Momentum in REITs

Algorithmic Trading Strategy with Full Code

Haixiang

2023.11 | Vol 50.

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

STRATEGY & ECONOMIC RATIONALE

The investment universe consists of all US REITs listed on markets. Every month, the investor ranks all available REITs by their past 11-month return one-month lagged and groups them into equally weighted tercile portfolios. He/she then goes long on the best performing tercile for three months. One-third of the portfolio is rebalanced this way monthly, and REITs are equally weighted.

This is not the only way to capture the momentum factor in REITs as a consequential portfolio could be formed as a long/short or from quartiles/quintiles/deciles instead of terciles or based on different formation and holding periods (additional types of this strategy are stated in the “Other papers” section).

BUY	SELL
goes long on the best performing tercile for three months	The opposite

PARAMETER & VARIABLES

PARAMETER	VALUE
MARKETS TRADED	REITs
FINANCIAL INSTRUMENTS	Stocks
REGION	United States
PERIOD OF REBALANCING	Monthly
NO. OF TRADED INSTRUMENTS	50
WEIGHTING	Equal weighting
LOOKBACK PERIODS	11 months
LONG/SHORT	Long & Short

ALGORITHM

```
from AlgorithmImports import *

class MomentumFactorEffectinREITs(QCAlgorithm):

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

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

        # EW Trenching.
        self.holding_period = 3
        self.managed_queue = []

        self.data = {}
        self.period = 12 * 21
        self.quantile = 3
        self.leverage = 5
```

```
        self.coarse_count = 500
        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 OnSecuritiesChanged(self, changes):
        for security in changes.AddedSecurities:
            security.SetFeeModel(CustomFeeModel())
            security.SetLeverage(self.leverage)

    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, 13)
            history = self.History(symbol, self.period * 30, 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])

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

    def FineSelectionFunction(self, fine):
        fine = [x.Symbol for x in fine if (x.CompanyReference.IsREIT == 1)]

        momentum = {x : self.data[x].performance(1) for x in fine}

        long = []
        short = []

        if len(momentum) >= self.quantile:
            sorted_by_momentum = sorted(momentum.items(), key = lambda x: x[1], reverse =
True)
            quantile = int(len(sorted_by_momentum) / self.quantile)
            long = [x[0] for x in sorted_by_momentum[:quantile]]

            weight = self.Portfolio.TotalPortfolioValue / self.holding_period / len(long)
            long_symbol_q = [(symbol, np.floor(weight / self.data[symbol].prices[0])) for
symbol in long]

            self.managed_queue.append(RebalanceQueueItem(long_symbol_q))

        return long

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

        # rebalance portfolio
        remove_item = None

        for item in self.managed_queue:
            if item.holding_period == self.holding_period: # all portfolio parts are held
for n months
                for symbol, quantity in item.opened_symbol_q:
                    self.MarketOrder(symbol, -quantity)

                remove_item = item

            # trade execution
            if item.holding_period == 0: # all portfolio parts are held for n months
                opened_symbol_q = []

                for symbol, quantity in item.opened_symbol_q:
                    if symbol in data and data[symbol]:
                        self.MarketOrder(symbol, quantity)
                        opened_symbol_q.append((symbol, quantity))
```

```
                # only opened orders will be closed
                item.opened_symbol_q = opened_symbol_q

            item.holding_period += 1

        # need to remove closed part of portfolio after loop. Otherwise it will miss one
item in self.managed_queue
        if remove_item:
            self.managed_queue.remove(remove_item)

    def Selection(self):
        self.selection_flag = True

class SymbolData():
    def __init__(self, symbol, period):
        self.symbol = symbol
        self.prices = RollingWindow[float](period)

    def update(self, value):
        self.prices.Add(value)

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

    # Performance, one month skipped.
    def performance(self, values_to_skip = 0) -> float:
        closes = [x for x in self.prices][values_to_skip:]
        return (closes[0] / closes[-1] - 1)

class RebalanceQueueItem():
    def __init__(self, symbol_q):
        # symbol/quantity collections
        self.opened_symbol_q = symbol_q
        self.holding_period = 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"))
```

BACKTESTING PERFORMANCE

Fig 1. Overall Performance

PSR	0.042%	Sharpe Ratio	0.444
Total Trades	4044	Average Win	0.39%
Average Loss	-0.40%	Compounding Annual Return	10.566%
Drawdown	62.600%	Expectancy	0.301
Net Profit	936.323%	Loss Rate	34%
Win Rate	66%	Profit-Loss Ratio	0.98
Alpha	0.043	Beta	0.97
Annual Standard Deviation	0.222	Annual Variance	0.049
Information Ratio	0.26	Tracking Error	0.157
Treynor Ratio	0.102	Total Fees	$4934.96
Estimated Strategy Capacity	$6900000.00	Lowest Capacity Asset	WY R735QTJ8XC9X
Portfolio Turnover	2.18%		

Fig 2. Performance Metrics

Fig 3. Drawdown

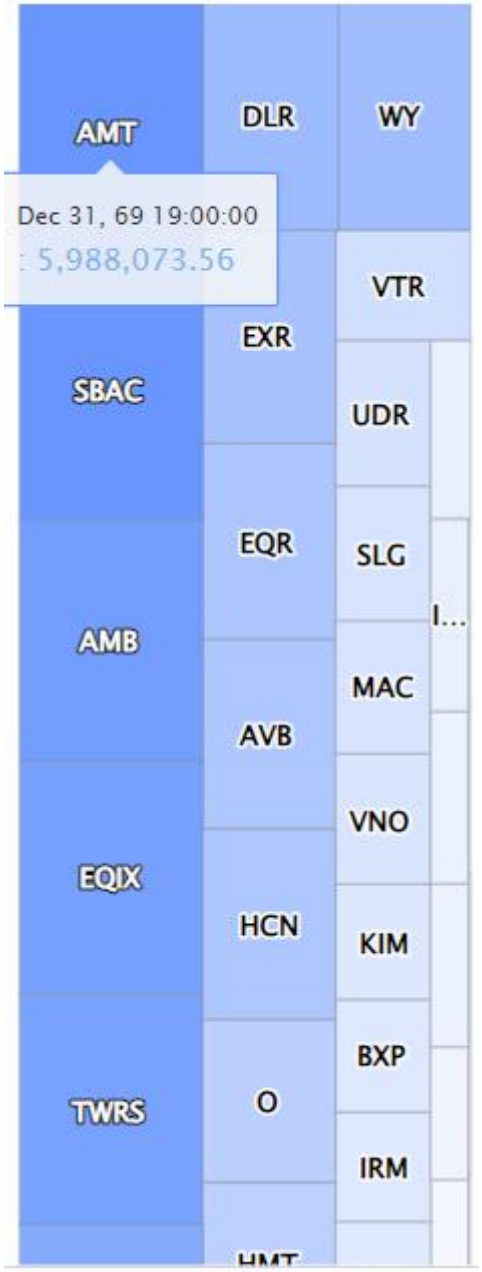

Fig 4. Assets Sales Volume