Not Over Thinking

Technical Indicators Predict Cross - Sectional Expected Stock Returns

Algorithmic Trading Strategy with Full Code

Haixiang

2024.02 | Vol 70.

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

STRATEGY & ECONOMIC RATIONALE

The investment universe consists of all firms from the CRSP database listed on NYSE, AMEX, and NASDAQ. Firstly, exclude all firms with less than 60 monthly return observations. Secondly, construct 14 firm-level technical indicators based on three trend-following strategies (moving average, momentum, and volume-based indicators).

The first strategy is based on the moving average rule, which forms the trading signals by comparing the two moving averages with different lengths.

The second strategy is based on the momentum trading rule, which generates the trading signals by comparing the current stock price with its level n months ago.

The third strategy is based on the "on-balance" volume rule, which generates the trading signals by evaluating the changes in stock trading volume. For a detailed description of the technical indicators' construction, see section 2.2. Thirdly, each month t regress the return of each stock i on 14 technical indicators from month t-1, using a fixed window of the latest 60 monthly observations to estimate the return over the next month (see equations 5 and 6).

To mitigate the overfitting problem, take the time-series average of the cross-sectional OLS estimated coefficients applying a 60-month smoothing window (see equations 7a, 7b, and 7c). At the end of each month, sort all stocks into value-weighted deciles based on their estimated returns in the next month.

Buy the top decile (stocks with the highest expected returns) and sell the bottom decile (stocks with the lowest expected returns). The resulting long-short portfolio is value-weighted and rebalanced monthly.

BUY	SELL
Buy the top decile (stocks with the highest expected returns)	sell the bottom decile (stocks with the lowest expected returns)

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	N/A
LONG/SHORT	Long only

ALGORITHM

```
from AlgorithmImports import *import statsmodels.api as sm# endregion
class TechnicalIndicatorsPredictCrossSectionalExpectedStockReturns(QCAlgorithm):

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

```
        self.quantile:int = 10
        self.month_period:int = 21
        self.regression_period:int = 60
        self.period:int = self.month_period * 12
        self.long_periods:list[int] = [9 * self.month_period, 12 * self.month_period]
        self.short_periods:list[int] = [1* self.month_period, 2 * self.month_period, 3 * self.month_period]

        self.last_fine:list = []

        self.data:dict = {}
        self.weights:dict = {}

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

        self.coarse_count:int = 500
        self.selection_flag:bool = False
        self.UniverseSettings.Resolution = Resolution.Daily
        self.AddUniverse(self.CoarseSelectionFunction, self.FineSelectionFunction)

        self.Schedule.On(self.DateRules.MonthStart(self.symbol), self.TimeRules.BeforeMarketClose(self.symbol, 0), self.Selection)

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

    def CoarseSelectionFunction(self, coarse):
        # update stocks data on daily basis
        for stock in coarse:
            symbol:Symbol = stock.Symbol

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

        if not self.selection_flag:
            return Universe.Unchanged

        selected:list = sorted([x for x in coarse if x.HasFundamentalData and x.Market == 'usa'],
                key=lambda x: x.DollarVolume, reverse=True)[:self.coarse_count]

        # warm up stock's data
        for stock in selected:
            symbol:Symbol = stock.Symbol

            if symbol not in self.data:
                self.data[symbol] = SymbolData(symbol, self.short_periods, self.long_periods, self.period)
                history = self.History(symbol, self.period, Resolution.Daily)
                if history.empty:
                    continue

                closes = history.loc[symbol].close
                volumes = history.loc[symbol].volume

                for (_, close), (_, volume) in zip(closes.iteritems(), volumes.iteritems()):
                    self.data[symbol].update(close, volume)

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

    def FineSelectionFunction(self, fine):
```

```
        fine = [x for x in fine if x.MarketCap != 0 and ((x.SecurityReference.ExchangeId == "NY
S") or (x.SecurityReference.ExchangeId == "NAS") or (x.SecurityReference.ExchangeId == "ASE"))]

        pred_returns:dict = {}

        for stock in fine:
            symbol:Symbol = stock.Symbol
            symbol_obj = self.data[symbol]

            # make sure data are consecutive
            if symbol not in self.last_fine:
                symbol_obj.clear_regression_data()

            # make sure regression data are ready
            if symbol_obj.is_regression_data_ready(self.regression_period):
                regression_x, regression_y = symbol_obj.get_regression_data(self.regression_per
iod)
                x_transpose:np.array = np.array(regression_x).T

                # skip x series with the same value throughout the whole series since there's n
ot clear decision to make for which zeroed series should be intercept
                x_variable_skip_indices:list[int] = self.GetIndicesOfSameValues(x_transpose=x_t
ranspose)

                # use adjusted x variable for model building and for prediction
                adjusted_x_variable:list = [x for i, x in enumerate(x_transpose) if i not in x_
variable_skip_indices]
                regression_x:np.array = np.array(adjusted_x_variable).T
                regression_model = sm.OLS(endog=regression_y, exog=regression_x).fit()
                regression_params:list[float] = list(regression_model.params)

                # update this month regression data
                symbol_obj.update_returns(self.month_period)
                symbol_obj.update_technical_indicators(self.long_periods)

                if symbol_obj.is_smoothing_window_ready(self.regression_period):
                    pred_params:list = symbol_obj.get_prediction_params(self.regression_period)
                    pred_x:list = symbol_obj.get_prediction_x()

                    # predict price based on previous technical indicators
                    stock_pred_return:float = self.CalcStockPrediction(pred_params, pred_x)

                    pred_returns[stock] = stock_pred_return

                # update smoothing window
                smoothing_window_entry:list[float] = []
                for i, x_series in enumerate(x_transpose):
                    if i in x_variable_skip_indices:
                        smoothing_window_entry.append(0)
                    else:
                        smoothing_window_entry.append(regression_params.pop(0))

                symbol_obj.update_smoothing_window(smoothing_window_entry)

            else:
                # update this month regression data
                symbol_obj.update_returns(self.month_period)
                symbol_obj.update_technical_indicators(self.long_periods)

        # last_fine helps to secure data consecution
        self.last_fine = [x.Symbol for x in fine]

        # make sure there are enough stock for selection
        if len(pred_returns) < self.quantile:
```

```
            return Universe.Unchanged

        quantile = int(len(pred_returns) / self.quantile)
        sorted_by_pred_returns = [x[0] for x in sorted(pred_returns.items(), key=lambda item: i
tem[1])]

        # buy stocks with the highest expected return
        long_part = sorted_by_pred_returns[-quantile:]

        # sell stocks with the lowest expected return
        short_part = sorted_by_pred_returns[:quantile]

        total_long_cap = sum([x.MarketCap for x in long_part])
        for stock in long_part:
            self.weights[stock.Symbol] = stock.MarketCap / total_long_cap

        total_short_cap = sum([x.MarketCap for x in short_part])
        for stock in short_part:
            self.weights[stock.Symbol] = -stock.MarketCap / total_short_cap

        return [x for x in self.weights]

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

        # trade execution
        invested:list = [x.Key for x in self.Portfolio if x.Value.Invested]
        for symbol in invested:
            if symbol not in self.weights:
                self.Liquidate(symbol)

        for symbol, w in self.weights.items():
            if self.Securities[symbol].Price != 0 and self.Securities[symbol].IsTradable:
                self.SetHoldings(symbol, w)

        self.weights.clear()

    def GetIndicesOfSameValues(self, x_transpose:np.array) -> list:
        x_variable_skip_indices:list= []

        for i, x_series in enumerate(x_transpose):
            # don't skip intercept
            if i != 0 and all(x_series[0] == x for x in x_series):
                x_variable_skip_indices.append(i)

        return x_variable_skip_indices

    def CalcStockPrediction(self, pred_params:list, pred_x:list) -> float:
        pred_value:float = 0

        for param, x_value in zip(pred_params, pred_x):
            pred_value += param * x_value

        return pred_value

    def Selection(self):
        self.selection_flag = True

class SymbolData():
    def __init__(self, symbol:Symbol, short_periods:list, long_periods:list, period:float) -> N
one:
```

```
        self.short_SMA:list = []
        self.long_SMA:list = []

        self.long_volumes:list = []
        self.short_volumes:list = []

        self.technical_indicators:list = []
        self.returns:list = []

        self.smoothing_window:list = []

        self.prices:RollingWindow = RollingWindow[float](period)

        for period in short_periods:
            self.short_SMA.append(RollingWindow[float](period))
            self.short_volumes.append(RollingWindow[float](period))

        for period in long_periods:
            self.long_SMA.append(RollingWindow[float](period))
            self.long_volumes.append(RollingWindow[float](period))

    def update(self, stock_price:float, stock_volume:float) -> None:
        for short_SMA_roll_win, short_volume in zip(self.short_SMA, self.short_volumes):
            short_SMA_roll_win.Add(stock_price)
            short_volume.Add(stock_volume)

        for long_SMA_roll_win, long_volume in zip(self.long_SMA, self.long_volumes):
            long_SMA_roll_win.Add(stock_price)
            long_volume.Add(stock_volume)

        self.prices.Add(stock_price)

    def is_ready(self) -> bool:
        for short_SMA_roll_win, short_volume in zip(self.short_SMA, self.short_volumes):
            if not short_SMA_roll_win.IsReady or not short_volume.IsReady:
                return False

        for long_SMA_roll_win, long_volume in zip(self.long_SMA, self.long_volumes):
            if not long_SMA_roll_win.IsReady or not long_volume.IsReady:
                return False

        return self.prices.IsReady

    def is_regression_data_ready(self, regression_period:int) -> bool:
        return len(self.technical_indicators) >= regression_period and len(self.returns) >= reg
ression_period

    def is_smoothing_window_ready(self, regression_period:int) -> bool:
        return len(self.smoothing_window) >= regression_period

    def clear_regression_data(self):
        self.technical_indicators.clear()
        self.smoothing_window.clear()
        self.returns.clear()

    def update_returns(self, period:int):
        # make sure between regression x and y is right shift
        if len(self.technical_indicators) > 0:
            prices:list = [x for x in self.prices][:period]
            return_value:float = (prices[0] - prices[-1]) / prices[-1]

            self.returns.append(return_value)

    def update_technical_indicators(self, periods:list) -> list:
```

```
        technical_indicators_values:list = []

        # MA and OBV technical indicators
        for long_SMA_roll_win, long_volume in zip(self.long_SMA, self.long_volumes):
            mean_long_volume:float = np.mean([x for x in long_volume])
            long_SMA_value:float = self.calc_simple_moving_average([x for x in long_SMA_roll_wi
n])

            for short_SMA_roll_win, short_volume in zip(self.short_SMA, self.short_volumes):
                mean_short_volume:float = np.mean([x for x in short_volume])
                short_SMA_value:float = self.calc_simple_moving_average([x for x in short_SMA_r
oll_win])

                if long_SMA_value > short_SMA_value:
                    technical_indicators_values.append(0)
                else:
                    technical_indicators_values.append(1)

                if mean_long_volume > mean_short_volume:
                    technical_indicators_values.append(0)
                else:
                    technical_indicators_values.append(1)

        prices:list = [x for x in self.prices]
        curr_price:float = prices[0]

        # MOM technical indicators
        for period in periods:
            if curr_price >= prices[period - 1]:
                technical_indicators_values.append(1)
            else:
                technical_indicators_values.append(0)

        self.technical_indicators.append(technical_indicators_values)

    def update_smoothing_window(self, smoothing_window_entry:list):
        self.smoothing_window.append(smoothing_window_entry)

    def calc_simple_moving_average(self, prices:list) -> float:
        return sum(prices) / len(prices)

    def get_regression_data(self, regression_period:int) -> list:
        x = self.technical_indicators[-regression_period:]
        # add constant
        x = [[1] + tech_indi for tech_indi in x]
        y = self.returns[-regression_period:]

        return x, y

    def get_prediction_params(self, regression_period:int) -> list:
        window_transpose:np.array = np.array(self.smoothing_window[-regression_period:]).T
        params:list = [np.mean(params_list) for params_list in window_transpose]

        return params

    def get_prediction_x(self) -> list:
        last_indicators:list = self.technical_indicators[-1]
        return [1] + last_indicators
# 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	8644	Average Win	0.36%
Average Loss	-0.34%	Compounding Annual Return	1.278%
Drawdown	38.700%	Expectancy	0.022
Net Profit	34.393%	Sharpe Ratio	0.145
Probabilistic Sharpe Ratio	0.000%	Loss Rate	50%
Win Rate	50%	Profit-Loss Ratio	1.05
Alpha	0.013	Beta	-0.006
Annual Standard Deviation	0.087	Annual Variance	0.008
Information Ratio	-0.245	Tracking Error	0.185
Treynor Ratio	-1.986	Total Fees	$3596.16
Estimated Strategy Capacity	$860000000.00	Lowest Capacity Asset	COF R735QTJ8XC9X
Portfolio Turnover	5.91%		

Fig 2. Performance Metrics