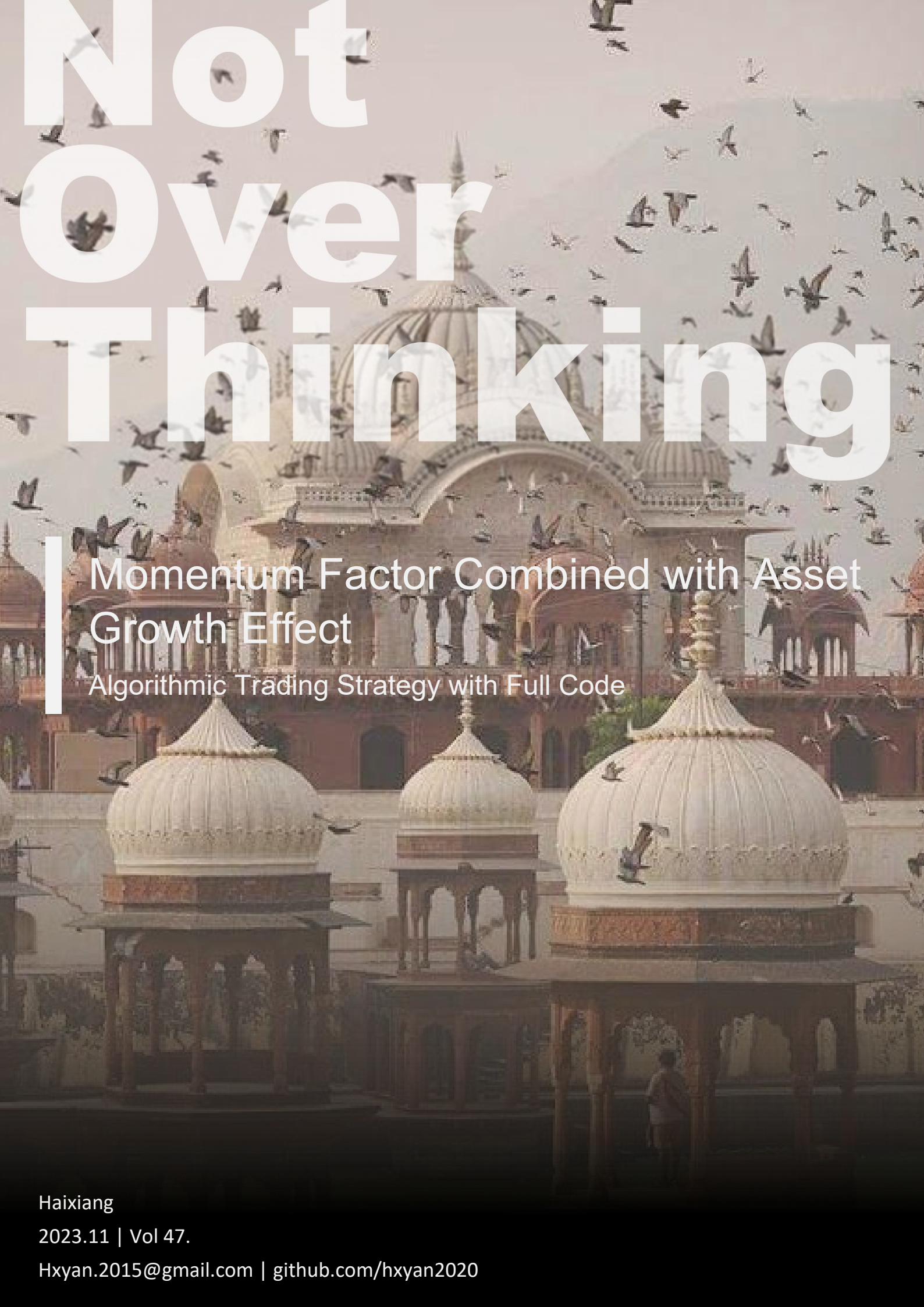

Not Over Thinking

Momentum Factor Combined with Asset Growth Effect

Algorithmic Trading Strategy with Full Code

Haixiang

2023.11 | Vol 47.

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

STRATEGY & ECONOMIC RATIONALE

The investment universe consists of NYSE, AMEX, and NASDAQ stocks (data for the backtest in the source paper are from Compustat). Stocks with a market capitalization less than the 20th NYSE percentile (smallest stocks) are removed.

The asset growth variable is defined as the yearly percentage change in the balance sheet total assets. Data from year t-2 to t-1 are used to calculate asset growth, and July is the cut-off month. Every month, stocks are then sorted into deciles based on asset growth, and only stocks with the highest asset growth are used.

The next step is to sort stocks from the highest asset growth decile into quintiles, based on their past 11-month return (with the last month's performance skipped in the calculation). The investor then goes long on stocks with the strongest momentum and short on stocks with the weakest momentum. The portfolio is equally weighted and is rebalanced monthly. The investor holds long-short portfolios only during February-December -> January is excluded as this month has been repeatedly documented as a negative month for a momentum strategy (see "January Effect Filter and Momentum in Stocks").

BUY	SELL
goes long on stocks with the strongest momentum	short on stocks with the weakest momentum

PARAMETER & VARIABLES

PARAMETER	VALUE
MARKETS TRADED	Equity
FINANCIAL INSTRUMENTS	Stocks
REGION	United States
PERIOD OF REBALANCING	Monthly
NO. OF TRADED INSTRUMENTS	1000
WEIGHTING	Equal weighting
LOOKBACK PERIODS	11 months
LONG/SHORT	Long & Short

ALGORITHM

```
from AlgorithmImports import *

class MomentumFactorAssetGrowthEffect(QCAlgorithm):

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

        # Monthly close data.
        self.data = {}
        self.period = 13
        self.total_assets_history_period = 2

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

```
        self.spy_consolidator = TradeBarConsolidator(timedelta(days=21))
        self.spy_consolidator.DataConsolidated += self.CustomHandler
        self.SubscriptionManager.AddConsolidator(self.symbol,
self.spy_consolidator)

        self.data[self.symbol] = SymbolData(self.symbol, self.period,
self.total_assets_history_period)
        # Warmup market history.
        history = self.History(self.symbol, self.period, Resolution.Daily)
        if not history.empty:
            closes = history.loc[self.symbol].close
            closes_len = len(closes.keys())
            # Find monthly closes.
            for index, time_close in enumerate(closes.iteritems()):
                # index out of bounds check.
                if index + 1 < closes_len:
                    date_month = time_close[0].date().month
                    next_date_month = closes.keys()[index + 1].month

                    # Found last day of month.
                    if date_month != next_date_month:
                        self.data[self.symbol].update(time_close[1])

        self.coarse_count = 500

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

        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.BeforeMarketClose(self.symbol), self.Selection)

    def CustomHandler(self, sender, consolidated):
        self.data[self.symbol].update(consolidated.Close)

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

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

        # Update the rolling window every month.
        for stock in coarse:
            symbol = stock.Symbol

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

```
        # 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,
self.total_assets_history_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

            closes_len = len(closes.keys())
            # Find monthly closes.
            for index, time_close in enumerate(closes.iteritems()):
                # index out of bounds check.
                if index + 1 < closes_len:
                    date_month = time_close[0].date().month
                    next_date_month = closes.keys()[index + 1].month

                    # Found last day of month.
                    if date_month != next_date_month:
                        self.data[symbol].update(time_close[1])

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

    def FineSelectionFunction(self, fine):
        fine = [x for x in fine if
x.FinancialStatements.BalanceSheet.TotalAssets.TwelveMonths > 0 and
                ((x.SecurityReference.ExchangeId == "NYS") or
(x.SecurityReference.ExchangeId == "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 = sorted_by_market_cap[:self.coarse_count]
        # else:
        #     top_by_market_cap = fine

        top_by_market_cap = fine

        # Asset growth calc.
        asset_growth = {}
        for stock in top_by_market_cap:
```

```
            symbol = stock.Symbol

            if self.data[symbol].asset_data_is_ready():
                asset_growth[symbol] = self.data[symbol].asset_growth()

            self.data[symbol].update_assets(stock.FinancialStatements.BalanceSheet.TotalAs
sets.TwelveMonths)

        sorted_by_growth = sorted(asset_growth.items(), key = lambda x: x[1], reverse =
True)
        decile = int(len(sorted_by_growth) / 10)
        top_by_growth = [x[0] for x in sorted_by_growth][:decile]

        performance = { x : self.data[x].performance(1) for x in top_by_growth}
        sorted_by_performance = sorted(performance.items(), key = lambda x: x[1], reverse
= True)
        quintile = int(len(sorted_by_performance) / 5)
        self.long = [x[0] for x in sorted_by_performance][:quintile]
        self.short = [x[0] for x in sorted_by_performance][-quintile:]

        return self.long + self.short

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

        # Trade execution.
        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:
            self.SetHoldings(symbol, 1 / len(self.long))
        for symbol in self.short:
            self.SetHoldings(symbol, -1 / len(self.short))

        self.long.clear()
        self.short.clear()

    def Selection(self):
        # Exclude January trading.
        if self.Time.month != 12:
            self.selection_flag = True
        else:
            self.Liquidate()

class SymbolData():
    def __init__(self, symbol, period, total_assets_history_period):
        self.Symbol = symbol
        self.Price = RollingWindow[float](period)
        self.TotalAssets = RollingWindow[float](total_assets_history_period)
```

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

    def update_assets(self, assets_value):
        self.TotalAssets.Add(assets_value)

    def asset_data_is_ready(self) -> bool:
        return self.TotalAssets.IsReady

    def asset_growth(self) -> float:
        asset_values = [x for x in self.TotalAssets]
        return (asset_values[0] - asset_values[1]) / asset_values[1]

    def is_ready(self) -> bool:
        return self.Price.IsReady

    # Performance, one month skipped.
    def performance(self, values_to_skip = 0) -> float:
        closes = [x for x in self.Price][values_to_skip:]
        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

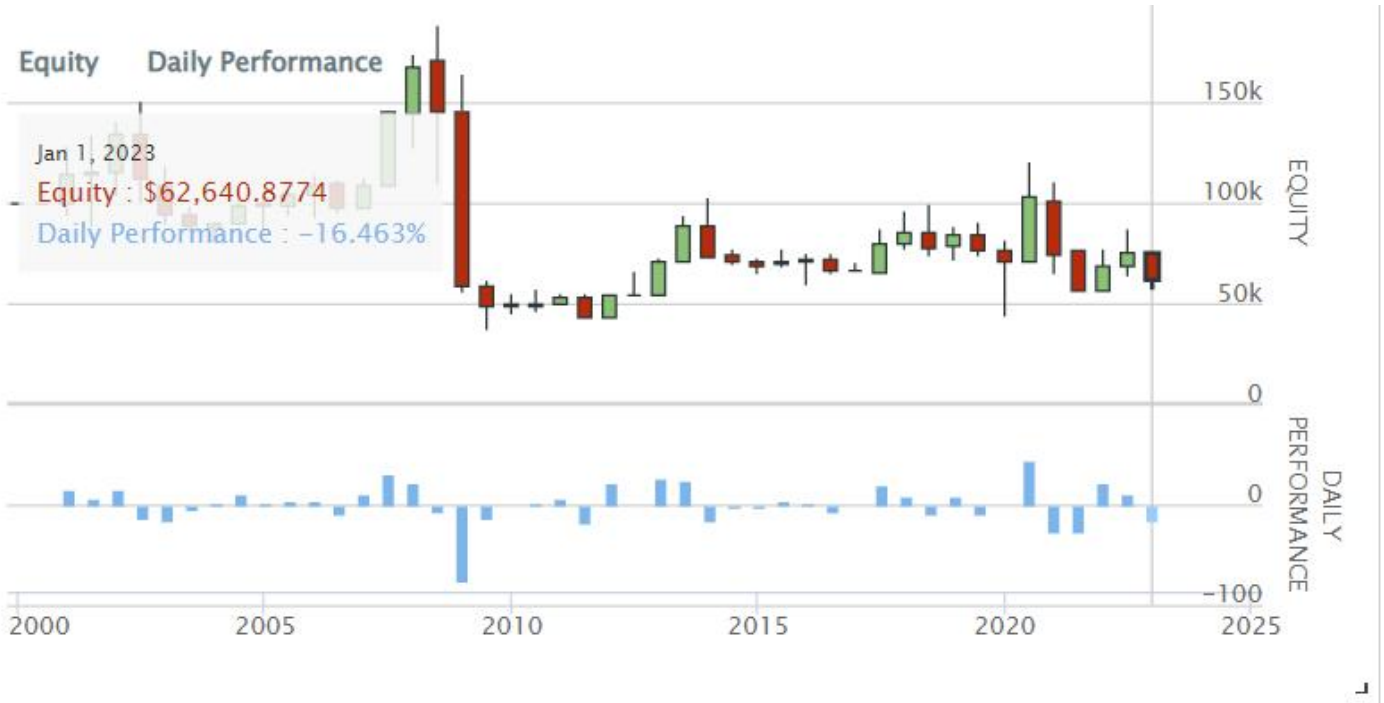

Fig 1. Overall Performance

PSR	0.000%	Sharpe Ratio	0.072
Total Trades	7830	Average Win	0.96%
Average Loss	-1.00%	Compounding Annual Return	-1.989%
Drawdown	80.600%	Expectancy	0.000
Net Profit	-37.359%	Loss Rate	49%
Win Rate	51%	Profit-Loss Ratio	0.96
Alpha	0.03	Beta	-0.203
Annual Standard Deviation	0.252	Annual Variance	0.063
Information Ratio	-0.125	Tracking Error	0.317
Treynor Ratio	-0.089	Total Fees	$3449.10
Estimated Strategy Capacity	$18000000.00	Lowest Capacity Asset	FTI S5ITKJ2R97C5
Portfolio Turnover	9.63%		

Fig 2. Performance Metrics

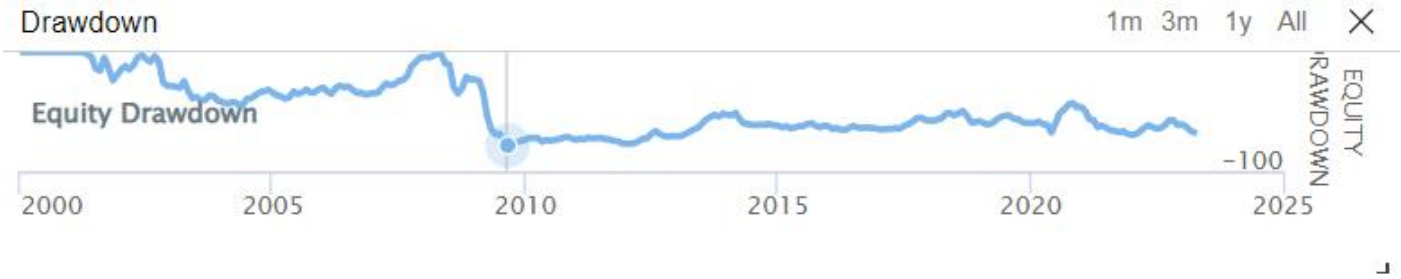

Fig 3. Drawdown

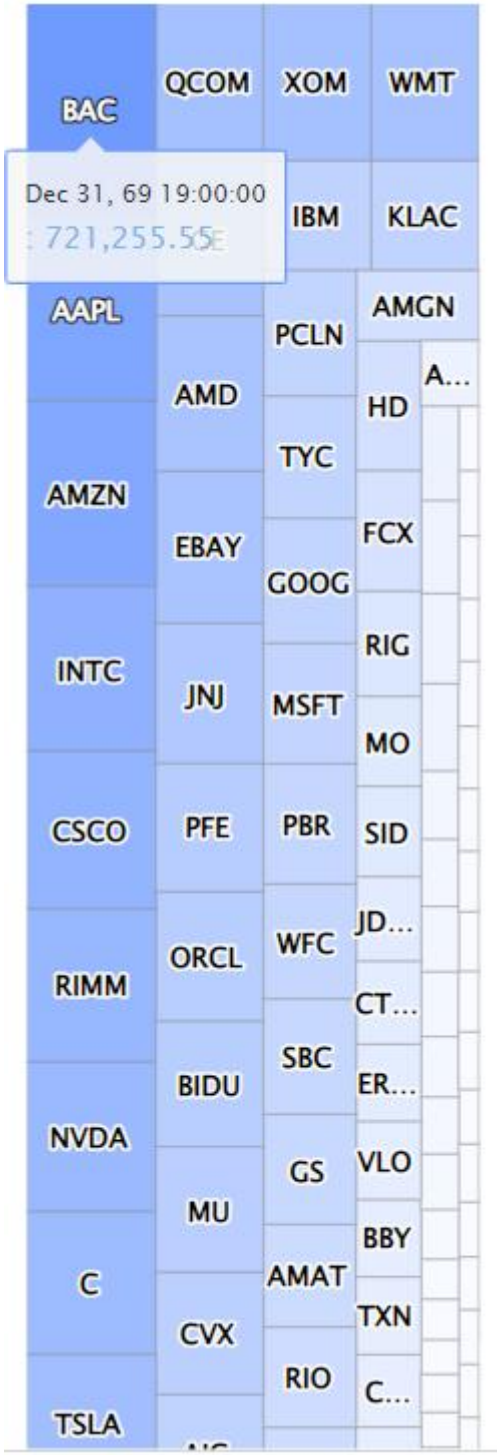

Fig 4. Assets Sales Volume