Not Over Thinking

Earnings Announcement Premium

Algorithmic Trading Strategy with Full Code

Haixiang

2023.10 | Vol 37.

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

STRATEGY & ECONOMIC RATIONALE

The investment universe consists of all stocks from the CRSP database. At the beginning of every calendar month, stocks are ranked in ascending order on the basis of the volume concentration ratio, which is defined as the volume of the previous 16 announcement months divided by the total volume in the previous 48 months.

The ranked stocks are assigned to one of 5 quintile portfolios. Within each quintile, stocks are assigned to one of two portfolios (expected announcers and expected non-announcers) using the predicted announcement based on the previous year. All stocks are value-weighted within a given portfolio, and portfolios are rebalanced every calendar month to maintain value weights.

The investor invests in a long-short portfolio, which is a zero-cost portfolio that holds the portfolio of high volume expected announcers and sells short the portfolio of high volume expected non-announcers.

BUY	SELL
portfolio of high volume expected announcers	portfolio of high volume expected non-announcers.

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	Monthly
LONG/SHORT	Long & Short

ALGORITHM

```python
from collections import deque
from AlgorithmImports import *

class EarningsAnnouncementPremium(QCAlgorithm):

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

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

        self.period = 21
        self.month_period = 48

        # Volume daily data.
        self.data = {}
        # Volume monthly data.
        self.monthly_volume = {}

        self.coarse_count = 1000
        self.weight = {}

        self.selection_flag = True
        self.UniverseSettings.Resolution = Resolution.Daily
        self.AddUniverse(self.CoarseSelectionFunction, self.FineSelectionFunction)
        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(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].Add(stock.Volume)

        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 volume rolling windows.
        for symbol in selected:
            # Warmup data.
            if symbol not in self.data:
                self.data[symbol] = RollingWindow[float](self.period)

                history = self.History(symbol, self.period, Resolution.Daily)
                if history.empty:
                    self.Debug(f"No history for {symbol} yet")
                    continue
                volumes = history.loc[symbol].volume
                for _, volume in volumes.iteritems():
                    self.data[symbol].Add(volume)

        return [x for x in selected if self.data[x].IsReady]

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

        fine_symbols = [x.Symbol for x in top_by_market_cap]

        # Ratio/market cap pair.
        volume_concentration_ratio = {}
        for stock in top_by_market_cap:
            symbol = stock.Symbol

            if symbol not in self.monthly_volume:
                self.monthly_volume[symbol] = deque(maxlen = self.month_period)

            monthly_vol = sum([x for x in self.data[symbol]])
            last_month_date = self.Time - timedelta(days = self.Time.day)
            last_file_date = stock.EarningReports.FileDate # stock annoucement day
            was_announcement_month = (last_file_date.year == last_month_date.year and 
last_file_date.month == last_month_date.month)    # Last month was announcement date.
            self.monthly_volume[symbol].append(VolumeData(last_month_date, monthly_vol, 
was_announcement_month))

            # 48 months of volume data is ready.
            if len(self.monthly_volume[symbol]) == self.monthly_volume[symbol].maxlen:
                # Volume concentration ratio calc.
                announcement_count = 16
                announcement_volumes = [x.Volume for x in self.monthly_volume[symbol] if 
x.WasAnnouncementMonth][-announcement_count:]

                if len(announcement_volumes) == announcement_count:
                    announcement_months_volume = sum(announcement_volumes)
                    total_volume = sum([x.Volume for x in self.monthly_volume[symbol]])

                    if announcement_months_volume != 0 and total_volume != 0:
                        # Store ratio, market cap pair.
                        volume_concentration_ratio[stock] = announcement_months_volume / 
total_volume

        # Volume sorting.
        sorted_by_volume = sorted(volume_concentration_ratio.items(), key = lambda x: x[1], 
reverse = True)
        quintile = int(len(sorted_by_volume) / 5)
        high_volume = [x[0] for x in sorted_by_volume[:quintile]]
        # Filering announcers and non-announcers.
        month_to_lookup = self.Time.month
        year_to_lookup = self.Time.year - 1

        long = []
        short = []
        for stock in high_volume:
            symbol = stock.Symbol

            announcement_dates = [[x.Date.year, x.Date.month] for x in 
self.monthly_volume[symbol] if x.WasAnnouncementMonth]
            if [year_to_lookup, month_to_lookup] in announcement_dates:
                long.append(stock)
            else:
                short.append(stock)

        # Delete not updated symbols.
        symbols_to_remove = []
        for symbol in self.monthly_volume:
            if symbol not in fine_symbols:
                symbols_to_remove.append(symbol)
        for symbol in symbols_to_remove:
            del self.monthly_volume[symbol]

        # Market cap weighting.
        total_market_cap_long = sum([x.MarketCap for x in long])
        for stock in long:
            self.weight[symbol] = stock.MarketCap  / total_market_cap_long

        total_market_cap_short = sum([x.MarketCap for x in short])
        for stock in short:
            self.weight[symbol] = -stock.MarketCap  / total_market_cap_short

        return [x[0] for x in self.weight.items()]

    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.weight:
                self.Liquidate(symbol)

        for symbol, w in self.weight.items():
            if self.Securities[symbol].Price != 0:  # Prevent error message.
                self.SetHoldings(symbol, w)

        self.weight.clear()

    def Selection(self):
        self.selection_flag = True

# Monthly volume data.
class VolumeData():
    def __init__(self, date, monthly_volume, was_announcement_month):
        self.Date = date
        self.Volume = monthly_volume
        self.WasAnnouncementMonth = was_announcement_month

# 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

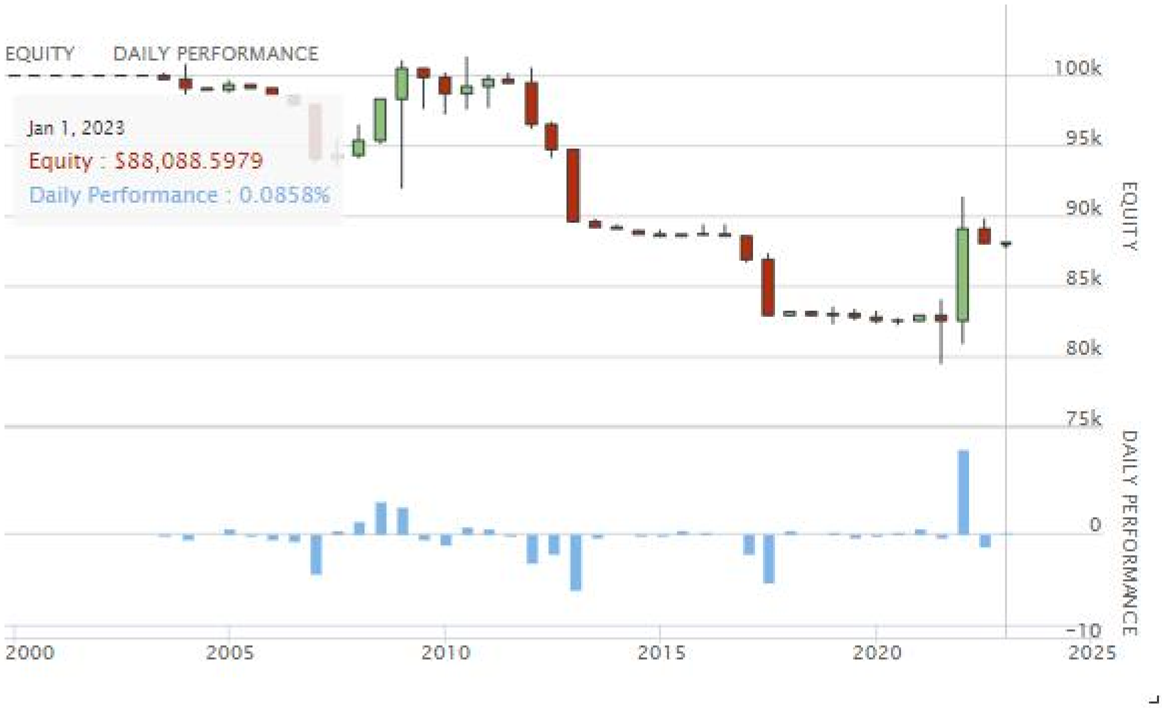
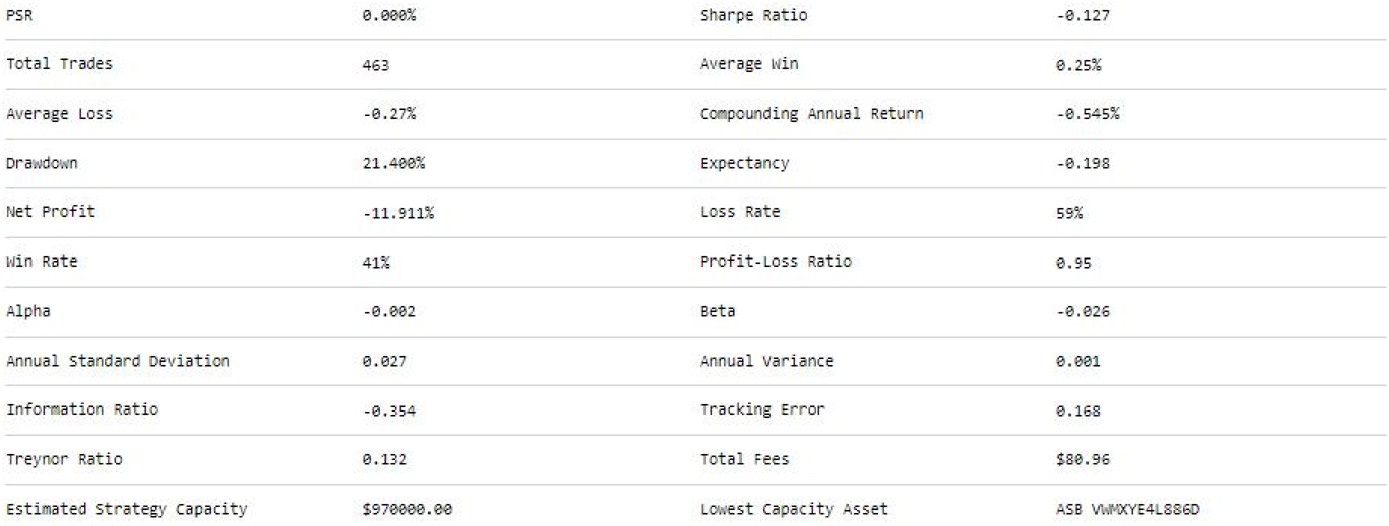


Fig 1. Overall Performance



PSR	0.000%	Sharpe Ratio	-0.127
Total Trades	463	Average Win	0.25%
Average Loss	-0.27%	Compounding Annual Return	-0.545%
Drawdown	21.400%	Expectancy	-0.198
Net Profit	-11.911%	Loss Rate	59%
Win Rate	41%	Profit-Loss Ratio	0.95
Alpha	-0.002	Beta	-0.026
Annual Standard Deviation	0.027	Annual Variance	0.001
Information Ratio	-0.354	Tracking Error	0.168
Treynor Ratio	0.132	Total Fees	$80.96
Estimated Strategy Capacity	$970000.00	Lowest Capacity Asset	ASB VWMXYE4L886D

Fig 2. Performance Metrics

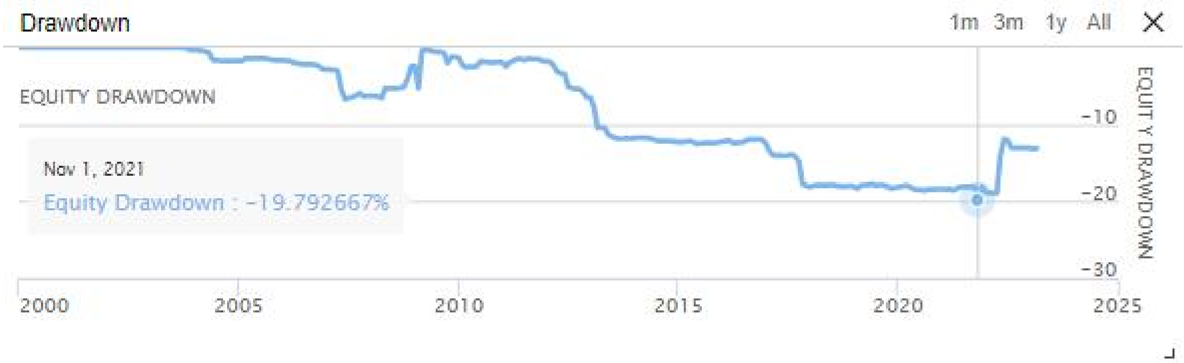


Fig 3. Drawdown

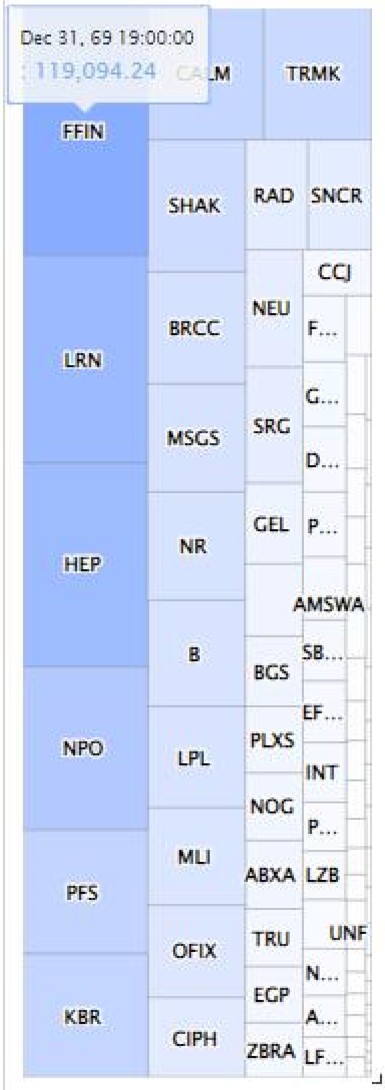


Fig 4. Assets Sales Volume