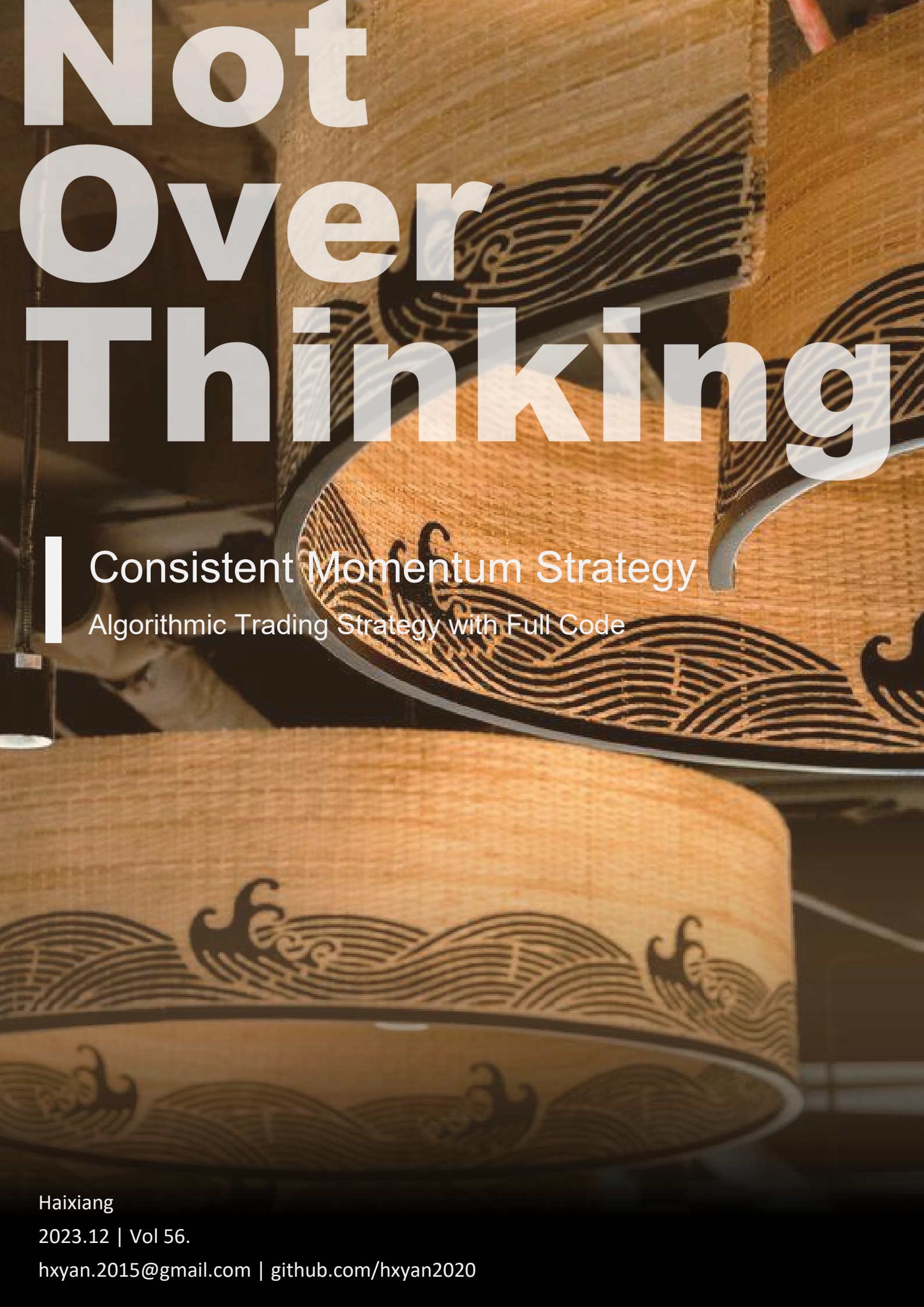

Not Over Thinking

Consistent Momentum Strategy

Algorithmic Trading Strategy with Full Code

Haixiang

2023.12 | Vol 56.

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

STRATEGY & ECONOMIC RATIONALE

The investment universe consists of stocks listed at NYSE, AMEX, and NASDAQ, whose price data (at least for the past seven months) are available at the CRSP database. The investor creates a zero-investment portfolio at the end of the month t, longing stocks that are in the top decile in terms of returns both in the period from t-7 to t-1 and from t-6 to t, while shorting stocks in the bottom decile in both periods (i.e. longing consistent winners and shorting consistent losers). The stocks in the portfolio are weighted equally. The holding period is six months, with no rebalancing during the period. There is a one-month skip between the formation and holding period.

BUY	SELL
longing stocks that are in the top decile in terms of returns both in the period from t-7 to t-1 and from t-6 to t	shorting stocks in the bottom decile in both periods

PARAMETER & VARIABLES

PARAMETER	VALUE
MARKETS TRADED	Equity
FINANCIAL INSTRUMENTS	Stocks
REGION	United States
PERIOD OF REBALANCING	Six months
NO. OF TRADED INSTRUMENTS	1000
WEIGHTING	Equal weighting
LOOKBACK PERIODS	Week
LONG/SHORT	Long & short

ALGORITHM

```
from AlgorithmImports import *

class ConsistentMomentumStrategy(QCAlgorithm):

    def Initialize(self):
        self.SetStartDate(2000, 1, 1)
        self.SetCash(100000)

        self.coarse_count = 500

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

        self.data = {}

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

        self.period = 7 * 21

        self.months = 0
        self.selection_flag = False
        self.UniverseSettings.Resolution = Resolution.Daily
```

```
        self.AddUniverse(self.CoarseSelectionFunction, self.FineSelectionFunction)

        self.Schedule.On(self.DateRules.MonthEnd(self.symbol), self.TimeRules.AfterMarketOpen(s
elf.symbol), self.Rebalance)

    def OnSecuritiesChanged(self, changes):
        for security in changes.AddedSecurities:
            symbol = security.Symbol

            security.SetFeeModel(CustomFeeModel())
            security.SetLeverage(10)

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

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

        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].update(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 and x.CompanyReference.IsREIT != 1 and  \
                    ((x.SecurityReference.ExchangeId == "NYS") or (x.SecurityReference.Exchange
Id == "NAS") or (x.SecurityReference.ExchangeId == "ASE"))]

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

        momentum_t71_t60 = { x : (self.data[x].performance_t7t1(), self.data[x].performance_t6t
0()) for x in top_by_market_cap}

        # Momentum t-7 to t-1 sorting
        sorted_by_perf_t71 = sorted(momentum_t71_t60.items(), key = lambda x: x[1][0], reverse
= True)
        decile = int(len(sorted_by_perf_t71) / 10)
```

```
        high_by_perf_t71 = [x[0] for x in sorted_by_perf_t71[:decile]]
        low_by_perf_t71 = [x[0] for x in sorted_by_perf_t71[-decile:]]

        # Momentum t-6 to t sorting
        sorted_by_perf_t60 = sorted(momentum_t71_t60.items(), key = lambda x: x[1][1], reverse
= True)
        decile = int(len(sorted_by_perf_t60) / 10)
        high_by_perf_t60 = [x[0] for x in sorted_by_perf_t60[:decile]]
        low_by_perf_t60 = [x[0] for x in sorted_by_perf_t60[-decile:]]

        self.long = [x for x in high_by_perf_t71 if x in high_by_perf_t60]
        self.short = [x for x in low_by_perf_t71 if x in low_by_perf_t60]

        self.selection_flag = False

        return self.long + self.short

    def Rebalance(self):
        if self.months == 0:
            self.selection_flag = True
            self.months += 1
            return

        if self.months == 1:
            # Trade execution and liquidation.
            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)

            long_count = len(self.long)
            short_count = len(self.short)

            for symbol in self.long:
                self.SetHoldings(symbol, 1/long_count)
            for symbol in self.short:
                self.SetHoldings(symbol, -1/short_count)

        self.months += 1

        if self.months == 6:
            self.months = 0

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_t7t1(self):
        closes = [x for x in self.Price][21:]
        return (closes[0] / closes[-1] - 1)

    def performance_t6t0(self):
        closes = [x for x in self.Price][:-21]
        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

Total Trades	4711	Average Win	1.08%
Average Loss	-1.16%	Compounding Annual Return	-1.700%
Drawdown	87.800%	Expectancy	-0.006
Net Profit	-32.805%	Sharpe Ratio	0.097
Probabilistic Sharpe Ratio	0.000%	Loss Rate	49%
Win Rate	51%	Profit-Loss Ratio	0.93
Alpha	0.035	Beta	-0.143
Annual Standard Deviation	0.274	Annual Variance	0.075
Information Ratio	-0.092	Tracking Error	0.33
Treynor Ratio	-0.186	Total Fees	$588.27
Estimated Strategy Capacity	$1900000.00	Lowest Capacity Asset	U XHYQYCUDLKKL

Fig 2. Performance Metrics