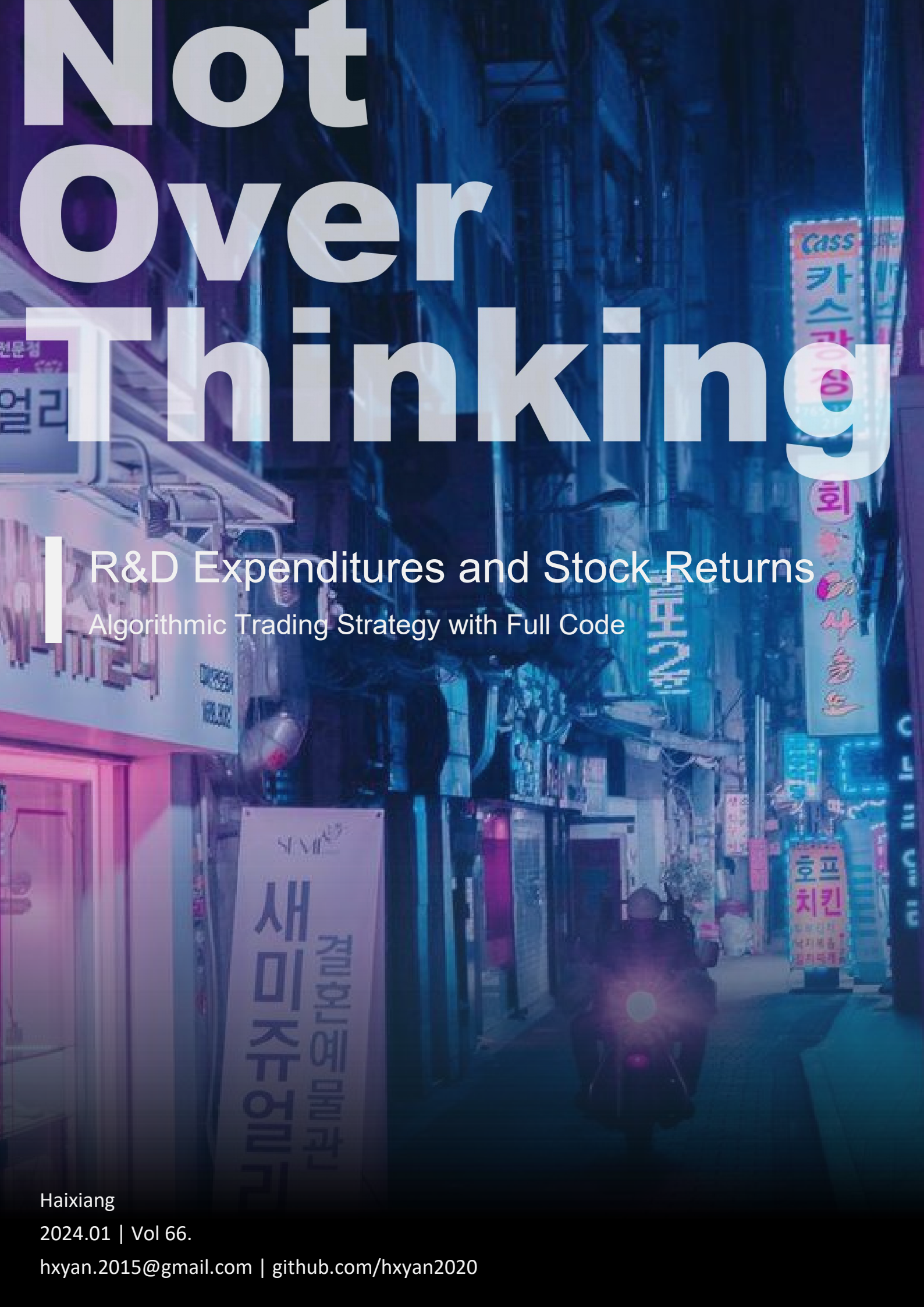
Not Over Thinking

R&D Expenditures and Stock Returns

Algorithmic Trading Strategy with Full Code



Haixiang

2024.01 | Vol 66.

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

STRATEGY & ECONOMIC RATIONALE

The investment universe consists of stocks that are listed on NYSE NASDAQ or AMEX. At the end o f April, for each stock in the universe, calculate a measure of total R&D expenditures in the p ast 5 years scaled by the firm's Market cap (defined on page 7, eq. 1). Go long (short) on the quintile of firms with the highest (lowest) R&D expenditures relative to their Market Cap. Weig ht the portfolio equally and rebalance next year. The backtested performance of the paper is su bstituted by our more recent backtest in Quantconnect.

BUY	SELL
Go long on the quintile of f irms with the highest R&D ex penditures relative to their Market Cap.	Go short on the quintile o f firms with the lowest R& D expenditures relative to their Market Cap.

PARAMETER & VARIABLES

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

ALGORITHM

```
from AlgorithmImports import *from numpy import log, averagefrom scipy import statsimport numpy
 as np#endregion
class RDExpendituresandStockReturns(QCAlgorithm):

    def Initialize(self):
        self.SetStartDate(1998, 1, 1)
        self.SetCash(100000)

        self.weight = {}
        self.coarse_count = 3000

        # R&D history.
        self.RD = {}
        self.rd_period = 5
        self.quantile = 5

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

        data = self.AddEquity('XLK', Resolution.Daily)
        data.SetLeverage(10)
        self.technology_sector = data.Symbol

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

```
        self.selection_flag = True
        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.Selection)

    def OnSecuritiesChanged(self, changes):
        for security in changes.AddedSecurities:
            security.SetLeverage(10)
            security.SetFeeModel(CustomFeeModel())

    def CoarseSelectionFunction(self, coarse):
        if not self.selection_flag:
            return Universe.Unchanged

        selected = [x.Symbol for x in coarse if x.HasFundamentalData and x.Price > 5]

        return selected

    def FineSelectionFunction(self, fine):
        fine = [x for x in fine if (x.FinancialStatements.IncomeStatement.ResearchAndDevelopmen
t.TwelveMonths) and \
                                    (x.MarketCap != 0) and \
                                    ((x.SecurityReference.ExchangeId == "NYS") or (x.SecurityRe
ference.ExchangeId == "NAS") or (x.SecurityReference.ExchangeId == "ASE"))]
                                    #and x.AssetClassification.MorningstarSectorCode == Morning
starSectorCode.Technology]

        top_by_market_cap = None
        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

        fine_symbols = [x.Symbol for x in top_by_market_cap]
        ability = {}

        updated_flag = []  # updated this year already

        for stock in top_by_market_cap:
            symbol = stock.Symbol

            # prevent storing duplicated value for the same stock in one year
            if symbol not in updated_flag:

                # Update RD.
                if symbol not in self.RD:
                    self.RD[symbol] = RollingWindow[float](self.rd_period)
                #rd = stock.FinancialStatements.IncomeStatement.ResearchAndDevelopment.TwelveMo
nths
                #self.RD[symbol].Add(rd)

                if self.RD[symbol].IsReady:
                    coefs = np.array([1, 0.8, 0.6, 0.4, 0.2])
                    rds = np.array([x for x in self.RD[symbol]])

                    rdc = sum(coefs * rds)
                    ability[stock] = rdc/stock.MarketCap

                rd = stock.FinancialStatements.IncomeStatement.ResearchAndDevelopment.TwelveMon
ths
                self.RD[symbol].Add(rd)
```

```
            # prevent storing duplicated value for the same stock in one year
            if fine_symbols.count(symbol) > 1:
                updated_flag.append(symbol)

        # Ability market cap weighting.
        #total_market_cap = sum([x.MarketCap for x in ability])
        #for stock, rdc in ability.items():
            #ability[stock] = rdc * (stock.MarketCap / total_market_cap)

        # Remove not updated symbols
        symbols_to_delete = []
        for symbol in self.RD.keys():
            if symbol not in fine_symbols:
                symbols_to_delete.append(symbol)
        for symbol in symbols_to_delete:
            if symbol in self.RD:
                del self.RD[symbol]

        # starts trading after data storing period
        if len(ability) >= self.quantile:
            # Ability sorting.
            sorted_by_ability = sorted(ability.items(), key = lambda x: x[1], reverse = True)
            quantile = int(len(sorted_by_ability) / self.quantile)
            high_by_ability = [x[0].Symbol for x in sorted_by_ability[:quantile]]
            low_by_ability = [x[0].Symbol for x in sorted_by_ability[-quantile:]]

            self.long = high_by_ability
            self.short = low_by_ability
            #self.short = [self.technology_sector]

        return self.long + self.short

    def Selection(self):
        if self.Time.month == 4:
            self.selection_flag = True

    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)

        stocks_invested = [x.Key for x in self.Portfolio if x.Value.Invested]
        for symbol in stocks_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)

        self.long.clear()
        self.short.clear()
        class SymbolData():
    def __init__(self, tested_growth, period):
        self.TestedGrowth = tested_growth
        self.RD = RollingWindow[float](period)
```

```
    def update(self, window_value):
        self.RD.Add(window_value)

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

BACKTESTING PERFORMANCE

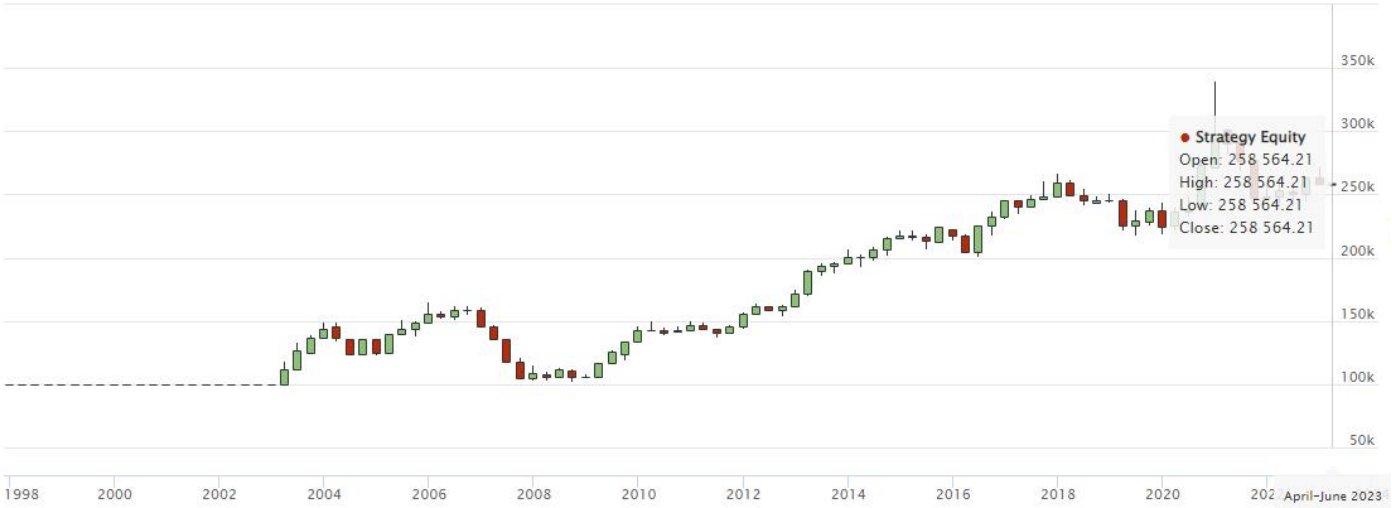


Fig 1. Overall Performance

Total Trades	3516	Average Win	0.37%
Average Loss	-0.21%	Compounding Annual Return	3.831%
Drawdown	38.600%	Expectancy	0.289
Net Profit	158.564%	Sharpe Ratio	0.362
Probabilistic Sharpe Ratio	0.005%	Loss Rate	53%
Win Rate	47%	Profit-Loss Ratio	1.72
Alpha	0.025	Beta	0.074
Annual Standard Deviation	0.082	Annual Variance	0.007
Information Ratio	-0.217	Tracking Error	0.171
Treynor Ratio	0.404	Total Fees	$236.27
Estimated Strategy Capacity	$31000.00	Lowest Capacity Asset	YNDX UWU1S0AN2N39
Portfolio Turnover	0.29%		

Fig 2. Performance Metrics