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:

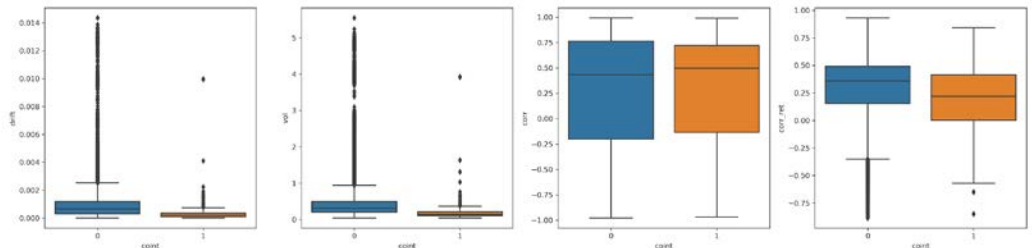

Figure 9.14: The distribution of heuristics, broken down by the significance of both cointegration tests

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:

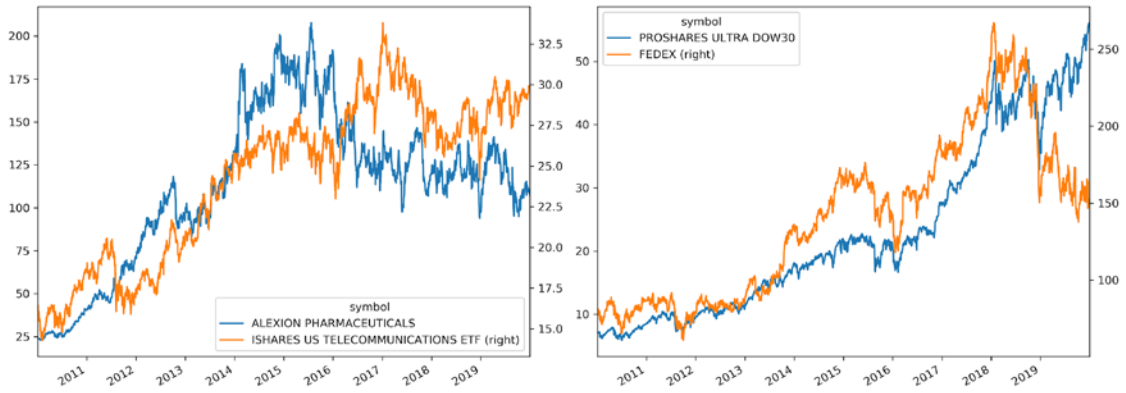

Figure 9.15: Price series for two selected pairs over the sample period

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, y], axis=1))

            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.fill()
        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 lookahead 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.