Not Over Thinking

Pairs Trading with Stocks

Algorithmic Trading Strategy with Full Code

Haixiang

2023.08 | Vol 21.

hxyan.2015@gmail.com | github.com/hxyan2020

STRATEGY & ECONOMIC RATIONALE

The investment universe consists of stocks from NYSE, AMEX, and NASDAQ, while illiquid stocks a re removed from the investment universe. Cumulative total return index is then created for each stock (dividends included), and the starting price during the formation period is set to $1 (p rice normalization).

Pairs are formed over twelve months (formation period) and are then traded in the next six-mont h period (trading period). The matching partner for each stock is found by looking for the secu rity that minimizes the sum of squared deviations between two normalized price series.

Top 20 pairs with the smallest historical distance measure are then traded.

BUY	SELL
long-short position is opene d when pair prices have dive rged by two standard deviati ons	position is closed when pr ices revert.

PARAMETER & VARIABLES

PARAMETER	VALUE
MARKETS TRADED	Equity
FINANCIAL INSTRUMENTS	Stock
REGION	United States
PERIOD OF REBALANCING	Daily
NO. OF TRADED INSTRUMENTS	40
WEIGHTING	Equal weighting
LOOKBACK PERIODS	12 months
LONG/SHORT	Long & short

ALGORITHM

```
import numpy as np
import itertools as it

class PairsTradingwithStocks(QCAlgorithm):

    def Initialize(self):
        self.SetStartDate(2005, 1, 1)
        self.SetCash(100000)

        self.symbol = self.AddEquity('SPY', Resolution.Daily).Symbol

        # Daily price data.
        self.history_price = {}
        self.period = 12 * 21

        # Equally weighted brackets.
        self.max_traded_pairs = 5
```

```
        self.traded_pairs = []
        self.traded_quantity = {}

        self.sorted_pairs = []

        self.coarse_count = 500
        self.month = 6
        self.selection_flag = True
        self.UniverseSettings.Resolution = Resolution.Daily
        self.AddUniverse(self.CoarseSelectionFunction)
        self.Schedule.On(self.DateRules.MonthStart(self.symbol),
self.TimeRules.AfterMarketOpen(self.symbol), self.Selection)

    def OnSecuritiesChanged(self, changes):
        for security in changes.AddedSecurities:
            security.SetFeeModel(CustomFeeModel())
            security.SetLeverage(5)

        for security in changes.RemovedSecurities:
            symbol = security.Symbol
            if symbol in self.history_price:
                del self.history_price[symbol]

        symbols = [x for x in self.history_price.keys() if x != self.symbol]
        self.symbol_pairs = list(it.combinations(symbols, 2))

        # minimize the sum of squared deviations
        distances = {}
        for pair in self.symbol_pairs:
            if self.history_price[pair[0]].IsReady and self.history_price[pair[1]].IsReady:
                distances[pair] = self.Distance(self.history_price[pair[0]],
self.history_price[pair[1]])

        if len(distances) != 0:
            self.sorted_pairs = [x[0] for x in sorted(distances.items(), key = lambda x:
x[1])[:20]]

        self.Liquidate()
        self.traded_pairs.clear()
        self.traded_quantity.clear()

    def CoarseSelectionFunction(self, coarse):
        # Update the rolling window every day.
        for stock in coarse:
            symbol = stock.Symbol

            if symbol in self.history_price:
                self.history_price[symbol].Add(stock.AdjustedPrice)

        if not self.selection_flag:
            return Universe.Unchanged
        self.selection_flag = False
```

```
        selected = sorted([x for x in coarse if x.HasFundamentalData and x.Price > 5 and
x.Market == 'usa'],
            key=lambda x: x.DollarVolume, reverse=True)[:self.coarse_count]

        # Warmup price rolling windows.
        for stock in selected:
            symbol = stock.Symbol

            if symbol in self.history_price:
                continue

            self.history_price[symbol] = RollingWindow[float](self.period)
            history = self.History(symbol, self.period, Resolution.Daily)
            if history.empty:
                self.Log(f"Not enough data for {symbol} yet")
                continue
            closes = history.loc[symbol].close
            for time, close in closes.iteritems():
                self.history_price[symbol].Add(close)

        return [x.Symbol for x in selected if self.history_price[x.Symbol].IsReady]

    def OnData(self, data):
        if self.sorted_pairs is None: return

        pairs_to_remove = []

        for pair in self.sorted_pairs:
            # Calculate the spread of two price series.
            price_a = [x for x in self.history_price[pair[0]]]
            price_b = [x for x in self.history_price[pair[1]]]
            norm_a = np.array(price_a) / price_a[-1]
            norm_b = np.array(price_b) / price_b[-1]

            spread = norm_a - norm_b
            mean = np.mean(spread)
            std = np.std(spread)
            actual_spread = spread[0]

            # Long-short position is opened when pair prices have diverged by two standard
deviations.
            traded_portfolio_value = self.Portfolio.TotalPortfolioValue /
self.max_traded_pairs
            if actual_spread > mean + 2*std or actual_spread < mean - 2*std:
                if pair not in self.traded_pairs:
                    # open new position for pair, if there's place for it.
                    if len(self.traded_pairs) < self.max_traded_pairs:
                        symbol_a = pair[0]
                        symbol_b = pair[1]
                        a_price_norm = norm_a[0]
                        b_price_norm = norm_b[0]
                        a_price = price_a[0]
                        b_price = price_b[0]
```

```
                        # a stock's price > b stock's price
                        if a_price_norm > b_price_norm:
                            long_q = traded_portfolio_value / b_price    # long b stock
                            short_q = -traded_portfolio_value / a_price  # short a stock
                            if self.Securities.ContainsKey(symbol_a) and
self.Securities.ContainsKey(symbol_b) and \
                                    self.Securities[symbol_a].Price != 0 and
self.Securities[symbol_a].IsTradable and \
                                    self.Securities[symbol_b].Price != 0 and
self.Securities[symbol_b].IsTradable:
                                    self.MarketOrder(symbol_a, short_q)
                                    self.MarketOrder(symbol_b, long_q)

                                    self.traded_quantity[pair] = (short_q, long_q)
                                    self.traded_pairs.append(pair)
                        # b stock's price > a stock's price
                        else:
                            long_q = traded_portfolio_value / a_price
                            short_q = -traded_portfolio_value / b_price
                            if self.Securities.ContainsKey(symbol_a) and
self.Securities.ContainsKey(symbol_b) and \
                                    self.Securities[symbol_a].Price != 0 and
self.Securities[symbol_a].IsTradable and \
                                    self.Securities[symbol_b].Price != 0 and
self.Securities[symbol_b].IsTradable:
                                    self.MarketOrder(symbol_a, long_q)
                                    self.MarketOrder(symbol_b, short_q)

                                    self.traded_quantity[pair] = (long_q, short_q)
                                    self.traded_pairs.append(pair)
            # The position is closed when prices revert back.
            else:
                if pair in self.traded_pairs and pair in self.traded_quantity:
                    # make opposite order to opened position
                    self.MarketOrder(pair[0], -self.traded_quantity[pair][0])
                    self.MarketOrder(pair[1], -self.traded_quantity[pair][1])
                    pairs_to_remove.append(pair)

        for pair in pairs_to_remove:
            self.traded_pairs.remove(pair)
            del self.traded_quantity[pair]

    def Distance(self, price_a, price_b):
        # Calculate the sum of squared deviations between two normalized price series.
        price_a = [x for x in price_a]
        price_b = [x for x in price_b]

        norm_a = np.array(price_a) / price_a[-1]
        norm_b = np.array(price_b) / price_b[-1]
        return sum((norm_a - norm_b)**2)

    def Selection(self):
```

```
        if self.month == 6:
            self.selection_flag = True

        self.month += 1
        if self.month > 12:
            self.month = 1

# Custom fee model.
class CustomFeeModel(FeeModel):
    def GetOrderFee(self, parameters):
        fee = parameters.Security.Price * parameters.Order.AbsoluteQuantity * 0.00005
        return OrderFee(CashAmount(fee, "USD"))
```

BACKTESTING PERFORMANCE

Fig 1. Overall Performance

PSR	1.307%	Sharpe Ratio	0.59
Total Trades	8048	Average Win	0.45%
Average Loss	-0.47%	Compounding Annual Return	6.732%
Drawdown	27.300%	Expectancy	0.070
Net Profit	227.058%	Loss Rate	45%
Win Rate	55%	Profit-Loss Ratio	0.95
Alpha	0.045	Beta	0.069
Annual Standard Deviation	0.084	Annual Variance	0.007
Information Ratio	-0.145	Tracking Error	0.171
Treynor Ratio	0.723	Total Fees	$18212.08
Estimated Strategy Capacity	$7900000.00	Lowest Capacity Asset	NVR R735QTJ8XC9X

Fig 2. Performance Metrics

Fig 3. Drawdown

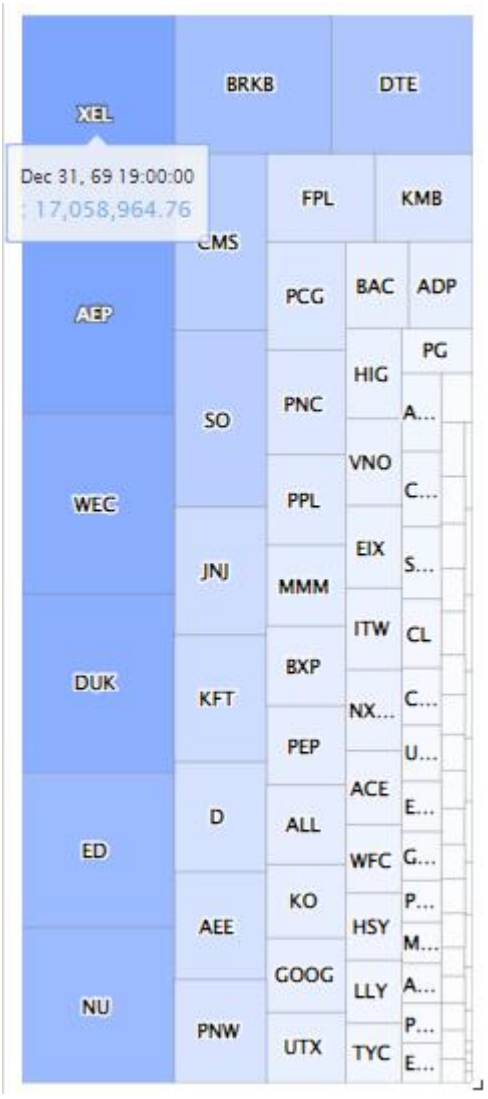

Fig 4. Assets Sales Volume