BIC	-2238.472
Sample:	01-01-1989 - 12-01-2017	HQIC	-2254.702
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	1.4934	0.104	14.351	0.000	1.289	1.697
ar.L2	-0.5159	0.102	-5.083	0.000	-0.715	-0.317
ma.L1	-0.5499	0.114	-4.813	0.000	-0.774	-0.326
ma.L2	0.2872	0.062	4.662	0.000	0.166	0.408
ma.L3	0.1815	0.070	2.589	0.010	0.044	0.319
ar.S.L12	-0.4486	0.047	-9.533	0.000	-0.541	-0.356
sigma2	8.141e-05	5.65e-06	14.399	0.000	7.03e-05	9.25e-05

Ljung-Box (Q):	61.58	Jarque-Bera (JB):	9.97
Prob(Q):	0.02	Prob(JB):	0.01
Heteroskedasticity (H):	1.07	Skew:	-0.20
Prob(H) (two-sided):	0.71	Kurtosis:	3.73

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

Figure 9.5: SARMAX model results

The coefficients are significant, and the Q-statistic rejects the hypothesis of further autocorrelation. The correlogram similarly indicates that we have successfully eliminated the series' autocorrelation:

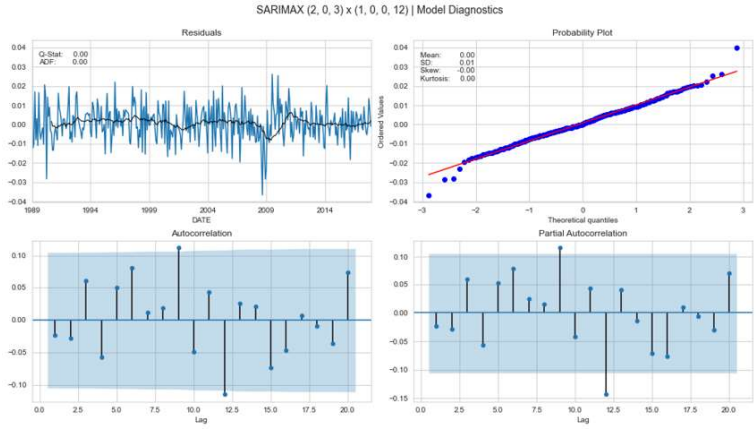

Figure 9.6: SARIMAX model diagnostics

How to use time-series models to forecast volatility

A particularly important application for univariate time-series models in finance is the prediction of volatility. This is because it is usually not constant over time, with bouts of volatility clustering together. Changes in variance create challenges for time-series forecasting using the classical ARIMA models that assume stationarity. To address this challenge, we will now model volatility so that we can predict changes in variance.

Heteroskedasticity is the technical term for changes in a variable's variance. The ARCH model expresses the variance of the error term as a function of the errors in previous periods. More specifically, it assumes that the error variance follows an AR(*p*) model.

The **generalized autoregressive conditional heteroskedasticity** (**GARCH**) model broadens the scope of ARCH to allow for ARMA models.

Time-series forecasting often combines ARIMA models for the expected mean and ARCH/GARCH models for the expected variance of a time series. The 2003 Nobel Prize in Economics was awarded to Robert Engle and Clive Granger for developing this class of models. The former also runs the Volatility Lab at New York University's Stern School (`vlab.stern.nyu.edu`), which has numerous online examples and tools concerning the models we will discuss.

The ARCH model

The ARCH(p) model is simply an AR(p) model that's applied to the variance of the residuals of a time-series model, which makes this variance at time t conditional on lagged observations of the variance.

More specifically, the error terms, ε_t, are residuals of a linear model, such as ARIMA, on the original time series and are split into a time-dependent standard deviation, σ_t, and a disturbance, z_t, as follows:

$$\begin{aligned} \text{ARCH}(p): \quad \text{var}(x_t) &= \sigma_t^2 \\ &= \omega + \alpha_1 \epsilon_{t-1}^2 + \cdots + \alpha_p \epsilon_{t-p}^2 \\ \epsilon_t &= \sigma_t z_t \\ z_t &\sim \text{i.i.d.} \end{aligned}$$

An ARCH(p) model can be estimated using OLS. Engle proposed a method to identify the appropriate ARCH order using the Lagrange multiplier test, which corresponds to the F-test of the hypothesis that all coefficients in linear regression are zero (see *Chapter 7, Linear Models – From Risk Factors to Return Forecasts*).

A key **strength** of the ARCH model is that it produces volatility estimates with positive excess kurtosis — that is, fat tails relative to the normal distribution — which, in turn, is in line with empirical observations about returns. **Weaknesses** include the assumption of the same effect for positive and negative volatility shocks, whereas asset prices tend to respond differently. It also does not explain the variations in volatility and is likely to overpredict volatility because they respond slowly to large, isolated shocks to the return series.

For a properly specified ARCH model, the standardized residuals (divided by the model estimate for the period of standard deviation) should resemble white noise and can be subjected to a Ljung-Box Q test.

Generalizing ARCH – the GARCH model

The ARCH model is relatively simple but often requires many parameters to capture the volatility patterns of an asset-return series. The GARCH model applies to a log-return series, r_t, with disturbances, $\epsilon_t = r_t - \mu$, that follow a GARCH(p, q) model if:

$$\epsilon_t = \sigma_t z_t, \;\; \sigma_t^2 = \omega + \sum_{i=1}^{p} \alpha_i \epsilon_{t-i}^2 + \sum_{j=1}^{q} \beta_i \sigma_{t-j}^2, \;\; z_t \sim \text{i. i. d}$$

The GARCH(p, q) model assumes an ARMA(p, q) model for the variance of the error term, ε_t.

Similar to ARCH models, the tail distribution of a GARCH(1,1) process is heavier than that of a normal distribution. The model encounters the same weaknesses as the ARCH model. For instance, it responds equally to positive and negative shocks.

To configure the lag order for ARCH and GARCH models, use the squared residuals of the time series trained to predict the mean of the original series. The residuals are zero-centered so that their squares are also the variance. Then, inspect the ACF and PACF plots of the squared residuals to identify autocorrelation patterns in the variance of the time series.

How to build a model that forecasts volatility

The development of a volatility model for an asset-return series consists of four steps:

1. Build an ARMA time-series model for the financial time series based on the serial dependence revealed by the ACF and PACF
2. Test the residuals of the model for ARCH/GARCH effects, again relying on the ACF and PACF for the series of the squared residual
3. Specify a volatility model if serial correlation effects are significant, and jointly estimate the mean and volatility equations
4. Check the fitted model carefully and refine it if necessary

When applying volatility forecasting to return series, the serial dependence may be limited so that a constant mean may be used instead of an ARMA model.

The `arch` library (see link to the documentation on GitHub) provides several options to estimate volatility-forecasting models. You can model the expected mean as a constant, as an AR(p) model, as discussed in the *How to build autoregressive models*, section or as more recent **heterogeneous autoregressive processes** (**HAR**), which use daily (1 day), weekly (5 days), and monthly (22 days) lags to capture the trading frequencies of short-, medium-, and long-term investors.

The mean models can be jointly defined and estimated with several conditional heteroskedasticity models that include, in addition to ARCH and GARCH, the **exponential GARCH** (**EGARCH**) model, which allows for asymmetric effects between positive and negative returns, and the **heterogeneous ARCH** (**HARCH**) model, which complements the HAR mean model.

We will use daily NASDAQ returns from 2000-2020 to demonstrate the usage of a GARCH model (see the notebook `arch_garch_models` for details):

```
nasdaq = web.DataReader('NASDAQCOM', 'fred', '2000', '2020').squeeze()
nasdaq_returns = np.log(nasdaq).diff().dropna().mul(100) # rescale to facilitate optimization
```

The rescaled daily return series exhibits only limited autocorrelation, but the squared deviations from the mean do have substantial memory reflected in the slowly decaying ACF and the PACF, which are high for the first two and cut off only after the first six lags:

```
plot_correlogram(nasdaq_returns.sub(nasdaq_returns.mean()).pow(2), lags=120,                    titl
```

The function `plot_correlogram` produces the following output:

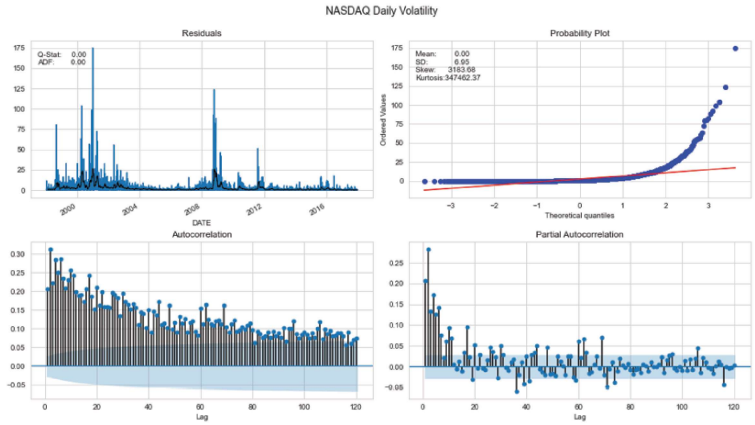

Figure 9.7: Daily NASDAQ composite volatility

Hence, we can estimate a GARCH model to capture the linear relationship of past volatilities. We will use rolling 10-year windows to estimate a GARCH(p, q) model with p and q ranging from 1-4 to generate one-step out-of-sample forecasts.

We then compare the RMSE of the predicted volatility relative to the actual squared deviation of the return from its mean to identify the most predictive model. We are using winsorized data to limit the impact of extreme return values being reflected in the very high positive skew of the volatility:

```
trainsize = 10 * 252  # 10 years
data = nasdaq_returns.clip(lower=nasdaq_returns.quantile(.05),
                           upper=nasdaq_returns.quantile(.95))
T = len(nasdaq_returns)
results = {}
for p in range(1, 5):
    for q in range(1, 5):
        print(f'{p} | {q}')
        result = []
        for s, t in enumerate(range(trainsize, T-1)):
            train_set = data.iloc[s: t]
            test_set = data.iloc[t+1]  # 1-step ahead forecast
            model = arch_model(y=train_set, p=p, q=q).fit(disp='off')
            forecast = model.forecast(horizon=1)
            mu = forecast.mean.iloc[-1, 0]
```

```
            var = forecast.variance.iloc[-1, 0]
            result.append([(test_set-mu)**2, var])
        df = pd.DataFrame(result, columns=['y_true', 'y_pred'])
        results[(p, q)] = np.sqrt(mean_squared_error(df.y_true, df.y_pred))
```

The GARCH(2, 2) model achieves the lowest RMSE (same value as GARCH(4, 2) but with fewer parameters), so we go ahead and estimate this model to inspect the summary:

```
am = ConstantMean(nasdaq_returns.clip(lower=nasdaq_returns.quantile(.05),
                                      upper=nasdaq_returns.quantile(.95)))
am.volatility = GARCH(2, 0, 2)
am.distribution = Normal()
best_model = am.fit(update_freq=5)
print(best_model.summary())
```

The output shows the maximized log-likelihood, as well as the AIC and BIC criteria, which are commonly minimized when selecting models based on in-sample performance (see *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*). It also displays the result for the mean model, which, in this case, is just a constant estimate, as well as the GARCH parameters for the constant omega, the AR parameters, α, and the MA parameters, β, all of which are statistically significant:

```
                     Constant Mean - GARCH Model Results
==============================================================================
Dep. Variable:              NASDAQCOM   R-squared:                      -0.001
Mean Model:             Constant Mean   Adj. R-squared:                 -0.001
Vol Model:                      GARCH   Log-Likelihood:               -7244.08
Distribution:                  Normal   AIC:                           14500.2
Method:            Maximum Likelihood   BIC:                           14539.1
                                        No. Observations:                 4851
Date:                Thu, Apr 16 2020   Df Residuals:                     4845
Time:                        22:41:39   Df Model:                            6
                            Mean Model
==============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
------------------------------------------------------------------------------
mu             0.0526  1.416e-02      3.714  2.043e-04 [2.484e-02,8.036e-02]
                              Volatility Model
==============================================================================
                 coef    std err          t      P>|t|      95.0% Conf. Int.
------------------------------------------------------------------------------
omega          0.0270  1.047e-02      2.574  1.005e-02 [6.430e-03,4.748e-02]
alpha[1]       0.0350  1.581e-02      2.215  2.678e-02 [4.027e-03,6.601e-02]
alpha[2]       0.0581  3.943e-02      1.473      0.141  [-1.919e-02,  0.135]
beta[1]        0.8675      0.535      1.622      0.105     [ -0.181,  1.916]
beta[2]        0.0179      0.495  3.618e-02      0.971     [ -0.952,  0.987]
==============================================================================

Covariance estimator: robust
```

Figure 9.8: GARCH Model results

Let's now explore models for multiple time series and the concept of cointegration, which will enable a new trading strategy.

Multivariate time-series models

Multivariate time-series models are designed to capture the dynamic of multiple time series simultaneously and leverage dependencies across these series for more reliable predictions. The most comprehensive introduction to this subject is Lütkepohl (2005).

Systems of equations

Univariate time-series models, like the ARMA approach we just discussed, are limited to statistical relationships between a target variable and its lagged values or lagged disturbances and exogenous series, in the case of ARMAX. In contrast, multivariate time-series models also allow for lagged values of other time series to affect the target. This effect applies to all series, resulting in complex interactions, as illustrated in the following diagram:

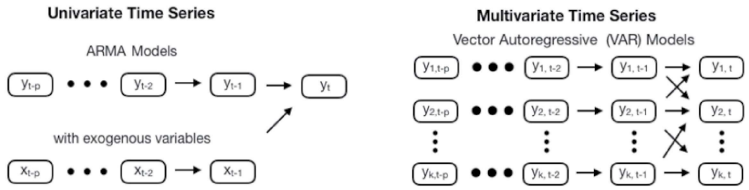

Figure 9.9: Interactions in univariate and multivariate time-series models

In addition to potentially better forecasting, multivariate time series are also used to gain insights into cross-series dependencies. For example, in economics, multivariate time series are used to understand how policy changes to one variable, such as an interest rate, may affect other variables over different horizons.

The **impulse-response** function produced by the multivariate model serves this purpose and allows us to simulate how one variable responds to a sudden change in other variables. The concept of **Granger causality** analyzes whether one variable is useful in forecasting another (in the least-squares sense). Furthermore, multivariate time-series models allow for a decomposition of the prediction error variance to analyze how other series contribute.

The vector autoregressive (VAR) model

We will see how the **vector autoregressive VAR(p) model** extends the AR(*p*) model to *k* series by creating a system of *k* equations, where each contains *p* lagged values of all *k* series. In the simplest case, a VAR(1) model for *k*=2 takes the following form:

$$y_{1,t} = c_1 + \alpha_{1,1} y_{1,t-1} + \alpha_{1,2} y_{2,t-1} + \epsilon_{1,t}$$
$$y_{2,t} = c_2 + \alpha_{2,1} y_{1,t-1} + \alpha_{2,2} y_{2,t-1} + \epsilon_{2,t}$$

This model can be expressed somewhat more concisely in **matrix form**:

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} + \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix} \begin{bmatrix} y_{1,t-1} \\ y_{2,t-2} \end{bmatrix} + \begin{bmatrix} \epsilon_{1,t} \\ \epsilon_{2,t} \end{bmatrix}$$

The **coefficients** on the lagged values of the output provide information about the dynamics of the series itself, whereas the cross-variable coeffi-

cients offer some insight into the interactions across the series. This notation extends to k time series and order p, as follows:

$$\underset{k \times 1}{\boldsymbol{y}_t} = \underset{k \times 1}{\boldsymbol{c}} + \underset{k \times k}{\boldsymbol{A}_1}\underset{k \times 1}{\boldsymbol{y}_{t-1}} + \ldots + \underset{k \times k}{\boldsymbol{A}_p}\underset{k \times 1}{\boldsymbol{y}_{t-p}} + \underset{k \times 1}{\boldsymbol{\epsilon}_t}$$

VAR(p) models also require **stationarity** so that the initial steps from univariate time-series modeling carry over. First, explore the series and determine the necessary transformations. Then, apply the augmented Dickey-Fuller test to verify that the stationarity criterion is met for each series and apply further transformations otherwise. It can be estimated with an OLS conditional on initial information or with MLE, which is the equivalent for normally distributed errors but not otherwise.

If some or all of the k series are unit-root non-stationary, they may be **cointegrated** (see the next section). This extension of the unit root concept to multiple time series means that a linear combination of two or more series is stationary and, hence, mean-reverting.

The VAR model is not equipped to handle this case without differencing; instead, use the **vector error correction model** (**VECM**, Johansen and Juselius 1990). We will further explore cointegration because, if present and assumed to persist, it can be leveraged for a pairs-trading strategy.

The **determination of the lag order** also takes its cues from the ACF and PACF for each series, but is constrained by the fact that the same lag order applies to all series. After model estimation, **residual diagnostics** also call for a result resembling white noise, and model selection can use in-sample information criteria or, if the goal is to use the model for prediction, out-of-sample predictive performance to cross-validate alternative model designs.

As mentioned in the univariate case, predictions of the original time series require us to reverse the transformations applied to make a series stationary before training the model.

Using the VAR model for macro forecasts

We will extend the univariate example of using a single time series of monthly data on industrial production and add a monthly time series on consumer sentiment, both of which are provided by the Federal Reserve's data service. We will use the familiar pandas-datareader library to retrieve data from 1970 through 2017:

```
df = web.DataReader(['UMCSENT', 'IPGMFN'],
                    'fred', '1970', '2017-12').dropna()
df.columns = ['sentiment', 'ip']
```

Log-transforming the industrial production series and seasonal differencing using a lag of 12 for both series yields stationary results:

```
df_transformed = pd.DataFrame({'ip': np.log(df.ip).diff(12),
                               'sentiment': df.sentiment.diff(12)}).dropna()
test_unit_root(df_transformed) # see notebook for details and additional plots
           p-value
ip          0.0003
sentiment   0.0000
```

This leaves us with the following series:

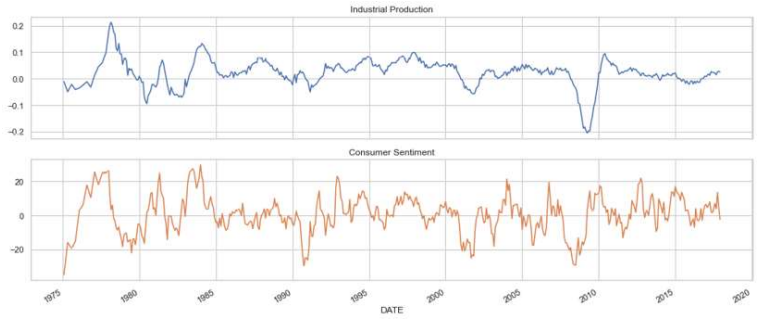

Figure 9.10: Transformed time series: industrial production and consumer sentiment

To limit the size of the output, we will just estimate a VAR(1) model using the statsmodels `VARMAX` implementation (which allows for optional exogenous variables) with a constant trend using the first 480 observations:

```
model = VARMAX(df_transformed.loc[:'2017'], order=(1,1),
               trend='c').fit(maxiter=1000)
```

This produces the following summary:

```
                               Statespace Model Results
==========================================================================================
Dep. Variable:     ['ip', 'sentiment']   No. Observations:                   468
Model:                      VARMA(1,1)   Log Likelihood                -71.870
                          + intercept   AIC                            169.741
Date:                Thu, 16 Apr 2020   BIC                            223.671
Time:                        22:55:23   HQIC                           190.962
Sample:                             0
                                  - 468
Covariance Type:                  opg
==========================================================================================
Ljung-Box (Q):                127.93, 161.51   Jarque-Bera (JB):         128.70, 17.04
Prob(Q):                        0.00, 0.00   Prob(JB):                   0.00, 0.00
Heteroskedasticity (H):         0.48, 1.10   Skew:                       0.19, 0.21
Prob(H) (two-sided):            0.00, 0.57   Kurtosis:                   5.54, 3.83
                                Results for equation ip
==========================================================================================
                     coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------------------
intercept          0.0015      0.001      2.401      0.016       0.000       0.003
L1.ip              0.9284      0.010     93.628      0.000       0.909       0.948
L1.sentiment       0.0006   6.03e-05     10.059      0.000       0.000       0.001
L1.e(ip)           0.0116      0.037      0.311      0.756      -0.062       0.085
L1.e(sentiment) -9.925e-05      0.000     -0.814      0.415      -0.000       0.000
                            Results for equation sentiment
==========================================================================================
                     coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------------------
intercept          0.3374      0.279      1.208      0.227      -0.210       0.885
L1.ip            -14.3677      5.450     -2.636      0.008     -25.049      -3.687
L1.sentiment       0.8801      0.023     37.598      0.000       0.834       0.926
L1.e(ip)          39.6834     18.798      2.111      0.035       2.839      76.528
L1.e(sentiment)    0.0509      0.052      0.983      0.326      -0.051       0.152
                              Error covariance matrix
==========================================================================================
                         coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------------------
sqrt.var.ip            0.0129      0.000     40.298      0.000       0.012       0.014
sqrt.cov.ip.sentiment  0.0368      0.231      0.159      0.873      -0.416       0.489
sqrt.var.sentiment     5.2738      0.148     35.519      0.000       4.983       5.565
==========================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

Figure 9.11: VAR(1) model results

The output contains the coefficients for both time-series equations, as outlined in the preceding VAR(1) illustration. statsmodels provides diagnostic plots to check whether the residuals meet the white noise assumptions.

This is not exactly the case in this simple example because the variance does not appear to be constant (upper left) and the quantile plot shows differences in the distribution, namely fat tails (lower left):

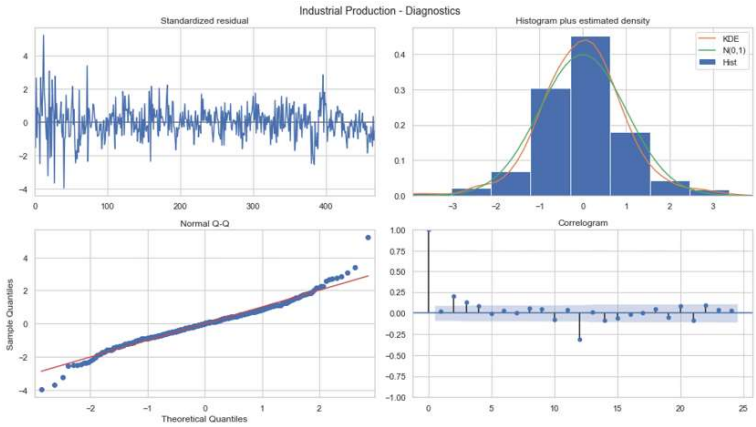

Figure 9.12: statsmodels VAR model diagnostic plot

You can generate out-of-sample predictions as follows:

```
preds = model.predict(start=480, end=len(df_transformed)-1)
```

The following visualization of actual and predicted values shows how the prediction lags the actual values and does not capture nonlinear, out-of-sample patterns well:

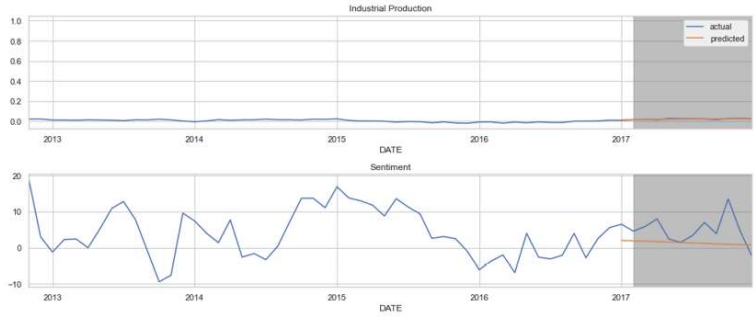

Figure 9.13: VAR model predictions versus actuals

Cointegration – time series with a shared trend

We briefly mentioned cointegration in the previous section on multivariate time-series models. Let's now explain this concept and how to diagnose its presence in more detail before leveraging it for a statistical arbitrage trading strategy.

We have seen how a time series can have a unit root that creates a stochastic trend and makes the time series highly persistent. When we use such an integrated time series in their original, rather than in differ-

enced, form as a feature in a linear regression model, its relationship with the outcome will often appear statistically significant, even though it is not. This phenomenon is called spurious regression (for details, see *Chapter 18*, *CNNs for Financial Time Series and Satellite Images*, in Wooldridge, 2008). Therefore, the recommended solution is to difference the time series so they become stationary before using them in a model.

However, there is an exception when there are cointegration relationships between the outcome and one or more input variables. To understand the concept of cointegration, let's first remember that the residuals of a regression model are a linear combination of the inputs and the output series.

Usually, the residuals of the regression of one integrated time series on one or more such series yields non-stationary residuals that are also integrated, and thus behave like a random walk. However, for some time series, this is not the case: the regression produces coefficients that yield a linear combination of the time series in the form of the residuals that are stationary, even though the individual series are not. Such time series are *cointegrated*.

A non-technical example is that of a drunken man on a random walk accompanied by his dog (on a leash). Both trajectories are non-stationary but cointegrated because the dog will occasionally revert to his owner. In the trading context, arbitrage constraints imply cointegration between spot and futures prices.

In other words, a **linear combination of two or more cointegrated series has a stable mean** to which this linear combination reverts. This also applies when the individual series are integrated of a higher order and the linear combination reduces the overall order of integration.

Cointegration differs from correlation: two series can be highly correlated but need not be cointegrated. For example, if two growing series are constant multiples of each other, their correlation will be high, but any linear combination will also grow rather than revert to a stable mean.

Cointegration is very useful: if two or more asset price series tend to revert to a common mean, we can leverage deviations from the trend because they should imply future price moves in the opposite direction. The mathematics behind cointegration is more involved, so we will only focus on the practical aspects; for an in-depth treatment, see Lütkepohl (2005).

In this section, we will address how we can identify pairs with such a long-term stationary relationship, estimate the expected time for any disequilibrium to correct, and how to utilize these tools to implement and backtest a long-short pairs trading strategy.

There are two approaches to testing for cointegration:

- The Engle-Granger two-step method
- The Johansen test

We'll discuss each in turn before we show how they help identify cointegrated securities that tend to revert to a common trend, a fact that we can leverage for a statistical arbitrage strategy.

The Engle-Granger two-step method

The **Engle-Granger method** is used to identify cointegration relationships between two series. It involves both of the following:

1. Regressing one series on another to estimate the stationary long-term relationship
2. Applying an ADF unit-root test to the regression residual

The null hypothesis is that the residuals have a unit root and are integrated; if we can reject it, then we assume that the residuals are stationary and, thus, the series are cointegrated (Engle and Granger 1987).

A key benefit of this approach is that the regression coefficient represents the multiplier that renders the combination stationary, that is, mean-reverting. Unfortunately, the test results will differ, depending on which variable we consider independent, so that we try both ways and then pick the relation with the more negative test statistic that has the lower p-value.

Another downside is that this test is limited to pairwise relationships. The more complex Johansen procedure can identify significant cointegration among up to a dozen time series.

The Johansen likelihood-ratio test

The **Johansen procedure**, in contrast, tests the restrictions imposed by cointegration on a VAR model, as discussed in the previous section. More specifically, after subtracting the target vector from both sides of a generic VAR(p) model, we obtain the **error correction model** (**ECM**) formulation:

$$\Delta \boldsymbol{y_t} = \boldsymbol{c} + \boldsymbol{\Pi y_{t-1}} + \boldsymbol{\Gamma_1 \Delta y_{t-1}} + \ldots + \boldsymbol{\Gamma_p \Delta y_{t-p}} + \boldsymbol{\epsilon_t}$$

The resulting modified VAR(p) equation has only one vector term in levels (y_{t-1}) that is not expressed as a difference using the Δ operator. The nature of cointegration depends on the rank of the coefficient matrix Π of this term (Johansen 1991).

While this equation appears structurally similar to the ADF test setup, there are now several potential constellations of common trends because there are multiple series involved. To identify the number of cointegration relationships, the Johansen test successively tests for an increasing rank of Π, starting at 0 (no cointegration). We will explore the application to the case of two series in the following section.

Gonzalo and Lee (1998) discuss practical challenges due to misspecified model dynamics and other implementation aspects, including how to

combine both test procedures that we will rely on for our sample statistical arbitrage strategy in the next section.

Statistical arbitrage with cointegration

Statistical arbitrage refers to strategies that employ some statistical model or method to take advantage of what appears to be relative mispricing of assets, while maintaining a level of market neutrality.

Pairs trading is a conceptually straightforward strategy that has been employed by algorithmic traders since at least the mid-eighties (Gatev, Goetzmann, and Rouwenhorst 2006). The goal is to find two assets whose prices have historically moved together, track the spread (the difference between their prices), and, once the spread widens, buy the loser that has dropped below the common trend and short the winner. If the relationship persists, the long and/or the short leg will deliver profits as prices converge and the positions are closed.

This approach extends to a multivariate context by forming baskets from multiple securities and trading one asset against a basket of two baskets against each other.

In practice, the strategy requires two steps:

1. **Formation phase**: Identify securities that have a long-term mean-reverting relationship. Ideally, the spread should have a high variance to allow for frequent profitable trades while reliably reverting to the common trend.
2. **Trading phase**: Trigger entry and exit trading rules as price movements cause the spread to diverge and converge.

Several approaches to the formation and trading phases have emerged from increasingly active research in this area, across multiple asset classes, over the last several years. The next subsection outlines the key differences before we dive into an example application.

How to select and trade comoving asset pairs

A recent comprehensive survey of pairs trading strategies (Krauss 2017) identified four different methodologies, plus a number of other more recent approaches, including ML-based forecasts:

- **Distance approach**: The oldest and most-studied method identifies candidate pairs with distance metrics like correlation and uses non-parametric thresholds like Bollinger Bands to trigger entry and exit trades. Its computational simplicity allows for large-scale applications with demonstrated profitability across markets and asset classes for extended periods of time since Gatev, et al. (2006). However, performance has decayed more recently.
- **Cointegration approach**: As outlined previously, this approach relies on an econometric model of a long-term relationship among two or more variables, and allows for statistical tests that promise more relia-

bility than simple distance metrics. Examples in this category use the Engle-Granger and Johansen procedures to identify pairs and baskets of securities, as well as simpler heuristics that aim to capture the concept (Vidyamurthy 2004). Trading rules often resemble the simple thresholds used with distance metrics.

- **Time-series approach**: With a focus on the trading phase, strategies in this category aim to model the spread as a mean-reverting stochastic process and optimize entry and exit rules accordingly (Elliott, Hoek, and Malcolm 2005). It assumes promising pairs have already been identified.
- **Stochastic control approach**: Similar to the time-series approach, the goal is to optimize trading rules using stochastic control theory to find value and policy functions to arrive at an optimal portfolio (Liu and Timmermann 2013). We will address this type of approach in *Chapter 21*, *Generative Adversarial Networks for Synthetic Time-Series Data*.
- **Other approaches**: Besides pair identification based on unsupervised learning like principal component analysis (see *Chapter 13*, *Data-Driven Risk Factors and Asset Allocation with Unsupervised Learning*) and statistical models like copulas (Patton 2012), machine learning has become popular more recently to identify pairs based on their relative price or return forecasts (Huck 2019). We will cover several ML algorithms that can be used for this purpose and illustrate corresponding multivariate pairs trading strategies in the coming chapters.

This summary of the various approaches offers barely a glimpse at the flexibility afforded by the design of a pairs trading strategy. In addition to higher-level questions about pair selection and trading rule logic, there are **numerous parameters** that we need **to define for implementation**. These parameters include the following:

- Investment universe to screen for potential pairs or baskets
- Length of the formation period
- Strength of the relationship used to pick tradeable candidates
- Degree of deviation from and convergence to their common means to trigger entry or exit trades or to adjust existing positions as spreads fluctuate

Pairs trading in practice

The **distance approach** identifies pairs using the correlation of (normalized) asset prices or their returns, and is simple and orders of magnitude less computationally intensive than cointegration tests. The notebook `cointegration_test` illustrates this for a sample of ~150 stocks with 4 years of daily data: it takes ~30ms to compute the correlation with the returns of an ETF, compared to 18 seconds for a suite of cointegration tests (using statsmodels) – 600x slower.

The **speed advantage** is particularly valuable. This is because the number of potential pairs is the product of the number of candidates to be considered on either side so that evaluating combinations of 100 stocks and 100 ETFs requires comparing 10,000 tests (we'll discuss the challenge of multiple testing bias later).

On the other hand, distance metrics do not necessarily select the most profitable pairs: correlation is maximized for perfect co-movement, which, in turn, eliminates actual trading opportunities. Empirical studies confirm that the volatility of the price spread of cointegrated pairs is almost twice as high as the volatility of the price spread of distance pairs (Huck and Afawubo 2015).

To balance the **tradeoff between computational cost and the quality of the resulting pairs**, Krauss (2017) recommends a procedure that combines both approaches based on his literature review:

1. Select pairs with a stable spread that shows little drift to reduce the number of candidates
2. Test the remaining pairs with the highest spread variance for cointegration

This process aims to select cointegrated pairs with lower divergence risk while ensuring more volatile spreads that, in turn, generate higher profit opportunities.

A large number of tests introduce **data snooping bias**, as discussed in *Chapter 6*, *The Machine Learning Process*: multiple testing is likely to increase the number of false positives that mistakenly reject the null hypothesis of no cointegration. While statistical significance may not be necessary for profitable trading (Chan 2008), a study of commodity pairs (Cummins and Bucca 2012) shows that controlling the familywise error rate to improve the tests' power, according to Romano and Wolf (2010), can lead to better performance.

In the following subsection, we'll take a closer look at how predictive various heuristics for the degree of comovement of asset prices are for the result of cointegration tests.

The example code uses a sample of 172 stocks and 138 ETFs traded on the NYSE and NASDAQ, with daily data from 2010 - 2019 provided by Stooq.

The securities represent the largest average dollar volume over the sample period in their respective class; highly correlated and stationary assets have been removed. See the notebook `create_datasets` in the `data` folder of the GitHub repository for instructions on how to obtain the data, and the notebook `cointegration_tests` for the relevant code and additional preprocessing and exploratory details.

Distance-based heuristics to find cointegrated pairs

`compute_pair_metrics()` computes the following distance metrics for over 23,000 pairs of stocks and **Exchange Traded Funds (ETFs)** for 2010-14 and 2015-19:

- The **drift of the spread**, defined as a linear regression of a time trend on the spread
- The **spread's volatility**
- The **correlations** between the normalized price series and between their returns

Low drift and volatility, as well as high correlation, are simple proxies for cointegration.

To evaluate the predictive power of these heuristics, we also run **Engle-Granger and Johansen cointegration** tests using statsmodels for the preceding pairs. This takes place in the loop in the second half of `compute_pair_metrics()`.

We first estimate the optimal number of lags that we need to specify for the Johansen test. For both tests, we assume that the cointegrated series (the spread) may have an intercept different from zero but no trend:

```
def compute_pair_metrics(security, candidates):
    security = security.div(security.iloc[0])
    ticker = security.name
    candidates = candidates.div(candidates.iloc[0])
    # compute heuristics
    spreads = candidates.sub(security, axis=0)
    n, m = spreads.shape
    X = np.ones(shape=(n, 2))
    X[:, 1] = np.arange(1, n + 1)
    drift = ((np.linalg.inv(X.T @ X) @ X.T @ spreads).iloc[1]
             .to_frame('drift'))
    vol = spreads.std().to_frame('vol')
    corr_ret = (candidates.pct_change()
                .corrwith(security.pct_change())
                .to_frame('corr_ret'))
    corr = candidates.corrwith(security).to_frame('corr')
    metrics = drift.join(vol).join(corr).join(corr_ret).assign(n=n)
    tests = []
    # compute cointegration tests
    for candidate, prices in candidates.items():
        df = pd.DataFrame({'s1': security, 's2': prices})
        var = VAR(df)
        lags = var.select_order() # select VAR order
        k_ar_diff = lags.selected_orders['aic']
        # Johansen Test with constant Term and estd. lag order
        cj0 = coint_johansen(df, det_order=0, k_ar_diff=k_ar_diff)
        # Engle-Granger Tests
        t1, p1 = coint(security, prices, trend='c')[:2]
        t2, p2 = coint(prices, security, trend='c')[:2]
        tests.append([ticker, candidate, t1, p1, t2, p2,
                      k_ar_diff, *cj0.lr1])

    return metrics.join(tests)
```

To check for the **significance of the cointegration tests**, we compare the Johansen trace statistic for rank 0 and 1 to their respective critical values and obtain the Engle-Granger p-value.

We follow the recommendation by Gonzalo and Lee (1998), mentioned at the end of the previous section, to apply both tests and accept pairs where they agree. The authors suggest additional due diligence in case of disagreement, which we are going to skip:

```
spreads['trace_sig'] = ((spreads.trace0 > trace0_cv) &
                        (spreads.trace1 > trace1_cv)).astype(int)
spreads['eg_sig'] = (spreads.p < .05).astype(int)
```

For the over 46,000 pairs across both sample periods, the Johansen test considers 3.2 percent of the relationships as significant, while the Engle-Granger considers 6.5 percent. They agree on 366 pairs (0.79 percent).

How well do the heuristics predict significant cointegration?

When we compare the distributions of the heuristics for series that are cointegrated according to both tests with the remainder that is not, volatility and drift are indeed lower (in absolute terms). *Figure 9.14* shows that the picture is less clear for the two correlation measures:

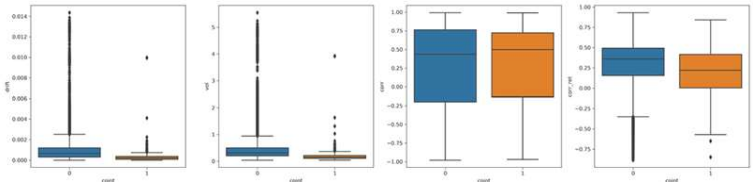

Figure 9.14: The distribution of heuristics, broken down by the significance of both cointegration tests

To evaluate the predictive accuracy of the heuristics, we first run a logistic regression model with these features to predict significant cointegration. It achieves an **area-under-the-curve (AUC)** cross-validation score of 0.815; excluding the correlation metrics, it still scores 0.804. A decision tree does slightly better at AUC=0.821, with or without the correlation features.

Not least due to the strong class imbalance, there are large numbers of false positives: correctly identifying 80 percent of the 366 cointegrated pairs implies over 16,500 false positives, but eliminates almost 30,000 of the candidates. See the notebook `cointegration_tests` for additional detail.

The **key takeaway** is that distance heuristics can help screen a large universe more efficiently, but this comes at a cost of missing some cointegrated pairs and still requires substantial testing.

Preparing the strategy backtest

In this section, we are going to implement a statistical arbitrage strategy based on cointegration for the sample of stocks and ETFs and the 2017-2019 period. Some aspects are simplified to streamline the presentation. See the notebook `statistical_arbitrage_with_cointegrated_pairs` for the code examples and additional detail.

We first generate and store the cointegration tests for all candidate pairs and the resulting trading signals. Then, we backtest a strategy based on these signals, given the computational intensity of the process.

Precomputing the cointegration tests

First, we run quarterly cointegration tests over a 2-year lookback period on each of the 23,000 potential pairs, Then, we select pairs where both the Johansen and the Engle-Granger tests agree for trading. We should exclude assets that are stationary during the lookback period, but we eliminated assets that are stationary for the entire period, so we skip this step to simplify it.

This procedure follows the steps outlined previously; please see the notebook for details.

Figure 9.15 shows the original stock and ETF series of the two different pairs selected for trading; note the clear presence of a common trend over the sample period:

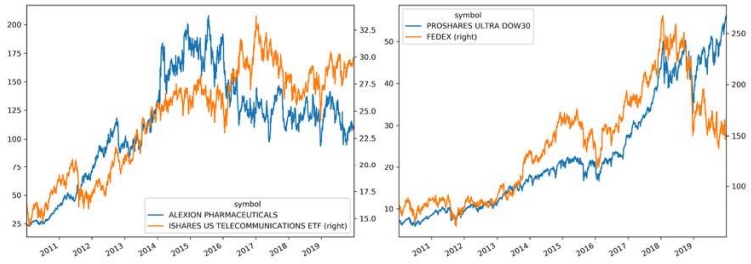

Figure 9.15: Price series for two selected pairs over the sample period

Getting entry and exit trades

Now, we can compute the spread for each candidate pair based on a rolling hedge ratio. We also calculate a **Bollinger Band** because we will consider moves of the spread larger than two rolling standard deviations away from its moving average as **long and short entry signals**, and crossings of the moving average in reverse as exit signals.

Smoothing prices with the Kalman filter

To this end, we first apply a rolling **Kalman filter** (**KF**) to remove some noise, as demonstrated in *Chapter 4, Financial Feature Engineering – How to Research Alpha Factors*:

```
def KFSmoother(prices):
    """Estimate rolling mean"""

    kf = KalmanFilter(transition_matrices=np.eye(1),
                      observation_matrices=np.eye(1),
                      initial_state_mean=0,
                      initial_state_covariance=1,
                      observation_covariance=1,
                      transition_covariance=.05)
    state_means, _ = kf.filter(prices.values)
    return pd.Series(state_means.flatten(),
                     index=prices.index)
```

Computing the rolling hedge ratio using the Kalman filter

To obtain a dynamic hedge ratio, we use the KF for rolling linear regression, as follows:

```
def KFHedgeRatio(x, y):
    """Estimate Hedge Ratio"""
    delta = 1e-3
    trans_cov = delta / (1 - delta) * np.eye(2)
    obs_mat = np.expand_dims(np.vstack([[x], [np.ones(len(x))]]).T, axis=1)
    kf = KalmanFilter(n_dim_obs=1, n_dim_state=2,
                      initial_state_mean=[0, 0],
                      initial_state_covariance=np.ones((2, 2)),
                      transition_matrices=np.eye(2),
                      observation_matrices=obs_mat,
                      observation_covariance=2,
                      transition_covariance=trans_cov)
    state_means, _ = kf.filter(y.values)
    return -state_means
```

Estimating the half-life of mean reversion

If we view the spread as a mean-reverting stochastic process in continuous time, we can model it as an Ornstein-Uhlenbeck process. The benefit of this perspective is that we gain a formula for the half-life of mean reversion, as an approximation of the time required for the spread to converge again after a deviation (see *Chapter 2*, *Market and Fundamental Data – Sources and Techniques*, in Chan 2013 for details):

```
def estimate_half_life(spread):
    X = spread.shift().iloc[1:].to_frame().assign(const=1)
    y = spread.diff().iloc[1:]
    beta = (np.linalg.inv(X.T@X)@X.T@y).iloc[0]
    halflife = int(round(-np.log(2) / beta, 0))
    return max(halflife, 1)
```

Computing spread and Bollinger Bands

The following function orchestrates the preceding computations and expresses the spread as a z-score that captures deviations from the moving average with a window equal to two half-lives in terms of the rolling standard deviations:

```
def get_spread(candidates, prices):
    pairs, half_lives = [], []
    periods = pd.DatetimeIndex(sorted(candidates.test_end.unique()))
    start = time()
    for p, test_end in enumerate(periods, 1):
        start_iteration = time()
        period_candidates = candidates.loc[candidates.test_end == test_end,
                                           ['y', 'x']]
        trading_start = test_end + pd.DateOffset(days=1)
        t = trading_start - pd.DateOffset(years=2)
        T = trading_start + pd.DateOffset(months=6) - pd.DateOffset(days=1)
        max_window = len(prices.loc[t: test_end].index)
        print(test_end.date(), len(period_candidates))
```

```
        for i, (y, x) in enumerate(zip(period_candidates.y,
                                       period_candidates.x), 1):
            pair = prices.loc[t: T, [y, x]]
            pair['hedge_ratio'] = KFHedgeRatio(
                y=KFSmoother(prices.loc[t: T, y]),
                x=KFSmoother(prices.loc[t: T, x]))[:, 0]
            pair['spread'] = pair[y].add(pair[x].mul(pair.hedge_ratio))
            half_life = estimate_half_life(pair.spread.loc[t: test_end])
            spread = pair.spread.rolling(window=min(2 * half_life,
                                                    max_window))
            pair['z_score'] = pair.spread.sub(spread.mean()).div(spread.
std())
            pairs.append(pair.loc[trading_start: T].assign(s1=y, s2=x, period=p, pair=i).drop([x,
            half_lives.append([test_end, y, x, half_life])
    return pairs, half_lives
```

Getting entry and exit dates for long and short positions

Finally, we use the set of z-scores to derive trading signals:

1. We enter a long (short) position if the z-score is below (above) two, which implies the spread has moved two rolling standard deviations below (above) the moving average
2. We exit trades when the spread crosses the moving average again

We derive rules on a quarterly basis for the set of pairs that passed the cointegration tests during the prior lookback period but allow pairs to exit during the subsequent 3 months.

We again simplify this by dropping pairs that do not close during this 6-month period. Alternatively, we could have handled this using the stop-loss risk management that we included in the strategy (see the next section on backtesting):

```
def get_trades(data):
    pair_trades = []
    for i, ((period, s1, s2), pair) in enumerate(
             data.groupby(['period', 's1', 's2']), 1):
        if i % 100 == 0:
            print(i)
        first3m = pair.first('3M').index
        last3m = pair.last('3M').index
        entry = pair.z_score.abs() > 2
        entry = ((entry.shift() != entry)
                 .mul(np.sign(pair.z_score))
                 .fillna(0)
                 .astype(int)
                 .sub(2))
        exit = (np.sign(pair.z_score.shift().fillna(method='bfill'))
                != np.sign(pair.z_score)).astype(int) - 1
        trades = (entry[entry != -2].append(exit[exit == 0])
                  .to_frame('side')
                  .sort_values(['date', 'side'])
                  .squeeze())
        trades.loc[trades < 0] += 2
        trades = trades[trades.abs().shift() != trades.abs()]
        window = trades.loc[first3m.min():first3m.max()]
```

```
            extra = trades.loc[last3m.min():last3m.max()]
            n = len(trades)
            if window.iloc[0] == 0:
                if n > 1:
                    print('shift')
                    window = window.iloc[1:]
            if window.iloc[-1] != 0:
                extra_exits = extra[extra == 0].head(1)
                if extra_exits.empty:
                    continue
                else:
                    window = window.append(extra_exits)
            trades = (pair[['s1', 's2', 'hedge_ratio', 'period', 'pair']]
                      .join(window. to_frame('side'), how='right'))
            trades.loc[trades.side == 0, 'hedge_ratio'] = np.nan
            trades.hedge_ratio = trades.hedge_ratio.ffill()
            pair_trades.append(trades)
        return pair_trades
```

Backtesting the strategy using backtrader

Now, we are ready to formulate our strategy on our backtesting platform, execute it, and evaluate the results. To do so, we need to track our pairs, in addition to individual portfolio positions, and monitor the spread of active and inactive pairs to apply our trading rules.

Tracking pairs with a custom DataClass

To account for active pairs, we define a `dataclass` (introduced in Python 3.7—see the Python documentation for details). This data structure, called `Pair`, allows us to store the pair components, their number of shares, and the hedge ratio, and compute the current spread and the return, among other things. See a simplified version in the following code:

```
@dataclass
class Pair:
    period: int
    s1: str
    s2: str
    size1: float
    size2: float
    long: bool
    hr: float
    p1: float
    p2: float
    entry_date: date = None
    exit_date: date = None
    entry_spread: float = np.nan
    exit_spread: float = np.nan
    def compute_spread(self, p1, p2):
        return p1 * self.size1 + p2 * self.size2
    def compute_spread_return(self, p1, p2):
        current_spread = self.compute_spread(p1, p2)
        delta = self.entry_spread - current_spread
        return (delta / (np.sign(self.entry_spread) *
                         self.entry_spread))
```

Running and evaluating the strategy

Key implementation aspects include:

- The daily exit from pairs that have either triggered the exit rule or exceeded a given negative return
- The opening of new long and short positions for pairs whose spreads triggered entry signals
- In addition, we adjust positions to account for the varying number of pairs

The code for the strategy itself takes up too much space to display here; see the notebook `pairs_trading_backtest` for details.

Figure 9.16 shows that, at least for the 2017-2019 period, this simplified strategy had its moments (note that we availed ourselves of some look-ahead bias and ignored transaction costs).

Under these lax assumptions, it underperformed the S&P 500 at the beginning and end of the period and was otherwise roughly in line (left panel). It yields an alpha of 0.08 and a negative beta of -0.14 (right panel), with an average Sharpe ratio of 0.75 and a Sortino ratio of 1.05 (central panel):

Figure 9.16: Strategy performance metrics

While we should take these performance metrics with a grain of salt, the strategy demonstrates the anatomy of a statistical arbitrage based on cointegration in the form of pairs trading. Let's take a look at a few steps you could take to build on this framework to produce better performance.

Extensions – how to do better

Cointegration is a very useful concept to identify pairs or groups of stocks that tend to move in unison. Compared to the statistical sophistication of cointegration, we used very simple and static trading rules; the computation on a quarterly basis also distorts the strategy, as the patterns of long and short holdings show (see notebook).

To be successful, you will, at a minimum, need to screen a larger universe and optimize several of the parameters, including the trading rules. Moreover, risk management should account for concentrated positions that arise when certain assets appear relatively often on the same side of a traded pair.

You could also operate with baskets as opposed to individual pairs; however, to address the growing number of candidates, you would likely need to constrain the composition of the baskets.

As mentioned in the *Pairs trading – statistical arbitrage with cointegration* section, there are alternatives that aim to predict price movements. In the following chapters, we will explore various machine learning models that aim to predict the absolute size or the direction of price movements for a given investment universe and horizon. Using these forecasts as long and short entry signals is a natural extension or alternative to the pairs trading framework that we studied in this section.

Summary

In this chapter, we explored linear time-series models for the univariate case of individual series, as well as multivariate models for several interacting series. We encountered applications that predict macro fundamentals, models that forecast asset or portfolio volatility with widespread use in risk management, and multivariate VAR models that capture the dynamics of multiple macro series. We also looked at the concept of cointegration, which underpins the popular pair-trading strategy.

Similar to *Chapter 7*, *Linear Models – From Risk Factors to Return Forecasts*, we saw how linear models impose a lot of structure, that is, they make strong assumptions that potentially require transformations and extensive testing to verify that these assumptions are met. If they are, model-training and interpretation are straightforward, and the models provide a good baseline that more complex models may be able to improve on. In the next two chapters, we will see two examples of this, namely random forests and gradient boosting models, and we will encounter several more in *Part 4*, which is on deep learning.