Not Over Thinking

Combining Smart Factors Momentum and Market Portfolio

Algorithmic Trading Strategy with Full Code

Haixiang

2024.01 | Vol 63.

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

STRATEGY & ECONOMIC RATIONALE

The investment universe consists of factors from the Alpha Architect's Factor Investing Data Library (factor for all major investment styles such as Value, Quality, Momentum, Size and Volatility) based on the top 1500 US stocks. Firstly construct the fast and slow signals for each factor. The fast signal is the past one-month return, and the slow signal is the past twelve-months return. For each type of signal, to obtain the weights, cross-sectionally rank signals' based on their absolute values.

The weight for the individual slow or fast signal is equal to the corresponding rank divided by the sum of all ranks and multiplied by the signal's sign (equations 3 and 4 in the paper). For the dynamically blended strategy (smart factors strategy), each factor has a final weight of three-quarters of the weight of fast signal plus one-quarter of the weight of slow signal (equation 12). Nextly, consider the top 1500 US stocks as the market portfolio. The combined smart factors and market strategy finds the weights of the market and factor portfolio using past moving averages of the returns.

The combined strategy looks back on the past twelve months, and twelve MAs of the returns. Suppose the MA for active investing (factor momentum) is larger than MA for market portfolio, then the active investing scores one point. Otherwise, the market portfolio gets one point. Therefore, each month, the weight of the factor momentum and market portfolio is determined by the number of "winning" (loosing) moving averages (equations 13 and 14). The strategy is rebalanced monthly.

BUY	SELL
(see above)	(see above)

PARAMETER & VARIABLES

PARAMETER	VALUE
MARKETS TRADED	Equity
FINANCIAL INSTRUMENTS	Srocks
REGION	Global
PERIOD OF REBALANCING	Monthly
NO. OF TRADED INSTRUMENTS	1500
WEIGHTING	Monthly
LOOKBACK PERIODS	N/A
LONG/SHORT	Long & Short

ALGORITHM

```python
from AlgorithmImports import *import numpy as np#endregion
class CombiningSmartFactorsMomentumandMarketPortfolio(QCAlgorithm):

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

        self.symbols = {
            'momentum' : 'US_EQUAL_DECILE_1500_12_2m_L_S',
            'value' : 'US_EQUAL_DECILE_1500_B_M_L_S',
            'quality' : 'US_EQUAL_DECILE_1500_ROA_L_S',
            'size' : 'US_EQUAL_DECILE_1500_Size_L_S',
            'volatility' : 'US_EQUAL_DECILE_1500_Volatility_L_S',
            }
        # monthly price data
        self.data = {}
        self.long_period = 13
        self.short_period = 2
        self.max_missing_days:int = 5

        self.monthly_returns = {}
        self.monthly_returns_period = 12

        for symbol, equity_symbol in self.symbols.items():
            data = self.AddData(USEquity, equity_symbol, Resolution.Daily)
            data.SetLeverage(10)
            data.SetFeeModel(CustomFeeModel())
            self.data[symbol] = RollingWindow[float](self.long_period)

        self.market = self.AddEquity("IWM", Resolution.Daily).Symbol
        self.data[self.market] = RollingWindow[float](self.short_period)

        self.monthly_returns['smart_factors'] = RollingWindow[float](self.monthly_returns_period)
        self.monthly_returns['market'] = RollingWindow[float](self.monthly_returns_period)

        self.recent_month:int = -1

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

    def OnData(self, data):
        # store factor monthly prices
        for symbol, equity_symbol in self.symbols.items():
            if equity_symbol in data and data[equity_symbol]:
                price = data[equity_symbol].Value
                self.data[symbol].Add(price)

        # store market prices
        if self.market in data and data[self.market]:
            market_price = data[self.market].Value
            self.data[self.market].Add(market_price)

        if self.recent_month == self.Time.month:
            return
        self.recent_month = self.Time.month

        slow_momentum = {}
        fast_momentum = {}

        # calculate both momentum values
        for symbol, equity_symbol in self.symbols.items():
            if self.Securities[equity_symbol].GetLastData() and (self.Time.date() - self.Securities[equity_symbol].GetLastData().Time.date()).days <= self.max_missing_days:
                if self.data[symbol].IsReady:
                    slow_momentum[symbol] = self.data[symbol][0] / self.data[symbol][self.long_period-1] - 1
                    fast_momentum[symbol] = self.data[symbol][0] / self.data[symbol][1] - 1

        total_weight = {}
        if len(fast_momentum) != 0:
            # momentum ranking

            # weights
            rank_sum = sum([x for x in range(1, len(slow_momentum)+1)])
            sorted_by_slow_momentum = sorted(slow_momentum.items(), key = lambda x: abs(x[1]), reverse = False)
            slow_weight = {}
            for i, (symbol, momentum) in enumerate(sorted_by_slow_momentum):
                rank = i+1
                slow_weight[symbol] = (rank / rank_sum) * np.sign(momentum)

            sorted_by_fast_momentum = sorted(fast_momentum.items(), key = lambda x: abs(x[1]), reverse = False)
            fast_weight = {}
            for i, (symbol, momentum) in enumerate(sorted_by_fast_momentum):
                rank = i+1
                fast_weight[symbol] = (rank / rank_sum) * np.sign(momentum)

            # total weight
            for symbol, equity_symbol in self.symbols.items():
                if symbol in slow_momentum and symbol in fast_momentum:
                    s_weight = slow_weight[symbol]
                    f_weight = fast_weight[symbol]
                    total_weight[symbol] = 0.75*f_weight + 0.25*s_weight

        # retrun calculation for market and smart factors
        if self.data[self.market].IsReady:
            market_return = self.data[self.market][0] / self.data[self.market][1] - 1
            self.monthly_returns['market'].Add(market_return)

            # smart factor return calculation
            smart_factors_return = 0
            for symbol, momentum_1M in fast_momentum.items():
                if symbol in total_weight:
                    w = total_weight[symbol]
                    symbol_ret = w*momentum_1M
                    smart_factors_return += symbol_ret

            if smart_factors_return != 0:
                self.monthly_returns['smart_factors'].Add(smart_factors_return)

            score = {}
            traded_weight = {}

            # calculate 12 SMA's
            if self.monthly_returns['smart_factors'].IsReady and self.monthly_returns['market'].IsReady:
                score['smart_factors'] = 0
                score['market'] = 0
                for sma_period in range(1, 13):
                    factor_returns = [x for x in self.monthly_returns['smart_factors']][:sma_period]
                    market_returns = [x for x in self.monthly_returns['market']][:sma_period]

                    factor_mean_return = np.mean(factor_returns)
                    market_mean_return = np.mean(market_returns)

                    if factor_mean_return > market_mean_return:
                        score['smart_factors'] += 1
                    else:
                        score['market'] += 1

                total_score = score['market'] + score['smart_factors']
                if total_score != 0:
                    traded_weight['market'] = score['market'] / total_score
                    traded_weight['smart_factors'] = score['smart_factors'] / total_score

                    # order execution
                    # market
                    self.SetHoldings(self.market, traded_weight['market'])

                    # smart factors
                    for symbol, equity_symbol in self.symbols.items():
                        if symbol in total_weight:
                            w = total_weight[symbol]
                            self.SetHoldings(equity_symbol, traded_weight['smart_factors'] * w)
                            class USEquity(PythonData):
    def GetSource(self, config, date, isLiveMode):
        return SubscriptionDataSource("data.quantpedia.com/backtesting_data/equity/us_ew_decile/{0}.
csv".format(config.Symbol.Value), SubscriptionTransportMedium.RemoteFile, FileFormat.Csv)

    # File example.
    # date;equity
    # 1992-01-31;0.98
    def Reader(self, config, line, date, isLiveMode):
        data = USEquity()
        data.Symbol = config.Symbol

        if not line[0].isdigit(): return None
        split = line.split(';')

        # Prevent lookahead bias.
        data.Time = datetime.strptime(split[0], "%Y-%m-%d") + timedelta(days=1)
        data.Value = float(split[1])

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

BACKTESTING PERFORMANCE

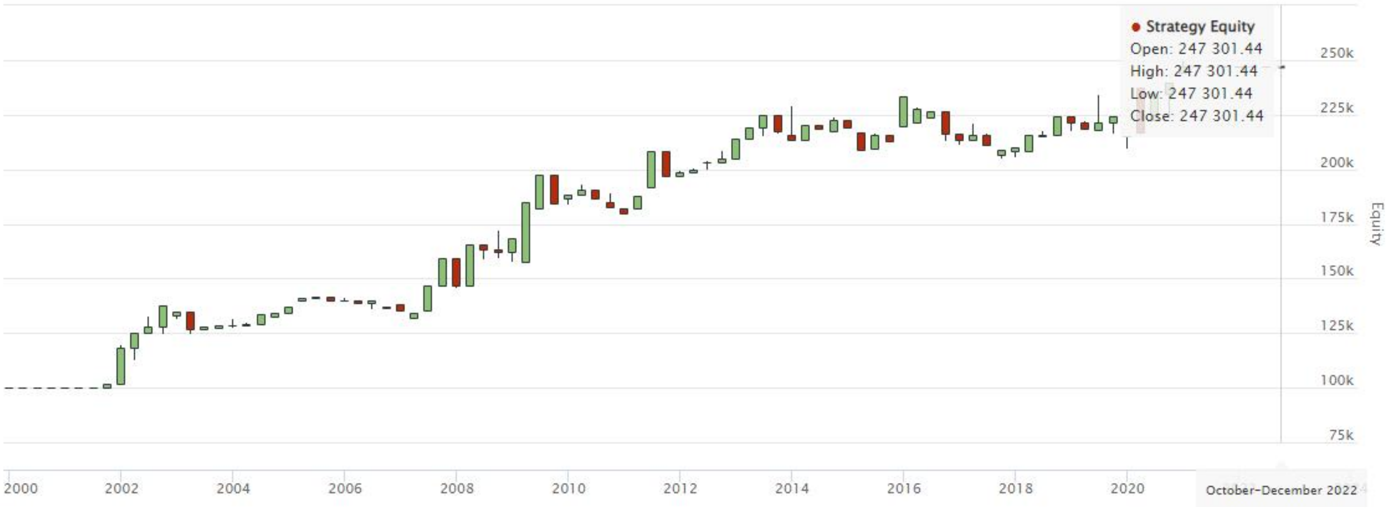
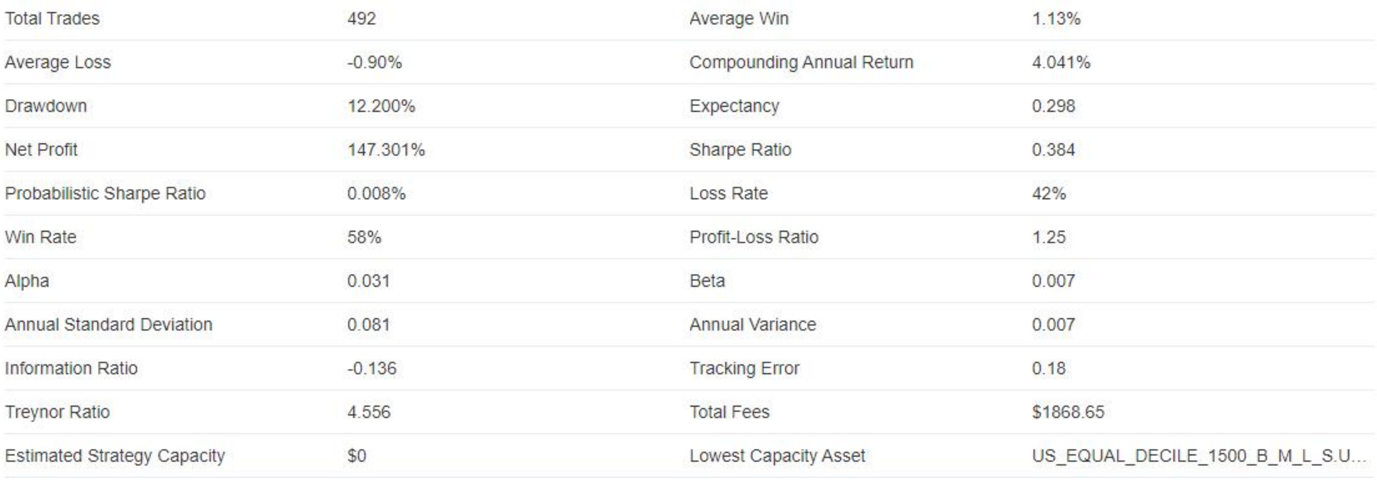


Fig 1. Overall Performance



Total Trades	492	Average Win	1.13%
Average Loss	-0.90%	Compounding Annual Return	4.041%
Drawdown	12.200%	Expectancy	0.298
Net Profit	147.301%	Sharpe Ratio	0.384
Probabilistic Sharpe Ratio	0.008%	Loss Rate	42%
Win Rate	58%	Profit-Loss Ratio	1.25
Alpha	0.031	Beta	0.007
Annual Standard Deviation	0.081	Annual Variance	0.007
Information Ratio	-0.136	Tracking Error	0.18
Treynor Ratio	4.556	Total Fees	$1868.65
Estimated Strategy Capacity	$0	Lowest Capacity Asset	US_EQUAL_DECILE_1500_B_M_L_S.U...

Fig 2. Performance Metrics