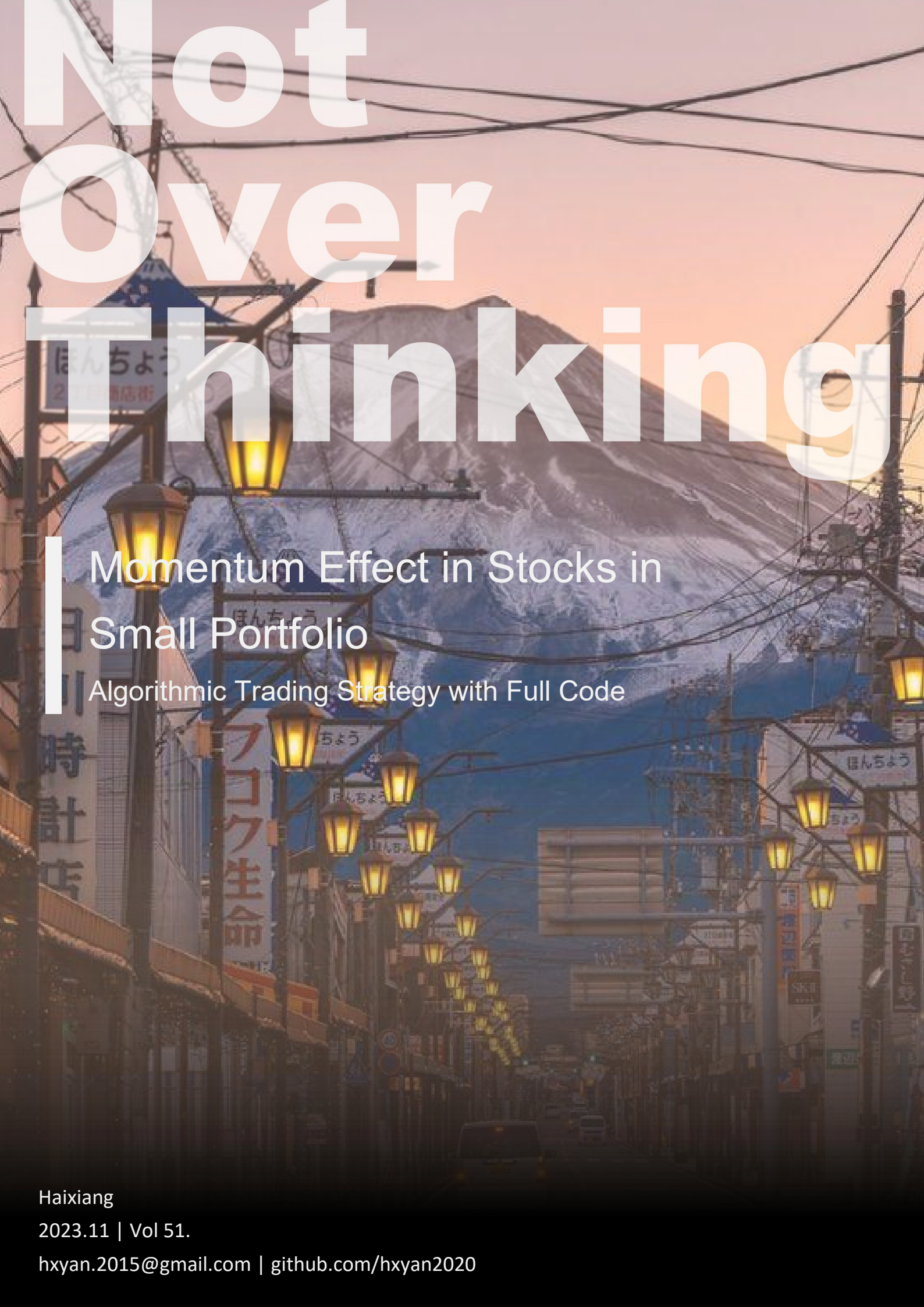

Momentum Effect in Stocks in Small Portfolio

Algorithmic Trading Strategy with Full Code

Haixiang

2023.11 | Vol 51.

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

STRATEGY & ECONOMIC RATIONALE

The investment universe consists of all UK listed companies (this is the investment universe used in the source academic study, and it could be easily changed into any other market – see Ammann, Moellenbeck, Schmid: Feasible Momentum Strategies in the US Stock Market). Stocks with the lowest market capitalization (25% of the universe) are excluded due to liquidity reasons. Momentum profits are calculated by ranking companies based on their stock market performance over the previous 12 months (the rank period). The investor goes long in the ten stocks with the highest performance and goes short in the ten stocks with the lowest performance. The portfolio is equally weighted and rebalanced yearly. We assume the investor has an account size of 10 000 pounds.

BUY	SELL
goes long in the ten stocks with the highest performance	goes short in the ten stocks with the lowest performance

PARAMETER & VARIABLES

PARAMETER	VALUE
MARKETS TRADED	Equity
FINANCIAL INSTRUMENTS	Stocks
REGION	Global
PERIOD OF REBALANCING	Yearly
NO. OF TRADED INSTRUMENTS	20
WEIGHTING	Equal weighting
LOOKBACK PERIODS	12 months
LONG/SHORT	Long & Short

ALGORITHM

```
from AlgorithmImports import *

class MomentumEffectinStocksinSmallPortfolios(QCAlgorithm):

    def Initialize(self):
        self.SetStartDate(2010, 1, 1)
        self.SetCash(100000)

        self.coarse_count = 500

        self.long = []
        self.short = []

        # Daily data.
        self.data = {}
        self.period = 12 * 21
        self.quantile = 10
        self.leverage = 5
```

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

        self.selection_flag = True
        self.UniverseSettings.Resolution = Resolution.Daily
        self.AddUniverse(self.CoarseSelectionFunction, self.FineSelectionFunction)

        self.month = 11
        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(self.leverage)

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

            if symbol in self.data:
                # Store daily price.
                self.data[symbol].update(stock.AdjustedPrice)

        # Selection once a month.
        if not self.selection_flag:
            return Universe.Unchanged

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

        # Warmup price rolling windows.
        for symbol in selected:
            if symbol in self.data:
                continue

            self.data[symbol] = SymbolData(symbol, 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.data[symbol].Price.Add(close)

        return [x for x in selected if self.data[x].is_ready()]

    def FineSelectionFunction(self, fine):
        fine = [x for x in fine if x.MarketCap != 0]
```

```
        # if len(fine) > self.coarse_count:
        #     sorted_by_market_cap = sorted(fine, key = lambda x: x.MarketCap,
reverse=True)
        #     top_by_market_cap = sorted_by_market_cap[:self.coarse_count]
        # else:
        #     top_by_market_cap = fine

        # Performance sorting.
        performance = {x.Symbol : self.data[x.Symbol].performance() for x in fine}

        if len(performance) >= self.quantile:
            decile = int(len(performance) / self.quantile)
            sorted_by_perf = sorted(performance.items(), key = lambda x: x[1], reverse =
True)
            self.long = [x[0] for x in sorted_by_perf[:decile]]
            self.short = [x[0] for x in sorted_by_perf[-decile:]]

        return self.long + self.short

    def OnData(self, data):
        if not self.selection_flag:
            return
        self.selection_flag = False

        # Trade execution.
        long_count = len(self.long)
        short_count = len(self.short)

        invested = [x.Key for x in self.Portfolio if x.Value.Invested]
        for symbol in invested:
            if symbol not in self.long + self.short:
                self.Liquidate(symbol)

        for symbol in self.long:
            if symbol in data and data[symbol]:
                self.SetHoldings(symbol, 1 / long_count)

        for symbol in self.short:
            if symbol in data and data[symbol]:
                self.SetHoldings(symbol, -1 / short_count)

    def Selection(self):
        # Rebalance every 12 months.
        if self.month == 12:
            self.selection_flag = True

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

class SymbolData():
    def __init__(self, symbol, period):
```

```
        self.Symbol = symbol
        self.Price = RollingWindow[float](period)

    def update(self, value):
        self.Price.Add(value)

    def is_ready(self):
        return self.Price.IsReady

    def performance(self):
        closes = [x for x in self.Price]
        return (closes[0] / closes[-1] - 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	0.000%	Sharpe Ratio	-0.123
Total Trades	2326	Average Win	0.75%
Average Loss	-0.92%	Compounding Annual Return	-7.772%
Drawdown	86.100%	Expectancy	-0.101
Net Profit	-65.828%	Loss Rate	50%
Win Rate	50%	Profit-Loss Ratio	0.81
Alpha	-0.047	Beta	0.191
Annual Standard Deviation	0.232	Annual Variance	0.054
Information Ratio	-0.481	Tracking Error	0.259
Treynor Ratio	-0.149	Total Fees	$332.23
Estimated Strategy Capacity	$280000000.00	Lowest Capacity Asset	ATKR WB9Q4ETW5V1H
Portfolio Turnover	1.16%		

Fig 2. Performance Metrics

Fig 3. Drawdown

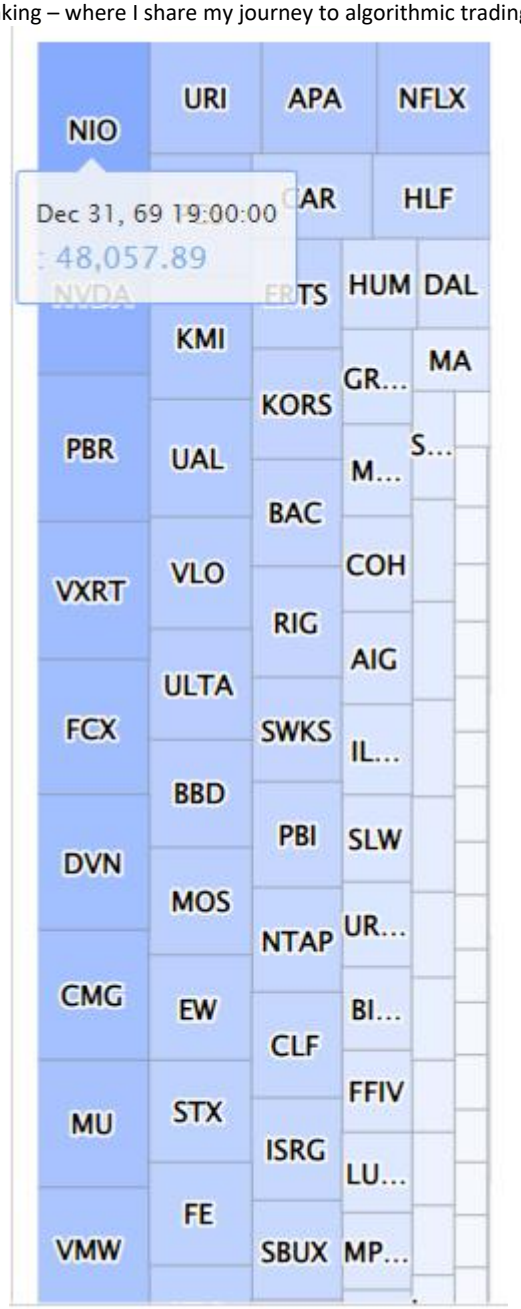

Fig 4. Assets Sales Volume