06_statistical_arbitrage_with_cointegrated_pairs

September 29, 2021

1 Statistical arbitrage with Cointegration

1.1 Pairs Trading & Statistical Arbitrage

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.

1.2 Pairs Trading in Practice

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 thespread 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 book outlines the key differences between them; the notebook dives into an example application.

1.3 Imports & Settings

[1]:
```
import warnings
warnings.filterwarnings('ignore')
```

[2]:
```
from collections import Counter

from time import time
```

```python
from pathlib import Path

import numpy as np
import pandas as pd

from pykalman import KalmanFilter
from statsmodels.tsa.stattools import coint
from statsmodels.tsa.vector_ar.vecm import coint_johansen
from statsmodels.tsa.api import VAR

import matplotlib.pyplot as plt
import seaborn as sns
```

[3]:
```python
idx = pd.IndexSlice
sns.set_style('whitegrid')
```

[5]:
```python
def format_time(t):
    m_, s = divmod(t, 60)
    h, m = divmod(m_, 60)
    return f'{h:>02.0f}:{m:>02.0f}:{s:>02.0f}'
```

1.3.1 Johansen Test Critical Values

[6]:
```python
critical_values = {0: {.9: 13.4294, .95: 15.4943, .99: 19.9349},
                   1: {.9: 2.7055, .95: 3.8415, .99: 6.6349}}
```

[7]:
```python
trace0_cv = critical_values[0][.95] # critical value for 0 cointegration␣
↪relationships
trace1_cv = critical_values[1][.95] # critical value for 1 cointegration␣
↪relationship
```

1.4 Load Data

[39]:
```python
DATA_PATH = Path('..', 'data')
STORE = DATA_PATH / 'assets.h5'
```

1.4.1 Get backtest prices

Combine OHLCV prices for relevant stock and ETF tickers.

[57]:
```python
def get_backtest_prices():
    with pd.HDFStore('data.h5') as store:
        tickers = store['tickers']

    with pd.HDFStore(STORE) as store:
        prices = (pd.concat([
            store['stooq/us/nyse/stocks/prices'],
```

```
                store['stooq/us/nyse/etfs/prices'],
                store['stooq/us/nasdaq/etfs/prices'],
                store['stooq/us/nasdaq/stocks/prices']])
                      .sort_index()
                      .loc[idx[tickers.index, '2016':'2019'], :])
    print(prices.info(null_counts=True))
    prices.to_hdf('backtest.h5', 'prices')
    tickers.to_hdf('backtest.h5', 'tickers')
```

[58]:
```
get_backtest_prices()
```

```
<class 'pandas.core.frame.DataFrame'>
MultiIndex: 312863 entries, ('AA.US', Timestamp('2016-01-04 00:00:00')) to
('YUM.US', Timestamp('2019-12-31 00:00:00'))
Data columns (total 5 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   open    312863 non-null  float64
 1   high    312863 non-null  float64
 2   low     312863 non-null  float64
 3   close   312863 non-null  float64
 4   volume  312863 non-null  int64
dtypes: float64(4), int64(1)
memory usage: 13.2+ MB
None
```

1.4.2 Load Stock Prices

[11]:
```
stocks = pd.read_hdf('data.h5', 'stocks/close').loc['2015':]
stocks.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1258 entries, 2015-01-02 to 2019-12-31
Columns: 172 entries, AA.US to YUM.US
dtypes: float64(172)
memory usage: 1.7 MB
```

1.4.3 Load ETF Data

[12]:
```
etfs = pd.read_hdf('data.h5', 'etfs/close').loc['2015':]
etfs.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1258 entries, 2015-01-02 to 2019-12-31
Columns: 139 entries, AAXJ.US to YCS.US
dtypes: float64(139)
memory usage: 1.3 MB
```

1.4.4 Load Ticker Dictionary

[13]:
```python
names = pd.read_hdf('data.h5', 'tickers').to_dict()
```

[14]:
```python
pd.Series(names).count()
```

[14]:
```
311
```

1.5 Precompute Cointegration

[15]:
```python
def test_cointegration(etfs, stocks, test_end, lookback=2):
    start = time()
    results = []
    test_start = test_end - pd.DateOffset(years=lookback) + pd.
 ↪DateOffset(days=1)
    etf_tickers = etfs.columns.tolist()
    etf_data = etfs.loc[str(test_start):str(test_end)]

    stock_tickers = stocks.columns.tolist()
    stock_data = stocks.loc[str(test_start):str(test_end)]
    n = len(etf_tickers) * len(stock_tickers)
    j = 0
    for i, s1 in enumerate(etf_tickers, 1):
        for s2 in stock_tickers:
            j += 1
            if j % 1000 == 0:
                print(f'\t{j:5,.0f} ({j/n:3.1%}) | {time() - start:.2f}')
            df = etf_data.loc[:, [s1]].dropna().join(stock_data.loc[:, [s2]].
 ↪dropna(), how='inner')
            with warnings.catch_warnings():
                warnings.simplefilter('ignore')
                var = VAR(df)
                lags = var.select_order()
                result = [test_end, s1, s2]
                order = lags.selected_orders['aic']
                result += [coint(df[s1], df[s2], trend='c')[1], coint(df[s2],␣
 ↪df[s1], trend='c')[1]]

            cj = coint_johansen(df, det_order=0, k_ar_diff=order)
            result += (list(cj.lr1) + list(cj.lr2) + list(cj.evec[:, cj.
 ↪ind[0]]))
            results.append(result)
    return results
```

1.5.1 Define Test Periods

[16]:
```python
dates = stocks.loc['2016-12':'2019-6'].resample('Q').last().index
dates
```

[16]:
```
DatetimeIndex(['2016-12-31', '2017-03-31', '2017-06-30', '2017-09-30',
               '2017-12-31', '2018-03-31', '2018-06-30', '2018-09-30',
               '2018-12-31', '2019-03-31', '2019-06-30'],
              dtype='datetime64[ns]', name='date', freq='Q-DEC')
```

1.5.2 Run Tests

[17]:
```python
test_results = []
columns = ['test_end', 's1', 's2', 'eg1', 'eg2',
           'trace0', 'trace1', 'eig0', 'eig1', 'w1', 'w2']

for test_end in dates:
    print(test_end)
    result = test_cointegration(etfs, stocks, test_end=test_end)
    test_results.append(pd.DataFrame(result, columns=columns))

pd.concat(test_results).to_hdf('backtest.h5', 'cointegration_test')
```

```
2016-12-31 00:00:00
        1,000 (4.2%) | 54.32
        2,000 (8.4%) | 105.25
        3,000 (12.5%) | 156.85
        4,000 (16.7%) | 208.33
        5,000 (20.9%) | 259.67
        6,000 (25.1%) | 310.20
        7,000 (29.3%) | 361.73
        8,000 (33.5%) | 415.96
        9,000 (37.6%) | 467.39
        10,000 (41.8%) | 519.14
        11,000 (46.0%) | 571.14
        12,000 (50.2%) | 623.57
        13,000 (54.4%) | 679.03
        14,000 (58.6%) | 731.01
        15,000 (62.7%) | 786.51
        16,000 (66.9%) | 838.69
        17,000 (71.1%) | 891.46
        18,000 (75.3%) | 947.83
        19,000 (79.5%) | 999.86
        20,000 (83.7%) | 1051.32
        21,000 (87.8%) | 1102.79
        22,000 (92.0%) | 1155.83
        23,000 (96.2%) | 1210.67
2017-03-31 00:00:00
```

```
        1,000 (4.2%) | 51.45
        2,000 (8.4%) | 103.60
        3,000 (12.5%) | 154.91
        4,000 (16.7%) | 207.16
        5,000 (20.9%) | 259.57
        6,000 (25.1%) | 313.52
        7,000 (29.3%) | 367.18
        8,000 (33.5%) | 419.41
        9,000 (37.6%) | 472.77
        10,000 (41.8%) | 524.24
        11,000 (46.0%) | 576.73
        12,000 (50.2%) | 628.75
        13,000 (54.4%) | 680.14
        14,000 (58.6%) | 732.15
        15,000 (62.7%) | 788.75
        16,000 (66.9%) | 839.75
        17,000 (71.1%) | 891.21
        18,000 (75.3%) | 943.21
        19,000 (79.5%) | 994.23
        20,000 (83.7%) | 1045.09
        21,000 (87.8%) | 1099.81
        22,000 (92.0%) | 1152.24
        23,000 (96.2%) | 1204.35
2017-06-30 00:00:00
        1,000 (4.2%) | 60.72
        2,000 (8.4%) | 112.32
        3,000 (12.5%) | 163.92
        4,000 (16.7%) | 216.93
        5,000 (20.9%) | 268.05
        6,000 (25.1%) | 319.51
        7,000 (29.3%) | 371.52
        8,000 (33.5%) | 421.85
        9,000 (37.6%) | 477.16
        10,000 (41.8%) | 527.80
        11,000 (46.0%) | 578.79
        12,000 (50.2%) | 634.85
        13,000 (54.4%) | 685.60
        14,000 (58.6%) | 742.40
        15,000 (62.7%) | 797.58
        16,000 (66.9%) | 852.54
        17,000 (71.1%) | 904.22
        18,000 (75.3%) | 954.77
        19,000 (79.5%) | 1006.83
        20,000 (83.7%) | 1057.93
        21,000 (87.8%) | 1114.17
        22,000 (92.0%) | 1165.59
        23,000 (96.2%) | 1217.66
2017-09-30 00:00:00
```

```
        1,000 (4.2%) | 52.47
        2,000 (8.4%) | 104.40
        3,000 (12.5%) | 157.41
        4,000 (16.7%) | 208.44
        5,000 (20.9%) | 260.18
        6,000 (25.1%) | 313.08
        7,000 (29.3%) | 366.05
        8,000 (33.5%) | 418.07
        9,000 (37.6%) | 470.27
        10,000 (41.8%) | 522.26
        11,000 (46.0%) | 574.02
        12,000 (50.2%) | 625.88
        13,000 (54.4%) | 677.30
        14,000 (58.6%) | 729.59
        15,000 (62.7%) | 780.49
        16,000 (66.9%) | 836.47
        17,000 (71.1%) | 888.55
        18,000 (75.3%) | 940.25
        19,000 (79.5%) | 992.75
        20,000 (83.7%) | 1044.53
        21,000 (87.8%) | 1096.46
        22,000 (92.0%) | 1147.74
        23,000 (96.2%) | 1199.66
2017-12-31 00:00:00
        1,000 (4.2%) | 52.34
        2,000 (8.4%) | 103.74
        3,000 (12.5%) | 155.49
        4,000 (16.7%) | 205.97
        5,000 (20.9%) | 247.07
        6,000 (25.1%) | 287.80
        7,000 (29.3%) | 328.58
        8,000 (33.5%) | 369.47
        9,000 (37.6%) | 410.48
        10,000 (41.8%) | 451.58
        11,000 (46.0%) | 492.52
        12,000 (50.2%) | 533.40
        13,000 (54.4%) | 574.31
        14,000 (58.6%) | 615.45
        15,000 (62.7%) | 656.33
        16,000 (66.9%) | 697.11
        17,000 (71.1%) | 737.81
        18,000 (75.3%) | 778.57
        19,000 (79.5%) | 819.32
        20,000 (83.7%) | 860.00
        21,000 (87.8%) | 901.04
        22,000 (92.0%) | 941.77
        23,000 (96.2%) | 982.65
2018-03-31 00:00:00
```

```
        1,000 (4.2%) | 40.97
        2,000 (8.4%) | 81.99
        3,000 (12.5%) | 123.05
        4,000 (16.7%) | 164.06
        5,000 (20.9%) | 205.03
        6,000 (25.1%) | 246.91
        7,000 (29.3%) | 288.14
        8,000 (33.5%) | 329.00
        9,000 (37.6%) | 369.95
        10,000 (41.8%) | 411.02
        11,000 (46.0%) | 451.95
        12,000 (50.2%) | 493.00
        13,000 (54.4%) | 533.98
        14,000 (58.6%) | 574.80
        15,000 (62.7%) | 615.67
        16,000 (66.9%) | 656.63
        17,000 (71.1%) | 697.80
        18,000 (75.3%) | 738.80
        19,000 (79.5%) | 779.77
        20,000 (83.7%) | 820.54
        21,000 (87.8%) | 861.56
        22,000 (92.0%) | 902.40
        23,000 (96.2%) | 943.44
2018-06-30 00:00:00
        1,000 (4.2%) | 40.92
        2,000 (8.4%) | 81.85
        3,000 (12.5%) | 122.98
        4,000 (16.7%) | 164.03
        5,000 (20.9%) | 205.11
        6,000 (25.1%) | 246.25
        7,000 (29.3%) | 287.36
        8,000 (33.5%) | 328.38
        9,000 (37.6%) | 369.17
        10,000 (41.8%) | 409.99
        11,000 (46.0%) | 450.90
        12,000 (50.2%) | 491.70
        13,000 (54.4%) | 532.74
        14,000 (58.6%) | 573.86
        15,000 (62.7%) | 614.66
        16,000 (66.9%) | 655.39
        17,000 (71.1%) | 696.38
        18,000 (75.3%) | 737.29
        19,000 (79.5%) | 778.26
        20,000 (83.7%) | 819.22
        21,000 (87.8%) | 860.34
        22,000 (92.0%) | 901.37
        23,000 (96.2%) | 942.38
2018-09-30 00:00:00
```

```
        1,000 (4.2%) | 41.14
        2,000 (8.4%) | 82.08
        3,000 (12.5%) | 123.01
        4,000 (16.7%) | 163.90
        5,000 (20.9%) | 204.80
        6,000 (25.1%) | 245.65
        7,000 (29.3%) | 286.63
        8,000 (33.5%) | 327.57
        9,000 (37.6%) | 368.46
        10,000 (41.8%) | 409.37
        11,000 (46.0%) | 450.29
        12,000 (50.2%) | 491.42
        13,000 (54.4%) | 532.83
        14,000 (58.6%) | 574.23
        15,000 (62.7%) | 615.76
        16,000 (66.9%) | 657.23
        17,000 (71.1%) | 698.85
        18,000 (75.3%) | 740.13
        19,000 (79.5%) | 781.41
        20,000 (83.7%) | 822.40
        21,000 (87.8%) | 863.44
        22,000 (92.0%) | 904.50
        23,000 (96.2%) | 945.50
2018-12-31 00:00:00
        1,000 (4.2%) | 40.96
        2,000 (8.4%) | 81.97
        3,000 (12.5%) | 123.01
        4,000 (16.7%) | 164.11
        5,000 (20.9%) | 204.93
        6,000 (25.1%) | 245.70
        7,000 (29.3%) | 286.53
        8,000 (33.5%) | 327.35
        9,000 (37.6%) | 368.21
        10,000 (41.8%) | 409.12
        11,000 (46.0%) | 450.09
        12,000 (50.2%) | 490.97
        13,000 (54.4%) | 532.01
        14,000 (58.6%) | 573.02
        15,000 (62.7%) | 614.00
        16,000 (66.9%) | 655.01
        17,000 (71.1%) | 695.99
        18,000 (75.3%) | 736.99
        19,000 (79.5%) | 777.94
        20,000 (83.7%) | 818.69
        21,000 (87.8%) | 859.55
        22,000 (92.0%) | 900.84
        23,000 (96.2%) | 941.59
2019-03-31 00:00:00
```

```
        1,000 (4.2%) | 40.96
        2,000 (8.4%) | 82.02
        3,000 (12.5%) | 122.92
        4,000 (16.7%) | 163.86
        5,000 (20.9%) | 204.86
        6,000 (25.1%) | 245.87
        7,000 (29.3%) | 286.86
        8,000 (33.5%) | 327.88
        9,000 (37.6%) | 368.71
        10,000 (41.8%) | 409.61
        11,000 (46.0%) | 450.55
        12,000 (50.2%) | 491.47
        13,000 (54.4%) | 532.39
        14,000 (58.6%) | 573.34
        15,000 (62.7%) | 614.34
        16,000 (66.9%) | 655.35
        17,000 (71.1%) | 696.43
        18,000 (75.3%) | 737.54
        19,000 (79.5%) | 778.59
        20,000 (83.7%) | 819.64
        21,000 (87.8%) | 860.45
        22,000 (92.0%) | 901.25
        23,000 (96.2%) | 942.15
2019-06-30 00:00:00
        1,000 (4.2%) | 40.89
        2,000 (8.4%) | 81.79
        3,000 (12.5%) | 122.80
        4,000 (16.7%) | 163.57
        5,000 (20.9%) | 204.28
        6,000 (25.1%) | 245.06
        7,000 (29.3%) | 285.77
        8,000 (33.5%) | 326.56
        9,000 (37.6%) | 367.20
        10,000 (41.8%) | 407.91
        11,000 (46.0%) | 448.76
        12,000 (50.2%) | 489.38
        13,000 (54.4%) | 530.03
        14,000 (58.6%) | 570.77
        15,000 (62.7%) | 611.43
        16,000 (66.9%) | 652.08
        17,000 (71.1%) | 692.72
        18,000 (75.3%) | 733.67
        19,000 (79.5%) | 774.62
        20,000 (83.7%) | 815.65
        21,000 (87.8%) | 856.57
        22,000 (92.0%) | 897.49
        23,000 (96.2%) | 938.58
```

Reload Test Results

[18]:
```
test_results = pd.read_hdf('backtest.h5', 'cointegration_test')
test_results.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 262988 entries, 0 to 23907
Data columns (total 11 columns):
 #   Column    Non-Null Count   Dtype
---  ------    --------------   -----
 0   test_end  262988 non-null  datetime64[ns]
 1   s1        262988 non-null  object
 2   s2        262988 non-null  object
 3   eg1       262988 non-null  float64
 4   eg2       262988 non-null  float64
 5   trace0    262988 non-null  float64
 6   trace1    262988 non-null  float64
 7   eig0      262988 non-null  float64
 8   eig1      262988 non-null  float64
 9   w1        262988 non-null  float64
 10  w2        262988 non-null  float64
dtypes: datetime64[ns](1), float64(8), object(2)
memory usage: 24.1+ MB
```

1.6 Identify Cointegrated Pairs

1.6.1 Significant Johansen Trace Statistic

[19]:
```
test_results['joh_sig'] = ((test_results.trace0 > trace0_cv) &
                           (test_results.trace1 > trace1_cv))
```

[20]:
```
test_results.joh_sig.value_counts(normalize=True)
```

[20]:
```
False    0.947211
True     0.052789
Name: joh_sig, dtype: float64
```

1.6.2 Significant Engle Granger Test

[21]:
```
test_results['eg'] = test_results[['eg1', 'eg2']].min(axis=1)
test_results['s1_dep'] = test_results.eg1 < test_results.eg2
test_results['eg_sig'] = (test_results.eg < .05)
```

[22]:
```
test_results.eg_sig.value_counts(normalize=True)
```

[22]:
```
False    0.91157
True     0.08843
Name: eg_sig, dtype: float64
```

1.6.3 Comparison Engle-Granger vs Johansen

[23]:
```
test_results['coint'] = (test_results.eg_sig & test_results.joh_sig)
test_results.coint.value_counts(normalize=True)
```

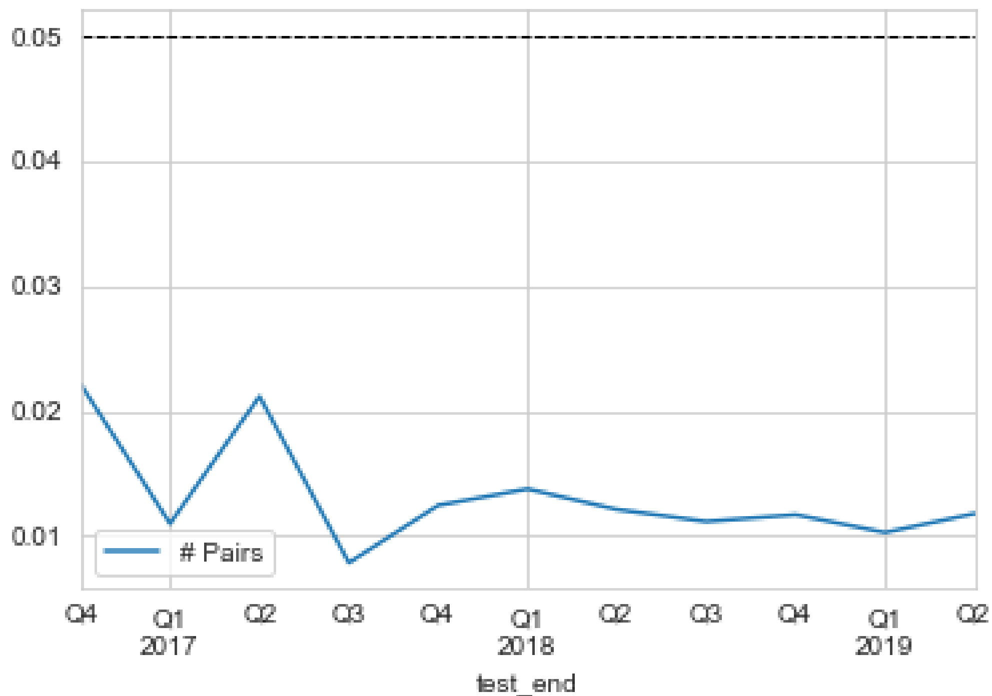
[23]:
```
False    0.986775
True     0.013225
Name: coint, dtype: float64
```

[24]:
```
test_results = test_results.drop(['eg1', 'eg2', 'trace0', 'trace1', 'eig0',␣
↪'eig1'], axis=1)
test_results.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 262988 entries, 0 to 23907
Data columns (total 10 columns):
 #   Column    Non-Null Count   Dtype
---  ------    --------------   -----
 0   test_end  262988 non-null  datetime64[ns]
 1   s1        262988 non-null  object
 2   s2        262988 non-null  object
 3   w1        262988 non-null  float64
 4   w2        262988 non-null  float64
 5   joh_sig   262988 non-null  bool
 6   eg        262988 non-null  float64
 7   s1_dep    262988 non-null  bool
 8   eg_sig    262988 non-null  bool
 9   coint     262988 non-null  bool
dtypes: bool(4), datetime64[ns](1), float64(3), object(2)
memory usage: 15.0+ MB
```

1.6.4 Comparison

[25]:
```
ax = test_results.groupby('test_end').coint.mean().to_frame('# Pairs').plot()
ax.axhline(.05, lw=1, ls='--', c='k');
```



1.6.5 Select Candidate Pairs

[26]:
```
def select_candidate_pairs(data):
    candidates = data[data.joh_sig | data.eg_sig]
    candidates['y'] = candidates.apply(lambda x: x.s1 if x.s1_dep else x.s2,␣
 ↪axis=1)
    candidates['x'] = candidates.apply(lambda x: x.s2 if x.s1_dep else x.s1,␣
 ↪axis=1)
    return candidates.drop(['s1_dep', 's1', 's2'], axis=1)
```

[27]:
```
candidates = select_candidate_pairs(test_results)
```

[28]:
```
candidates.to_hdf('backtest.h5', 'candidates')
```

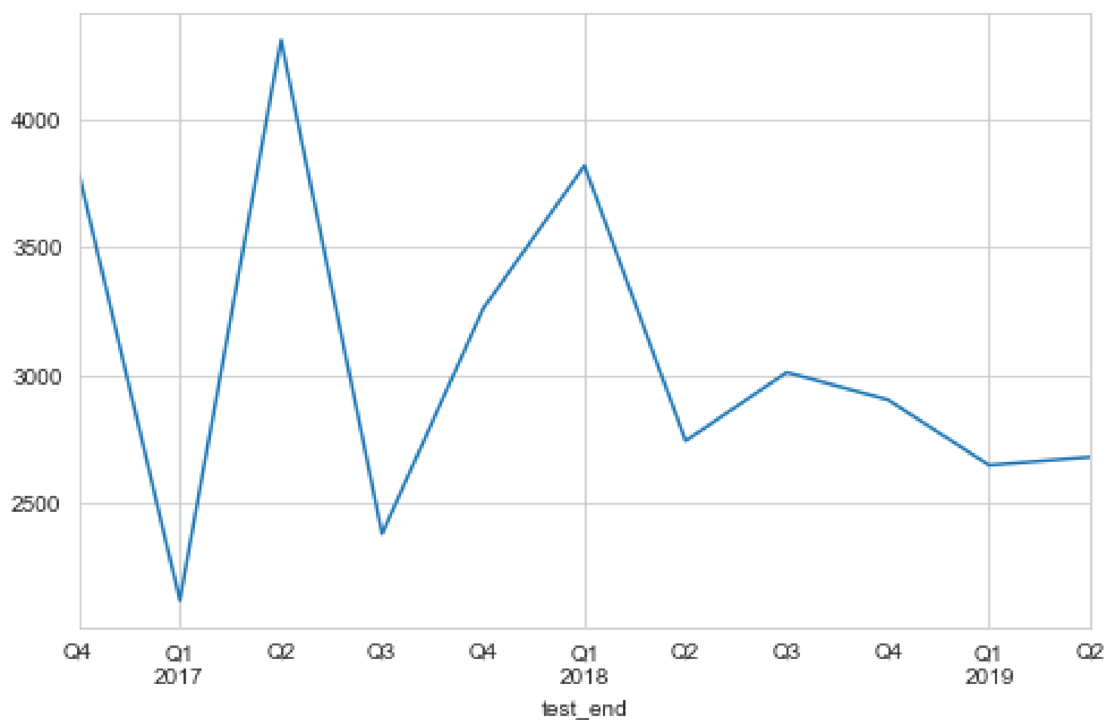
[29]:
```
candidates = pd.read_hdf('backtest.h5', 'candidates')
candidates.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 33661 entries, 7 to 23906
Data columns (total 9 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   test_end  33661 non-null  datetime64[ns]
 1   w1        33661 non-null  float64
```

```
 2   w2         33661 non-null  float64
 3   joh_sig    33661 non-null  bool
 4   eg         33661 non-null  float64
 5   eg_sig     33661 non-null  bool
 6   coint      33661 non-null  bool
 7   y          33661 non-null  object
 8   x          33661 non-null  object
dtypes: bool(3), datetime64[ns](1), float64(3), object(2)
memory usage: 1.9+ MB
```

Candidates over Time

[30]:
```
candidates.groupby('test_end').size().plot(figsize=(8, 5))
```



Most Common Pairs

[31]:
```
with pd.HDFStore('data.h5') as store:
    print(store.info())
    tickers = store['tickers']
```

```
<class 'pandas.io.pytables.HDFStore'>
File path: data.h5
/etfs/close                  frame         (shape->[2516,139])
/stocks/close                frame         (shape->[2516,172])
/tickers                     series        (shape->[1])
```

[32]:
```python
with pd.HDFStore('backtest.h5') as store:
    print(store.info())
```

```
<class 'pandas.io.pytables.HDFStore'>
File path: backtest.h5
/candidates                    frame        (shape->[33661,9])
/cointegration_test            frame        (shape->[262988,11])
```

[33]:
```python
counter = Counter()
for s1, s2 in zip(candidates[candidates.joh_sig & candidates.eg_sig].y,
                  candidates[candidates.joh_sig & candidates.eg_sig].x):
    if s1 > s2:
        counter[(s2, s1)] += 1
    else:
        counter[(s1, s2)] += 1
```

[34]:
```python
most_common_pairs = pd.DataFrame(counter.most_common(10))
most_common_pairs = pd.DataFrame(most_common_pairs[0].values.tolist(),␣
 ↪columns=['s1', 's2'])
most_common_pairs
```

[34]:

	s1	s2
0	T.US	VOX.US
1	FXF.US	MDLZ.US
2	FXF.US	NOV.US
3	FXF.US	RIG.US
4	AMJ.US	MDLZ.US
5	DIG.US	MDLZ.US
6	DJP.US	MDLZ.US
7	ERX.US	MDLZ.US
8	FXN.US	MDLZ.US
9	IYE.US	MDLZ.US

[59]:
```python
with pd.HDFStore('backtest.h5') as store:
    prices = store['prices'].close.unstack('ticker').ffill(limit=5)
    tickers = store['tickers'].to_dict()
```

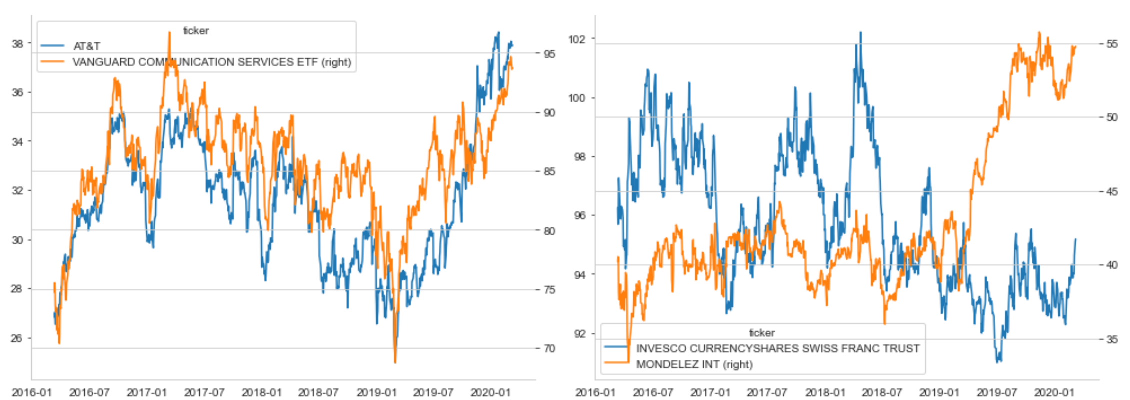
[60]:
```python
cnt = pd.Series(counter).reset_index()
cnt.columns = ['s1', 's2', 'n']
cnt['name1'] = cnt.s1.map(tickers)
cnt['name2'] = cnt.s2.map(tickers)
cnt.nlargest(10, columns='n')
```

[60]:
```
           s1       s2  n                                  name1  \
1352     T.US   VOX.US  6                                   AT&T
384    FXF.US  MDLZ.US  5  INVESCO CURRENCYSHARES SWISS FRANC TRUST
388    FXF.US   NOV.US  5  INVESCO CURRENCYSHARES SWISS FRANC TRUST
```

```
391   FXF.US   RIG.US  5  INVESCO CURRENCYSHARES SWISS FRANC TRUST
532   AMJ.US  MDLZ.US  5           JPMORGAN ALERIAN MLP INDEX ETN
547   DIG.US  MDLZ.US  5                 PROSHARES ULTRA OIL & GAS
549   DJP.US  MDLZ.US  5    IPATH BLOOMBERG COMMODITY INDEX TR ETN
571   ERX.US  MDLZ.US  5      DIREXION DAILY ENERGY BULL 2X SHARES
630   FXN.US  MDLZ.US  5          FIRST TRUST ENERGY ALPHADEX FUND
644   IYE.US  MDLZ.US  5                     ISHARES US ENERGY ETF

                                    name2
1352  VANGUARD COMMUNICATION SERVICES ETF
384                          MONDELEZ INT
388                 NATIONAL OILWELL VARCO
391                            TRANSOCEAN
532                          MONDELEZ INT
547                          MONDELEZ INT
549                          MONDELEZ INT
571                          MONDELEZ INT
630                          MONDELEZ INT
644                          MONDELEZ INT
```

[63]:

```
fig, axes = plt.subplots(ncols=2, figsize=(14, 5))
for i in [0, 1]:
    s1, s2 = most_common_pairs.at[i, 's1'], most_common_pairs.at[i, 's2']
    prices.loc[:, [s1, s2]].rename(columns=tickers).
 ↪plot(secondary_y=tickers[s2],
                                                         ax=axes[i],
                                                         rot=0)

    axes[i].grid(False)
    axes[i].set_xlabel('')

sns.despine()
fig.tight_layout()
```



1.7 Get Entry and Exit Dates

1.7.1 Smooth prices using Kalman filter

[64]:
```
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)
```

[65]:
```
smoothed_prices = prices.apply(KFSmoother)
smoothed_prices.to_hdf('tmp.h5', 'smoothed')
```

[66]:
```
smoothed_prices = pd.read_hdf('tmp.h5', 'smoothed')
```

1.7.2 Compute rolling hedge ratio using Kalman Filter

[67]:
```
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
```

1.7.3 Estimate mean reversion half life

[68]:
```
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)
```

1.7.4 Compute Spread & Bollinger Bands

[69]:
```
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):
            if i % 1000 == 0:
                msg = f'{i:5.0f} | {time() - start_iteration:7.1f} | {time() -␣
↪start:10.1f}'
                print(msg)
            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
```

[70]:
```
candidates = pd.read_hdf('backtest.h5', 'candidates')
candidates.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 33661 entries, 7 to 23906
Data columns (total 9 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   test_end  33661 non-null  datetime64[ns]
 1   w1        33661 non-null  float64
 2   w2        33661 non-null  float64
 3   joh_sig   33661 non-null  bool
 4   eg        33661 non-null  float64
 5   eg_sig    33661 non-null  bool
 6   coint     33661 non-null  bool
 7   y         33661 non-null  object
 8   x         33661 non-null  object
dtypes: bool(3), datetime64[ns](1), float64(3), object(2)
memory usage: 1.9+ MB
```

[71]:

```
pairs, half_lives = get_spread(candidates, smoothed_prices)
```

```
2016-12-31 3793
 1000 |   180.5 |      180.5
 2000 |   360.7 |      360.7
 3000 |   536.6 |      536.6
2017-03-31 2118
 1000 |   206.2 |      883.1
 2000 |   415.7 |     1092.6
2017-06-30 4311
 1000 |   244.6 |     1363.1
 2000 |   486.8 |     1605.3
 3000 |   750.6 |     1869.1
 4000 |   989.5 |     2108.0
2017-09-30 2379
 1000 |   271.5 |     2481.2
 2000 |   538.0 |     2747.7
2017-12-31 3260
 1000 |   706.0 |     3554.9
 2000 |  2552.6 |     5401.5
 3000 |  2927.6 |     5776.4
2018-03-31 3819
 1000 |   354.2 |     6228.5
 2000 |   652.1 |     6526.5
 3000 |   954.5 |     6828.9
2018-06-30 2742
 1000 |   299.7 |     7372.7
 2000 |   595.4 |     7668.4
2018-09-30 3010
 1000 |   292.7 |     8182.6
 2000 |   588.5 |     8478.3
```

```
 3000 |    883.9 |     8773.7
2018-12-31 2903
 1000 |    292.5 |     9069.4
 2000 |    582.5 |     9359.4
2019-03-31 2647
 1000 |    295.1 |     9920.7
 2000 |    588.3 |    10213.9
2019-06-30 2679
 1000 |    290.2 |    10693.5
 2000 |    585.2 |    10988.6
```

1.7.5 Collect Results

Half Lives

[72]:
```python
hl = pd.DataFrame(half_lives, columns=['test_end', 's1', 's2', 'half_life'])
hl.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 33661 entries, 0 to 33660
Data columns (total 4 columns):
 #   Column     Non-Null Count  Dtype
---  ------     --------------  -----
 0   test_end   33661 non-null  datetime64[ns]
 1   s1         33661 non-null  object
 2   s2         33661 non-null  object
 3   half_life  33661 non-null  int64
dtypes: datetime64[ns](1), int64(1), object(2)
memory usage: 1.0+ MB
```

[73]:
```python
hl.half_life.describe()
```

[73]:
```
count    33661.000000
mean        25.199519
std         10.223437
min          1.000000
25%         20.000000
50%         24.000000
75%         28.000000
max       1057.000000
Name: half_life, dtype: float64
```

[74]:
```python
hl.to_hdf('backtest.h5', 'half_lives')
```

Pair Data

[75]:
```python
pair_data = pd.concat(pairs)
pair_data.info(null_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 4229473 entries, 2017-01-03 to 2019-12-31
Data columns (total 7 columns):
 #   Column        Non-Null Count    Dtype
---  ------        --------------    -----
 0   hedge_ratio   4229473 non-null  float64
 1   spread        4229473 non-null  float64
 2   z_score       4229473 non-null  float64
 3   s1            4229473 non-null  object
 4   s2            4229473 non-null  object
 5   period        4229473 non-null  int64
 6   pair          4229473 non-null  int64
dtypes: float64(3), int64(2), object(2)
memory usage: 258.1+ MB
```

[76]:
```
pair_data.to_hdf('backtest.h5', 'pair_data')
```

[77]:
```
pair_data = pd.read_hdf('backtest.h5', 'pair_data')
```

1.7.6 Identify Long & Short Entry and Exit Dates

[78]:
```
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())
        if not isinstance(trades, pd.Series):
            continue
```

```python
        try:
            trades.loc[trades < 0] += 2
        except:
            print(type(trades))
            print(trades)
            print(pair.z_score.describe())
            break

        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
```

[79]:
```python
pair_trades = get_trades(pair_data)
```

```
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
```

1500
1600
1700
1800
1900
2000
2100
2200
2300
2400
2500
2600
2700
2800
2900
3000
3100
3200
3300
3400
3500
3600
3700
3800
3900
4000
4100
4200
4300
4400
4500
4600
4700
4800
4900
5000
5100
5200
5300
5400
5500
5600
5700
5800
5900
6000
6100
6200

6300
6400
6500
6600
6700
6800
6900
7000
7100
7200
7300
7400
7500
7600
7700
7800
7900
8000
8100
8200
8300
8400
8500
8600
8700
8800
8900
9000
9100
9200
9300
9400
9500
9600
9700
9800
9900
10000
10100
10200
10300
10400
10500
10600
10700
10800
10900
11000

11100
11200
11300
11400
11500
11600
11700
11800
11900
12000
12100
12200
12300
12400
12500
12600
12700
12800
12900
13000
13100
13200
13300
13400
13500
13600
13700
13800
13900
14000
14100
14200
14300
14400
14500
14600
14700
14800
14900
15000
15100
15200
15300
15400
15500
15600
15700
15800

15900
16000
16100
16200
16300
16400
16500
16600
16700
16800
16900
17000
17100
17200
17300
17400
17500
17600
17700
17800
17900
18000
18100
18200
18300
18400
18500
18600
18700
18800
18900
19000
19100
19200
19300
19400
19500
19600
19700
19800
19900
20000
20100
20200
20300
20400
20500
20600

20700
20800
20900
21000
21100
21200
21300
21400
21500
21600
21700
21800
21900
22000
22100
22200
22300
22400
22500
22600
22700
22800
22900
23000
23100
23200
23300
23400
23500
23600
23700
23800
23900
24000
24100
24200
24300
24400
24500
24600
24700
24800
24900
25000
25100
25200
25300
25400

25500
25600
25700
25800
25900
26000
26100
26200
26300
26400
26500
26600
26700
26800
26900
27000
27100
27200
27300
27400
27500
27600
27700
27800
27900
28000
28100
28200
28300
28400
28500
28600
28700
28800
28900
29000
29100
29200
29300
29400
29500
29600
29700
29800
29900
30000
30100
30200

```
30300
30400
30500
30600
30700
30800
30900
31000
31100
31200
31300
31400
31500
31600
31700
31800
31900
32000
32100
32200
32300
32400
32500
32600
32700
32800
32900
33000
33100
33200
33300
33400
33500
33600
```

[80]:
```python
pair_trade_data = pd.concat(pair_trades)
pair_trade_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 134450 entries, 2017-01-03 to 2019-10-04
Data columns (total 6 columns):
 #   Column       Non-Null Count   Dtype
---  ------       --------------   -----
 0   s1           134450 non-null  object
 1   s2           134450 non-null  object
 2   hedge_ratio  134450 non-null  float64
 3   period       134450 non-null  int64
 4   pair         134450 non-null  int64
```

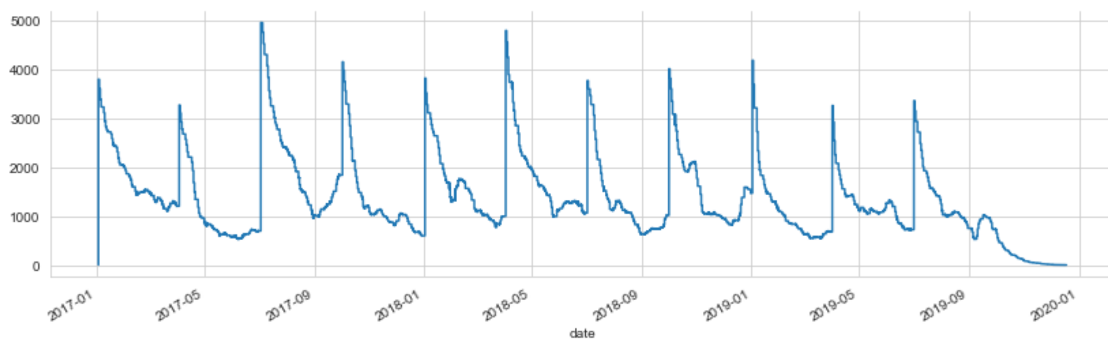
```
 5   side         134450 non-null  int64
dtypes: float64(1), int64(3), object(2)
memory usage: 7.2+ MB
```

[81]:
```
pair_trade_data.head()
```

[81]:

	s1	s2	hedge_ratio	period	pair	side
date						
2017-01-03	AA.US	ACWI.US	-0.533861	1	16	-1
2017-01-12	AA.US	ACWI.US	-0.533861	1	16	0
2017-01-03	AA.US	ACWX.US	-0.799916	1	54	-1
2017-01-12	AA.US	ACWX.US	-0.799916	1	54	0
2017-01-03	AA.US	DEM.US	-0.896395	1	376	-1

[84]:
```
trades = pair_trade_data['side'].copy()
trades.loc[trades != 0] = 1
trades.loc[trades == 0] = -1
trades.sort_index().cumsum().plot(figsize=(14, 4))
sns.despine()
```



[83]:
```
pair_trade_data.to_hdf('backtest.h5', 'pair_trades')
```