Pairs Trading: Building a Backtesting Environment with Python

Introduction

Statistical arbitrage is a class of trading strategies that profit from exploiting what are believed to be market inefficiencies. These inefficiencies are determined through statistical and econometric techniques. Note that the arbitrage part should by no means suggest a riskless strategy, rather a strategy in which risk is statistically assessed.

One of the earliest and simplest types of statistical arbitrage is pairs trading. In short, pairs trading is a strategy in which two securities that have been cointegrated over the years are identified and short-term deviation from their long-term equilibrium creates trading opportunities.

In this article, we show how to build a backtesting environment with Python, which you can find in this GitHub repository. For a more extensive explanation of pairs trading please refer to this article. Before diving into the implementation of the code, it is important to deeply understand how to test a hypothetical statistical relationship. To do that, we recall the concepts of stationarity, correlation and cointegration.

A stationary process is characterized by constant mean, variance and autocorrelation. Nonstationary time series do not have the tendency to revert to their mean and vice versa.

Correlation is a measure of the linear relationship between stationary variables. As a consequence, when dealing with non-stationary variables, correlation is often spurious. For this reason, it is good practice to avoid calculating correlation coefficients or regressing two non-stationary processes. However, most of the processes that we deal with in the real world are non-stationary. This leads to the necessity to study them despite some imperfections.

To do this, cointegration comes into play. Suppose two time series X and Y are non-stationary and integrated of order d, then they are cointegrated if it is possible to find a linear combination $Y - \beta X$ that is integrated of order less than d. In our case, if two securities are non-stationary time series of order 1 and if a linear combination between two securities (spread) is tested to be integrated of order 0, in other words stationary, then we define the two securities as cointegrated.

While the order of integration is usually tested through the Augmented Dickey-Fuller test, to test cointegration Engle-Granger seems the perfect choice. After having determined a certain coefficient β by regressing X on Y, the residuals (spread) are tested for stationarity through an Augmented Dickey-Fuller test. In simple terms, the Dickey-Fuller test is a procedure in which the coefficient of an autoregressive process is tested for its significance. If the coefficient is 1, the process is a random walk and the unit root (non-stationarity) cannot be rejected. On the other hand, if the coefficient is significantly less than 1, then the unit root is rejected and the process is considered to be stationary.

To summarize, cointegration implies that the variables share similar stochastic trends, and therefore their residuals follow a stationary process. Two cointegrated variables may not necessarily move in the same direction (correlation), but the distance between them will remain constant over time.

Moreover, in selecting our pairs, we will also consider the Hurst exponent, which measures the long-term memory of time series. By identifying the persistence of trends, it distinguishes random from non-random systems. A value between 0.5 and 1 indicates persistent behavior (positive autocorrelation - trend). A value between 0 and 0.5 indicates an anti-persistent behavior (negative autocorrelation - mean reversion) and a value of 0.5 indicates a true

random process (Brownian time series). The exponent is calculated using the so-called Rescaled Range analysis, but we will not go through its computation due to the numerous steps required.

The code

The two main classes are *Portfolio* and *Pair*. The former receives the stocks dataframe as input and generates all the possible pairs combinations via **itertools.combinations**. After doing that, a for loop goes through all the possible pairs and, for each of them, it creates a *Pair* object.

```
class Portfolio:
    def __init__(self, stocks_df):
        if not isinstance(stocks_df, pd.core.frame.DataFrame):
            raise Exception("Symbols must be provided in a Pandas DataFrame")

        self.time_series_df = stocks_df
        self.time_series_df.dropna(inplace = True)
        self.symbols_list = self.time_series_df.columns
        self.all_possible_pairs = list(combinations(self.symbols_list, 2))
        self.selected_pairs = list()

        for pair_symbols in self.all_possible_pairs:
            pair = Pair(self.time_series_df[pair_symbols[0]],
                        self.time_series_df[pair_symbols[1]])
            if pair.eligible():
                self.selected_pairs.append(pair)
```

When a *Pair* object is created, an OLS is performed between the two stocks, and its Beta is used to compute the spread $Y - \beta X$ between the two securities. In addition to this, the spread is normalized and the p-value of Engle-Granger test and the Hurst Exponent of the spread are computed.

```
class Pair:
    P_VALUE_THRESHOLD = 0.05
    HURST_THRESHOLD = 0.5
    TRADING_BOUND = 1
    EXIT_PROFIT = 0
    STOP_LOSS = 2
    TRADING_PERIOD = 253

    def __init__(self, stockX, stockY, trading_period = TRADING_PERIOD, trading_bound = TRADING_BOUND,
                 exit_profit = EXIT_PROFIT, stop_loss = STOP_LOSS):
        self.name = f"{stockX.name}, {stockY.name}"
        self.stockX_trading = stockX[-trading_period:]
        self.stockY_trading = stockY[-trading_period:]

        self.trading_period = trading_period
        self.exit_profit = exit_profit
        self.trading_bound = trading_bound
        self.stop_loss = stop_loss

        self.error = False
        try:
            beta = OLS(stockY[-trading_period:], stockX[-trading_period:]).fit().params[0]
            self.spread = stockY[-trading_period] - beta * stockX[-trading_period:]
            self.normalized_spread_trading = (self.spread - self.spread.mean()) / self.spread.std()
            self.p_value = coint(stockX, stockY)[1]
            self.hurst = compute_Hc(self.spread)[0]
            self.generate_trading_signals()
        except Exception as e:
            print(e)
            print(f"Error encountered with pair [{self.stockX_trading.name}, {self.stockY_trading.name}]")
            self.error = True
```

These two values are then used in the **Pair.eligible** method, in which the pairs are tested for eligibility. In particular, the default thresholds are set respectively to 0.05 and 0.5, but they can be modified by the end user. This means that, for a pair in order to be eligible, it must have a p-value smaller than 0.05 and a Hurst exponent smaller than 0.5. This method is then used in the *Portfolio* class in order to check which pairs are going to be included in the **selected_pairs** list, on which the trading strategy will be applied.

```
def eligible(self, p_value_threshold = P_VALUE_THRESHOLD, hurst_threshold = HURST_THRESHOLD):
    if self.error:
        return False
    elif self.p_value <= p_value_threshold and self.hurst <= hurst_threshold:
        return True
    return False
```

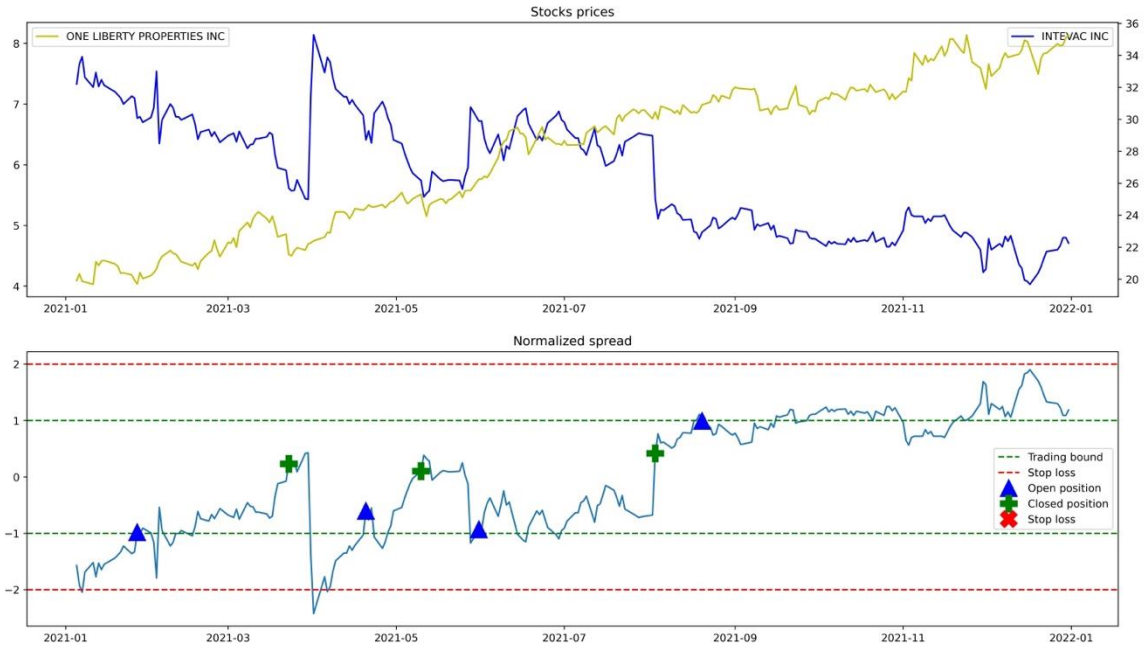
Finally, the method **plot_pair** shows a plot of the normalized spread of a pair, as well as the price of the two securities and, most importantly, the trading signals adopted by the trading strategy.

Trading strategy

Once all the pairs have been filtered out from the initial stocks, a pairs trading strategy should be defined. This consists primarily in setting the trading period, as well as the thresholds at which to enter a position and to exit from it, either with a profit or by incurring in a stop loss. In this case, all these thresholds are directly set in the constructor of the *Pair* class, and they always refer to the normalized spread between the two securities. Their default values are the following:

- Trading period: 253 days;
- Entry point: 1 std;
- Exit point: 0 std;
- Stop Loss: 2 std.

In order to detect when the spread crosses a certain threshold, the **__level_crosses** method is used, but we won't go into the details of this. To understand how the strategy works, let's have a look at this pair:



Entry points, exit points and stop losses are represented by blue triangles, green plus signs and red crosses.

As it has been already mentioned, the strategy is based on the assumption that short-term deviations from the long-term equilibrium of the two securities create trading opportunities, and for this reason we enter in a position whenever the normalized spread diverges more than a certain amount, which is depicted by the dotted green lines. But what do we exactly do when entering a position? Recall the spread formula:

$$S = Y - \beta X$$

If the spread is too high (above 1std) it means that *Y* is the rich leg (overvalued) to short and *X* is the cheap leg on which to go long. If the spread is too low (below 1std) we do the exact opposite.

We close the position as soon as the spread gets back to its normal level, id est when it reaches the Exit Point threshold. Alternatively, if the spread diverges to an excessive extent, which is set by default to 2 std, we incur in a stop loss. We set the stop loss so as to prevent exogenous shocks from causing huge losses.

Since every trade consists in both a long position and a short position, these positions can be considered entirely or at least partially self-financing. Indeed, the capital raised from selling a stock will be reinvested in buying the other stock. The computation of returns on zero-cost portfolios is a problematic concept and different solutions have been proposed to overcome this problem: returns may be calculated only on the long leg of the position, on the sum of the long leg and the absolute value of the short leg of the position or may take in consideration the margin capital needed to undertake the short position.

As returns can vary significantly according to the method chosen, in our backtesting environment we preferred to keep them as a simple difference between closing and initial spread. A less sensitive parameter that might be useful to evaluate the performance of the strategy is the percentage of profitable trades.

```
def generate_trading_signals(self):
    upper_trading = self.__level_crosses(self.normalized_spread_trading, level = self.trading_bound)
    lower_trading = self.__level_crosses(self.normalized_spread_trading, level = -self.trading_bound)
    upper_stop = self.__level_crosses(self.normalized_spread_trading, level = self.stop_loss)
    lower_stop = self.__level_crosses(self.normalized_spread_trading, level = -self.stop_loss)
    mean = self.__level_crosses(self.normalized_spread_trading, level = self.exit_profit)

    open_position = False
    entry_level = 0

    self.stop_losses = []
    self.open_positions = []
    self.closed_positions = []
    profits = []

    for i in range(0, len(self.normalized_spread_trading)):
        if open_position:
            if upper_stop[i] == 1 or lower_stop[i] == -1:
                #STOP LOSS
                open_position = False
                profits.append(-abs(self.spread[i] - entry_level))
                self.stop_losses.append(i)
            elif mean[i] != 0:
                #CLOSING WITH PROFIT
                open_position = False
                profits.append(abs(self.spread[i] - entry_level))
                self.closed_positions.append(i)
            else:
                profits.append(0)
        else:
            if upper_trading[i] == -1 or lower_trading[i] == 1:
                #ENTERING THE POSITION
                open_position = True
                entry_level = self.spread[i]
                self.open_positions.append(i)
            profits.append(0)

    self.profits_series = pd.Series(index = self.stockX_trading.index, data = profits)
    self.cum_profits = self.profits_series.cumsum()
```

```
def __level_crosses(self, series, level = 2):
    change = []
    for i, el in enumerate(series):
        if i != 0 and el > level and series[i-1] < level:
            change.append(1)
        elif i != 0 and el < level and series[i-1] > level:
            change.append(-1)
        else:
            change.append(0)
    return change
```

How to test the code with your data

In order to feed the program with stocks data, we have downloaded the CRSP Daily Stock database, with a date range from 01/01/2000 to 31/12/2021. However, the end user can adopt whatever stocks database he/she prefers, as long as it is similar to the following structure:

- Each column corresponds to a certain company, and each row to all the observations on a certain date.
- NaN values should be interpolated, if present.
- Prices columns should be made of nonnegative entries: for instance CRSP Daily Stock uses negative sign as a convention for when the closing price is not available on any given trading day.

Here's an example of how the database should look like after going through these preprocessing procedures:

COMNAM	1 800 FLOWERS COM INC	1ST CONSTITUTION BANCORP	1ST SOURCE CORP	3 D SYSTEMS CORP DEL	3M CO	51JOB INC	8X8 INC NEW	A A R CORP	A B B LTD	A B M INDUSTRIES INC	...	XILINX INC	XINYUAN REAL ESTATE CO LTD	Y P F SOCIEDAD ANONIMA
date														
2010-01-04	2.61	6.620	16.28	11.65	83.02	17.90	1.55	23.77	19.64	21.26	...	25.38	4.5000	44.6300
2010-01-05	2.52	6.000	15.23	11.63	82.50	18.30	1.57	24.23	19.77	20.97	...	25.06	4.5900	43.8300
2010-01-06	2.59	6.550	14.82	11.76	83.67	18.44	1.60	25.45	20.00	20.97	...	24.89	4.6900	44.7500
2010-01-07	2.56	6.170	14.79	12.16	83.73	19.00	1.63	25.85	20.17	21.24	...	24.64	4.6600	44.3200
2010-01-08	2.54	6.625	15.06	12.23	84.32	19.04	1.60	25.44	20.74	21.22	...	25.00	4.5900	44.8999
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2021-12-27	23.26	25.460	48.91	22.00	176.70	47.50	17.60	38.70	38.27	41.38	...	222.78	0.6163	4.0700
2021-12-28	23.01	25.880	48.80	21.31	177.64	48.62	17.45	38.99	38.38	41.08	...	220.27	0.5940	4.0700
2021-12-29	23.24	25.970	49.02	21.18	178.41	45.42	17.45	39.08	38.65	40.59	...	217.62	0.5451	3.9900
2021-12-30	23.67	25.520	48.69	21.76	177.64	48.94	17.46	38.55	38.14	40.54	...	213.89	0.5800	3.9100
2021-12-31	23.37	25.640	49.60	21.54	177.63	48.93	16.76	39.03	38.17	40.85	...	212.03	0.6300	3.8200

Future improvements

As we have shown, in order to select the eligible pairs from a portfolio of n stocks, it's necessary to find out all the possible combinations of stocks, which are exactly $\binom{n}{2}$, and for each combination we perform a linear regression, so as to calculate β, as well as a few statistical tests, namely, the Engle-Granger test and the calculation of the Hurst exponent. We can quickly see of this becomes infeasible as n grows. For instance, if $n = 3000$, the number of iterations is in the vicinity of 4.5 mln, which is quite computationally heavy for a standard computer.

For this reason, various Machine Learning techniques have been adopted in the last few years (see Han, He, Toh 2021). In particular, clustering algorithms manage to discover all the eligible pairs without confronting each stock pairwise. What's even more, it's possible to embed in the input of the algorithm not only past price data, like with the cointegration approach, but firm characteristics as well, and by doing this the Sharpe ratio of the selected portfolio increases sensibly.

TAGS: Pairs Trading, Python, Backtesting