Not Over Thinking

Reversal in Post-Earnings Announcement Drift

Algorithmic Trading Strategy with Full Code

Haixiang
2023.12 | Vol 54.
hxyan.2015@gmail.com | github.com/hxyan2020

STRATEGY & ECONOMIC RATIONALE

The investment universe consists of all stocks from NYSE, AMEX, and NASDAQ with active options market (so mostly large-cap stocks). Each day investor selects stocks which would have earnings announcement during the next working day. He then checks the abnormal performance of these stocks during the previous earnings announcement. Investor goes long decile of stocks with the lowest abnormal past earnings announcement performance and goes short stocks with the highest abnormal past performance. Stocks are held for two days, and the portfolio is weighted equally.

BUY	SELL
goes long decile of stocks with the lowest abnormal past earnings announcement performance	goes short stocks with the highest abnormal past performance

PARAMETER & VARIABLES

PARAMETER	VALUE
MARKETS TRADED	Equity
FINANCIAL INSTRUMENTS	Stocks
REGION	United States
PERIOD OF REBALANCING	2 days
NO. OF TRADED INSTRUMENTS	4
WEIGHTING	Equal weighting
LOOKBACK PERIODS	Approximately a quarter
LONG/SHORT	Long & Short

ALGORITHM

<data_tools.py>

```
from AlgorithmImports import *
import numpy as np
from collections import deque
from scipy.optimize import minimize

class SymbolData:
    def __init__(self, period:int) -> None:
        self.closes:deque = deque(maxlen=period)
        self.times:deque = deque(maxlen=period)

    def update(self, time:datetime, close:float) -> None:
        self.times.append(time)
        self.closes.append(close)

    def is_ready(self) -> bool:
        return len(self.closes) == self.closes.maxlen and len(self.times) ==
self.times.maxlen

    def get_prices(self, list_period:List[datetime.date]) -> float:
```

```
        return_prices:List[float] = []

        for time, close in zip(self.times, self.closes):
            if time in list_period:
                return_prices.append(close)

        # check if there are enough data for performance calculation
        if len(return_prices) < 2:
            return None

        return (return_prices[-1] - return_prices[0]) / return_prices[0]

# Custom fee model
class CustomFeeModel(FeeModel):
    def GetOrderFee(self, parameters):
        fee = parameters.Security.Price * parameters.Order.AbsoluteQuantity * 0.00005
        return OrderFee(CashAmount(fee, "USD"))

# NOTE: Manager for new trades. It's represented by certain count of equally weighted
brackets for long and short positions.
# If there's a place for new trade, it will be managed for time of holding period.
class TradeManager():
    def __init__(self, algorithm, long_size, short_size, holding_period):
        self.algorithm = algorithm  # algorithm to execute orders in.

        self.long_size = long_size
        self.short_size = short_size

        self.long_len = 0
        self.short_len = 0

        # Arrays of ManagedSymbols
        self.symbols = []

        self.holding_period = holding_period    # Days of holding.

    # Add stock symbol object
    def Add(self, symbol, long_flag):
        # Open new long trade.
        managed_symbol = ManagedSymbol(symbol, self.holding_period, long_flag)

        if long_flag:
            # If there's a place for it.
            if self.long_len < self.long_size:
                self.symbols.append(managed_symbol)
                self.algorithm.SetHoldings(symbol, 1 / self.long_size)
                self.long_len += 1
            else:
                self.algorithm.Log("There's not place for additional trade.")

        # Open new short trade.
        else:
            # If there's a place for it.
```

```
            if self.short_len < self.short_size:
                self.symbols.append(managed_symbol)
                self.algorithm.SetHoldings(symbol, - 1 / self.short_size)
                self.short_len += 1
            else:
                self.algorithm.Log("There's not place for additional trade.")

    # Decrement holding period and liquidate symbols.
    def TryLiquidate(self):
        symbols_to_delete = []
        for managed_symbol in self.symbols:
            managed_symbol.days_to_liquidate -= 1

            # Liquidate.
            if managed_symbol.days_to_liquidate == 0:
                symbols_to_delete.append(managed_symbol)
                self.algorithm.Liquidate(managed_symbol.symbol)

                if managed_symbol.long_flag: self.long_len -= 1
                else: self.short_len -= 1

        # Remove symbols from management.
        for managed_symbol in symbols_to_delete:
            self.symbols.remove(managed_symbol)

    def LiquidateTicker(self, ticker):
        symbol_to_delete = None
        for managed_symbol in self.symbols:
            if managed_symbol.symbol.Value == ticker:
                self.algorithm.Liquidate(managed_symbol.symbol)
                symbol_to_delete = managed_symbol
                if managed_symbol.long_flag: self.long_len -= 1
                else: self.short_len -= 1

                break

        if symbol_to_delete: self.symbols.remove(symbol_to_delete)
        else: self.algorithm.Debug("Ticker is not held in portfolio!")

class ManagedSymbol():
    def __init__(self, symbol, days_to_liquidate, long_flag):
        self.symbol = symbol
        self.days_to_liquidate = days_to_liquidate
        self.long_flag = long_flag
```

<main.py>

```
from data_tools import SymbolData, CustomFeeModel, TradeManager
from AlgorithmImports import *
import numpy as np
from collections import deque
from typing import Dict, List
from pandas.tseries.offsets import BDay
```

```
from dateutil.relativedelta import relativedelta

class ReversalPostEarningsAnnouncementDrift(QCAlgorithm):

    def Initialize(self):
        self.SetStartDate(2009, 1, 1) # earnings dates starts in 2010
        self.SetCash(100000)

        self.leverage:int = 5

        self.ear_period:int = 30

        self.prev_month_year:int = -1
        self.prev_month:int = -1

        self.data:Dict[Symbol, SymbolData] = {}

        # EAR last quarter data
        self.ear_data:dict[Symbol, List[datetime.date, float]] = {}

        self.earnings_data:Dict[datetime.date, List[str]] = {}
        self.eps_data:Dict[int, Dict[int, Dict[str, Dict[datetime.date, float]]]] = {}

        self.first_date:datetime.date|None = None

        earnings_data:str =
self.Download('data.quantpedia.com/backtesting_data/economic/earnings_dates_eps.json')
        earnings_data_json:List[dict] = json.loads(earnings_data)

        for obj in earnings_data_json:
            date:datetime.date = datetime.strptime(obj['date'], '%Y-%m-%d').date()
            year:int = date.year
            month:int = date.month

            self.earnings_data[date] = []

            if not self.first_date: self.first_date = date

            for stock_data in obj['stocks']:
                ticker:str = stock_data['ticker']

                self.earnings_data[date].append(ticker)

                if stock_data['eps'] == '':
                    continue

                if year not in self.eps_data:
                    self.eps_data[year] = {}

                if month not in self.eps_data[year]:
                    self.eps_data[year][month] = {}

                if ticker not in self.eps_data[year][month]:
```

```
                self.eps_data[year][month][ticker] = {}

            self.eps_data[year][month][ticker][date] = float(stock_data['eps'])

        # EAR quarters history
        self.current_quarter_ears:List[float] = []
        self.previous_quarter_ears:List[float] = []

        # equally weighted brackets for traded symbols - 10 symbols long and short, 2 days
of holding
        self.trade_manager:TradeManager = TradeManager(self, 10, 10, 2)

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

        self.selection_flag:bool = False
        self.store_sales_data_flag:bool = True
        self.sales_growth_sort_flag:bool = False
        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:SecurityChanges) -> None:
        for security in changes.AddedSecurities:
            security.SetFeeModel(CustomFeeModel())
            security.SetLeverage(self.leverage)

    def CoarseSelectionFunction(self, coarse:List[CoarseFundamental]) -> List[Symbol]:
        # update daily prices
        for stock in coarse:
            symbol:Symbol = stock.Symbol

            if symbol in self.data:
                self.data[symbol].update(self.Time, stock.AdjustedPrice)

        # monthly selection
        if not self.selection_flag:
            return Universe.Unchanged
        self.selection_flag = False

        prev_month_date:datetime.date = (self.Time - relativedelta(months=1)).date()
        self.prev_month_year:int = prev_month_date.year
        self.prev_month:int = prev_month_date.month

        if self.prev_month_year not in self.eps_data or self.prev_month not in
self.eps_data[self.prev_month_year]:
            return Universe.Unchanged

        # select every stock, which had earnings in previous month
        stocks_with_prev_month_eps:Dict[str, Dict[datetime.date, float]] =
self.eps_data[self.prev_month_year][self.prev_month]
        selected_symbols:List[Symbol] = [x.Symbol for x in coarse if x.Symbol.Value in
stocks_with_prev_month_eps]
```

```
        for symbol in selected_symbols + [self.symbol]:
            if symbol in self.data:
                continue

            # warm up stock prices
            self.data[symbol] = SymbolData(self.ear_period)
            history = self.History(symbol, self.ear_period, Resolution.Daily)
            if history.empty:
                continue

            closes = history.loc[symbol].close
            for time, close in closes.iteritems():
                self.data[symbol].update(self.Time, close)

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

    def FineSelectionFunction(self, fine:List[FineFundamental]) -> List[Symbol]:
        for stock in fine:
            symbol:Symbol = stock.Symbol
            ticker:str = symbol.Value

            # get all stock's eps from previous month
            stock_prev_month_eps:Dict[datetime.date, float] =
self.eps_data[self.prev_month_year][self.prev_month][ticker]
            # get the date of the latest eps in previous month
            stock_latest_eps_date:datetime.date = list(stock_prev_month_eps.keys())[-1]

            # get 4 days around earnings and calculate EAR
            date_from:datetime = stock_latest_eps_date - BDay(2)
            date_to:datetime = stock_latest_eps_date + BDay(1)

            market_return:float = self.data[self.symbol].get_prices([date_from, date_to])
            stock_return:float = self.data[symbol].get_prices([date_from, date_to])

            # check if returns are ready
            if market_return and stock_return:
                ear:float = stock_return - market_return

                ear_data:List[datetime.date] = (stock_latest_eps_date, ear)
                self.ear_data[symbol] = ear_data

                # store ear in this month's history
                self.current_quarter_ears.append(ear)

        # check if there are any symbols, which can be traded
        if len(self.ear_data) == 0:
            return Universe.Unchanged

        # return symbols from self.ear_data, because they will be traded
        return list(self.ear_data.keys())

    def OnData(self, data):
```

```
        # open trades on earnings day
        date_to_lookup:datetime.date = self.Time.date()

        # if there is no earnings data yet
        if date_to_lookup < self.first_date:
            return

        # liquidate opened symbols after holding period
        self.trade_manager.TryLiquidate()

        # wait until we have history data for previous three months
        if len(self.previous_quarter_ears) == 0:
            return

        ear_values:List[float] = [x for x in self.previous_quarter_ears]
        top_ear_decile = np.percentile(ear_values, 90)
        bottom_ear_decile = np.percentile(ear_values, 10)

        # Open new trades.
        if date_to_lookup in self.earnings_data:
            symbols_to_trade:List[Symbol] = [symbol for symbol in self.ear_data if
symbol.Value in self.earnings_data[date_to_lookup]]

            symbols_to_delete:List[Symbol] = []
            for symbol in symbols_to_trade:
                # last earnings was less than three months ago
                last_earnings_date:datetime.date = self.ear_data[symbol][0]

                if last_earnings_date >= (self.Time - relativedelta(months=3)).date():
                    if symbol in data and data[symbol]:
                        if self.ear_data[symbol][1] >= top_ear_decile:
                            self.trade_manager.Add(symbol, True)
                            symbols_to_delete.append(symbol)
                        elif self.ear_data[symbol][1] <= bottom_ear_decile:
                            self.trade_manager.Add(symbol, False)
                            symbols_to_delete.append(symbol)

            # delete already traded symbols from symbol to trade
            for symbol in symbols_to_delete:
                del self.ear_data[symbol]

    def Selection(self) -> None:
        self.selection_flag = True

        if self.Time.month % 3 == 0:
            # store previous quarter's history
            self.previous_quarter_ears = [x for x in self.current_quarter_ears]
            self.current_quarter_ears.clear()
```

BACKTESTING PERFORMANCE

Fig 1. Overall Performance

Total Trades	12725	Average Win	0.61%
Average Loss	-0.63%	Compounding Annual Return	-17.339%
Drawdown	94.400%	Expectancy	-0.060
Net Profit	-93.271%	Sharpe Ratio	-0.617
Probabilistic Sharpe Ratio	0.000%	Loss Rate	52%
Win Rate	48%	Profit-Loss Ratio	0.97
Alpha	-0.108	Beta	-0.012
Annual Standard Deviation	0.177	Annual Variance	0.031
Information Ratio	-0.898	Tracking Error	0.235
Treynor Ratio	9.103	Total Fees	$3172.33

Fig 2. Performance Metrics