Factor-based analysis and algorithm design

Note: This document is a translated version of a report originally written in Chinese. The translation was done using machine translation tools, it may contain inaccuracies or inconsistencies. Additionally, this report was produced during my past internship experience and may not fully reflect my current level of expertise or capabilities.Thank you for your understanding.

Table of Contents

1. single factor analysis

Factor name 1	Book-to-market ratio (BP)
intuitive interpretation	Book value is the net worth of a company's assets as recorded in its financial statements, i.e., total assets minus total liabilities. It represents the intrinsic value of the company or the value of shareholders' equity. A company's market capitalization is the current market price of its stock multiplied by the total number of shares outstanding. It reflects the total value that investors are willing to pay for the company. When the BP ratio is high, it means that the company's book value is high relative to its market value, which may indicate that the stock is undervalued and investors are paying less than the company's intrinsic value. Conversely, if the BP ratio is low, it may mean that the company's stock is overvalued or that the market has high expectations for the company's future growth.
Detailed description (calculations)	
book value = total assets - total liabilities Market capitalization = current price of the stock x total number of shares issued and outstanding book-to-market ratio (BP) = book value/market value	

Factor name 2	the full changeover days factor
intuitive interpretation	The Days to Full Turnover factor measures the number of days it takes for a stock to reach 100% cumulative turnover in a given period of time and is a sentiment factor. If a stock reaches 100% cumulative turnover very quickly, i.e., has a low number of days to full turnover, this is usually an indication that the stock has high trading activity and liquidity. In some cases, this can be seen as a sign of high market sentiment, which may signal the potential for higher prices in the short term. Conversely, if a stock takes a longer time to reach a cumulative turnover rate of 100 percent, i.e., a higher number of days to full turnover, this could mean that the stock is trading in low volumes and that there is insufficient liquidity and market participation. This may be because the stock is less visible to the market or investors are cautious about the company's prospects. However, the performance of this factor is unstable and may reverse based on shifting market conditions and different stock pools selected.
Detailed description (calculations)	
1. Determining the timeframe for calculating the cumulative turnover rate 2,Collect the daily turnover data of the stock for the selected time frame. Turnover rate (turnover) = (turnover over a certain period of time)/(total number of shares issued) x 100% 3,Starting from the latest day, the daily turnover rate is accumulated until the cumulative turnover rate reaches or exceeds 100% for the first time. 4. Record the number of days required to reach 100% cumulative turnover	

Factor name 3	Price factor without reweighting (price_no_fq)
intuitive	Based on a reverse investment strategy, whereby one buys stocks that are

interpretation	currently underpriced and sells stocks that are currently overpriced. This is similar to the "buy undervalued, sell overvalued" strategy. Under certain market conditions, underpriced stocks may have the potential to rebound, while overpriced stocks may face the risk of a pullback. In this case, lower-priced stocks may be undervalued, while higher-priced stocks may be overvalued. Over time, the market may correct these mispricings, providing positive returns for lower-priced stocks and negative returns for higher-priced stocks.
Detailed description (calculations)	
directly using the current market price of the stock, which has not been subject to any weighting	

2. single-factor trading algorithms that

algorithmic factors	Book-to-market ratio (BP)
A detailed description of the intuition behind the single-factor algorithm that	
The strategy uses BP (book-to-market ratio) as the core factor to find stocks with higher expected returns. Typically, a low book-to-market ratio implies that a stock is undervalued and may have higher future returns. This strategy uses the CSI 800 Index (symbol 000906.XSHG) as the benchmark stock pool, using the past 21 days of data to observe the performance of stocks and calculate the factor value to reflect the short-term fundamental changes in stocks, the stocks in the pool are ranked according to the BP value (descending), the strategy focuses on only the top 10% of the ranked stocks. At the end of each month, the strategy evaluates whether or not to trade. If it is a trading day, the positions are updated based on the factor values, the top 10% of stocks are bought, and the investments are made in proportion to the established positions.	

Below is the code for the implemented algorithm

```
import datetime
import numpy as np
import pandas as pd
import time
from jqdata import *
from pandas import Series, DataFrame
import statsmodels.api as sm
from jqfactor import get_factor_values

def initialize(context):
set_params() #1 sets the curated parameters
set_variables() #2 sets the intermediate variables, the
set_backtest() #3 set the backtest condition that

#1 Set the policy parameters
def set_params():
```

```
g.factor = 'BP' # the current backtested single factor
g.shift = 21 # Set an observation day (days)
g.precent = 0.10 # Positions as a percentage of the optional stock pool, the
g.index='000906.XSHG' # Define stock pool, CSI 800

# Define the factors and how to rank them, extract BP single factor analysis from multiple fundamental factors
   g.factors = {'BP': False, 'net_profit_increase':True,'inc_net_profit_year_on_year':True,'operating_profit':True,
   'inc_revenue_year_on_year':True
   }

   g.sort_rank = True
g.quantile = (0, 10) # Focus on stocks that are in the top 10% (0 to 10th percentile) of the sort, the

#2 Setting of intermediate variables
def set_variables():
g.feasible_stocks = [] # the current pool of tradable stocks
g.if_trade = False # Whether or not the day is traded
g.num_stocks = 0 # Set the number of stocks to hold

#3 Setting up backtesting conditions
def set_backtest():
set_benchmark('000906.XSHG') # set as benchmark
set_option('use_real_price', True) # Trade at the real price
log.set_level('order', 'error') # Set the error reporting level of the

#Things to do every day before the opening bell
def before_trading_start(context):
# Get the current date, #
   day = context.current_dt.day
   yesterday = context.previous_date
   rebalance_day = shift_trading_day(yesterday, 1)
   if yesterday.month != rebalance_day.month:
      if yesterday.day > rebalance_day.day:
         g.if_trade = True
#5 Set feasible stock pools: get the current open stock pool and exclude stocks that are currently or during the calculation sample
period suspended from trading
         g.feasible_stocks = set_feasible_stocks(get_index_stocks(g.index), g.shift,context)
               #6 Setting of slippage and handling charges
         set_slip_fee(context)
# Purchase stocks in proportion to the pool of viable stocks
         g.num_stocks = int(len(g.feasible_stocks)*g.precent)

#4
def shift_trading_day(date,shift):
# Get all trading days, return a list of all trading days with datetime.date type.
   tradingday = get_all_trade_days()
# Get the line number in the list for the day after date shift day Returns a number
   shiftday_index = list(tradingday).index(date)+shift
# Returns the date of the day based on the line number as a datetime.date type
   return tradingday[shiftday_index]

#5
# Setting up a pool of viable stocks

def set_feasible_stocks(stock_list,days,context):
# dataframe with information about whether the license is suspended or not, 1 for suspended, 0 for not suspended.
   suspened_info_df = get_price(list(stock_list),
                  start_date=context.current_dt,
```

```
                end_date=context.current_dt,
                frequency='daily',
                fields='paused'
    )['paused'].T
# Filter for suspended stocks return dataframe
    unsuspened_index = suspened_info_df.iloc[:,0]<1
# Get the codes of the stocks that are not suspended for the day list.
    unsuspened_stocks = suspened_info_df[unsuspened_index].index
# Further, screen the list of stocks that have not been suspended in the last few days.
    feasible_stocks = []
    current_data = get_current_data()
    for stock in unsuspened_stocks:
        if sum(attribute_history(stock,
                                 days,
                                 unit = '1d',
                                 fields = ('paused'),
                                 skip_paused = False
                                )
            )[0] == 0:
            feasible_stocks.append(stock)
# Excluding ST shares
    st_data = get_extras('is_st', feasible_stocks, end_date = context.previous_date, count = 1)
    stockList = [stock for stock in feasible_stocks if not st_data[stock][0]]
    return stockList

#6 Slippage and handling fees based on different time periods
def set_slip_fee(context):
# Set the slippage point to 0
    set_slippage(FixedSlippage(0))
# Setting handling fees based on different time periods
    dt=context.current_dt

    if dt>datetime.datetime(2013,1, 1):
        set_commission(PerTrade(buy_cost=0.0003,
                                sell_cost=0.0013,
                                min_cost=5))

    elif dt>datetime.datetime(2011,1, 1):
        set_commission(PerTrade(buy_cost=0.001,
                                sell_cost=0.002,
                                min_cost=5))

    elif dt>datetime.datetime(2009,1, 1):
        set_commission(PerTrade(buy_cost=0.002,
                                sell_cost=0.003,
                                min_cost=5))

    else:
        set_commission(PerTrade(buy_cost=0.003,
                                sell_cost=0.004,
                                min_cost=5))

def handle_data(context, data):
# If it's a trading day
    if g.if_trade == True:
# 7 Get buy and sell signals, input context, output stock list list
# The corresponding default value in the dictionary is false holding_list is filtered to true, then the factor with the highest score is
selected
```

```
        holding_list = get_stocks(g.feasible_stocks, context, asc=g.sort_rank)
# new additions, calculating holding_list length, # new additions, calculating holding_list length
        total_number = len(holding_list)
        # print 'feasible_stocks is %d, holding is %d' % (len(g.feasible_stocks), total_number)
# Extract the required quantile information
        (start_q, end_q) = g.quantile
# 8 Rebalance position, enter context, use signal result holding_list
        rebalance(context, holding_list, start_q, end_q, total_number)
g.if_trade = False # Assignment

#7 Raw Data Re-Extraction Factor Scoring Ranking
def get_stocks(stocks_list, context, asc):
# Construct a new string, called 'get_df_' + 'key'.
    tmp='get_df' + '_' + g.factor
# Declare that the string is an equation
    aa = globals()[tmp](stocks_list, context, g.factors[g.factor])
#3 times the standard deviation to the extremes
    #aa = winsorize(aa,g.factor,std = 3,have_negative = True)
#z standardization
    #aa = standardize(aa,g.factor,ty = 2)

# Delete nan in case an item in the data does not generate nan
    #aa = aa[pd.notnull(aa['BP'])]
# Generate ranking ordinal numbers
    #aa['BP_sorted_rank'] = aa['BP'].rank(ascending = asc, method = 'dense')
    score = g.factor + '_' + 'sorted_rank'
    #stocks = list(aa.sort(score, ascending = asc).index)
    stocks = list(aa.sort_values(by=score, ascending=asc).index)
    print (stocks)
    return stocks

#8
# Get a list of stocks to buy based on the strategy's buy signals

def rebalance(context, holding_list, start_q, end_q, total_number):
    if end_q == 100:
        end_q = 100
# Amount purchased per stock
    every_stock = context.portfolio.portfolio_value/g.num_stocks
# Empty positions only buy operations
    if len(list(context.portfolio.positions.keys()))==0:
# The original reset scort starts with a return-related scoring calculation, where returns are sorted in ascending order
        # for stock_to_buy in holding_list[int(round(start_q * total_number / 100)) : int(round(end_q * total_number / 100))]:
        for stock_to_buy in holding_list[int(round(start_q * total_number / 100)) : int(round(end_q * total_number / 100))]:
            order_target_value(stock_to_buy, every_stock)
    else :
# Not a short position to first sell shares held but not on the buy list
        for stock_to_sell in list(context.portfolio.positions.keys()):
            if stock_to_sell not in holding_list[int(round(start_q * total_number / 100)) : int(round(end_q * total_number / 100))]:
                order_target_value(stock_to_sell, 0)
# Because the order function is adjusted to the order of adjustment, in order to prevent the first line to adjust the position of the stock due to the latter line to adjust the position of the stock accounted for the amount of too large can not be adjusted in place at once, here to run twice to solve this problem
        for stock_to_buy in holding_list[int(round(start_q * total_number / 100)) : int(round(end_q * total_number / 100))]:
            order_target_value(stock_to_buy, every_stock)
        for stock_to_buy in holding_list[int(round(start_q * total_number / 100)) : int(round(end_q * total_number / 100))]:
            order_target_value(stock_to_buy, every_stock)
# BP
```

```
# Get a dataframe: containing the ticker symbol, book-to-market ratio BP and corresponding rank BP_sorted_rank
# Default date = the day before context.current_dt
def get_df_BP(stock_list, context, asc):
    df_BP = get_fundamentals(query(valuation.code, valuation.pb_ratio
                    ).filter(valuation.code.in_(stock_list)))
# Get the pb countdown, #
    df_BP['BP'] = df_BP['pb_ratio'].apply(lambda x: 1/x)
# Delete nan in case an item in the data does not generate nan
    df_BP = df_BP[pd.notnull(df_BP['BP'])]
# Generate ranking ordinal numbers
    df_BP['BP_sorted_rank'] = df_BP['BP'].rank(ascending = asc, method = 'dense')
# Use the ticker symbol as the index
    df_BP.index = df_BP.code
# Delete useless data
    del df_BP['code']
    print (df_BP)
    return df_BP

# Depolarization function (3x standard deviation depolarization)
def winsorize(factor_data, factor, std=3,  have_negative = True):

    r=factor_data[factor]
    if have_negative == False:
        r = r[r>=0]
    else:
        pass
# Taking extreme values
    edge_up = r.mean()+std*r.std()
    edge_low = r.mean()-std*r.std()
    r[r>edge_up] = edge_up
    r[r<edge_low] = edge_low
    r = pd.DataFrame(r)
     return r

# z-score standardized function.
def standardize(factor_data,factor,ty=2):

    temp=factor_data[factor]
    if int(ty)==1:
        re = (temp - temp.min())/(temp.max() - temp.min())
    elif ty==2:
        re = (temp - temp.mean())/temp.std()
elif ty==3
re = temp/10**np.ceil(np.log10(temp.abs().max()))
     return pd.DataFrame(re)
```

Condition settings and results of experiments with book-to-market ratio factor algorithms

1. Sample screening

Sample range: CSI 800 Index Constituents
Test Sample Period:2023-01-01 to 2023-12-31
(1) Eliminate stocks with ST/PT on the stock selection date.
(2) Exclusion of stocks listed for less than one year.
(3) Excluding stocks that cannot be purchased on the stock selection date due to suspension of trading or other reasons.

2. Data cleansing

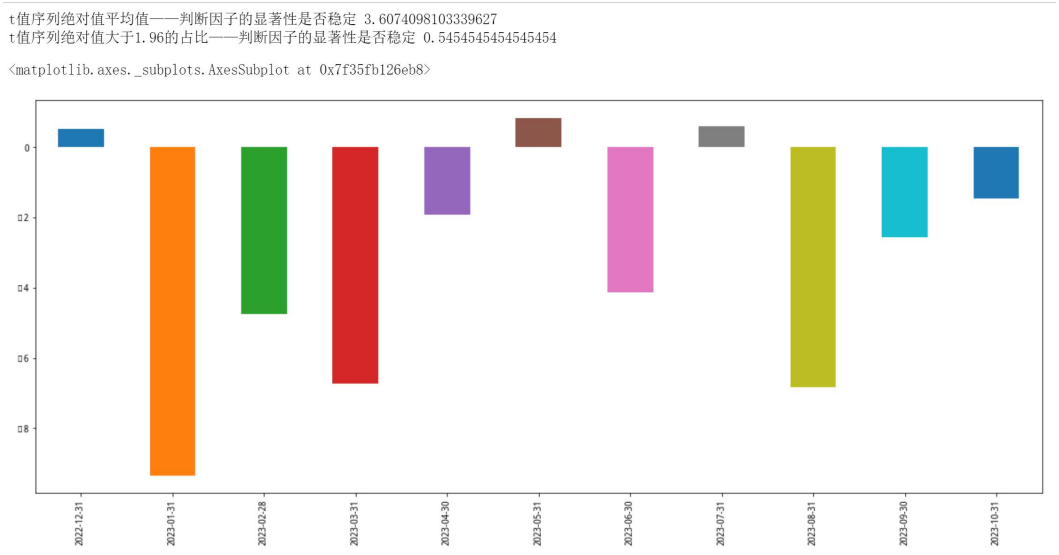
The winsorize shrinkage process was used to take a 95% confidence interval, and outliers were defined as outliers. For outliers, the value is changed to the boundary of the 95% confidence interval.

3. factor standardization

Rank standardized processing factor data.

4. factor validity analysis

(1) T-test

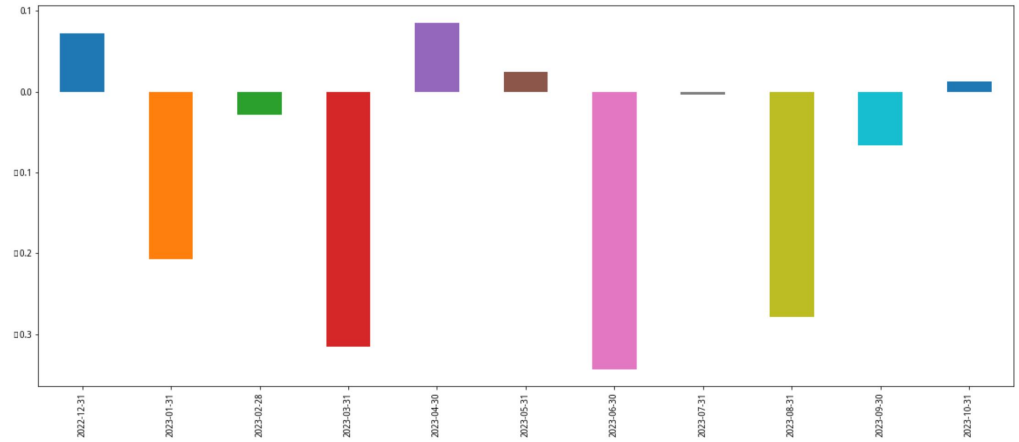


The mean absolute value of t-value is 3.6074, which is much higher than 2, indicating that the BP factor is statistically significant. More than half of the t-values have an absolute value greater than 1.96, indicating that the BP factor is significant at the 95% confidence level most of the time.

(2) IC Analysis

```
IC 值序列的均值大小 -0.09559535211025177
IC 值序列的标准差 0.16008204218330424
IR 比率（IC值序列均值与标准差的比值） -0.5971647463166977
IC 值序列大于零的占比 0.36363636363636365
IC 值序列绝对值大于0.02的占比 0.8181818181818182
```

: <matplotlib.axes._subplots.AxesSubplot at 0x7f35fb130be0>

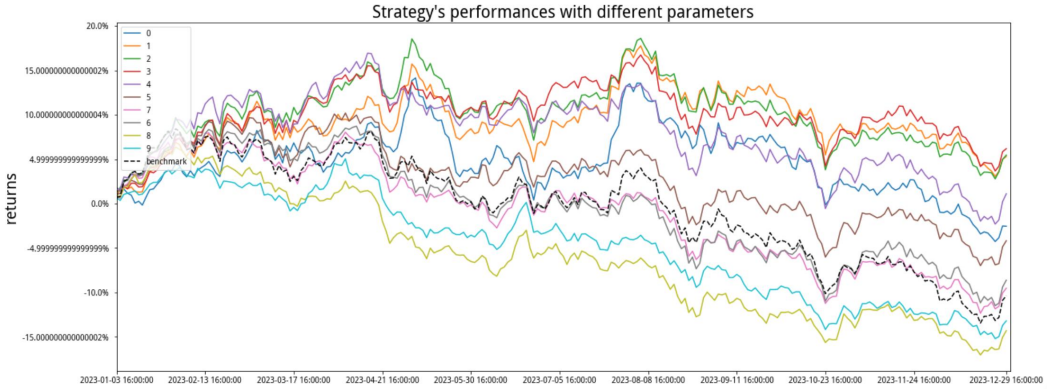


It shows that the BP factor is negatively correlated with future returns, and the IC value fluctuates a lot, so the predictive ability of the factor is unstable, but the absolute value of the IC value is higher than 0.02, which indicates that the factor still has some predictive significance.

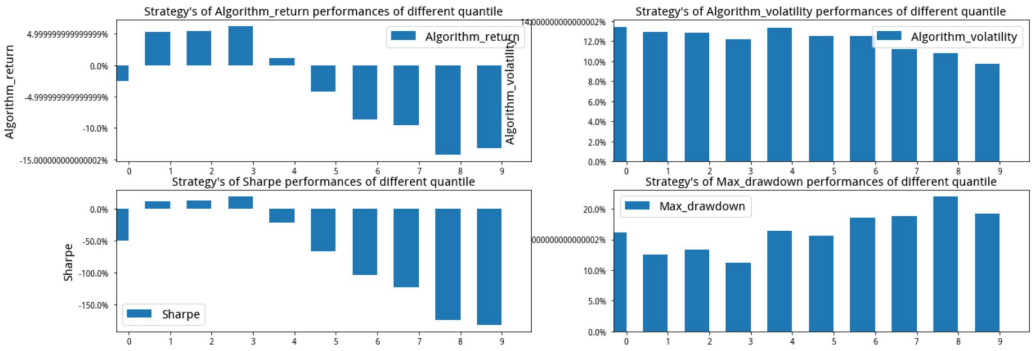
(3) Grouping back

At the end of each period, the data were divided into 10 equal parts according to the order of the factor values, and each group was backtested separately.

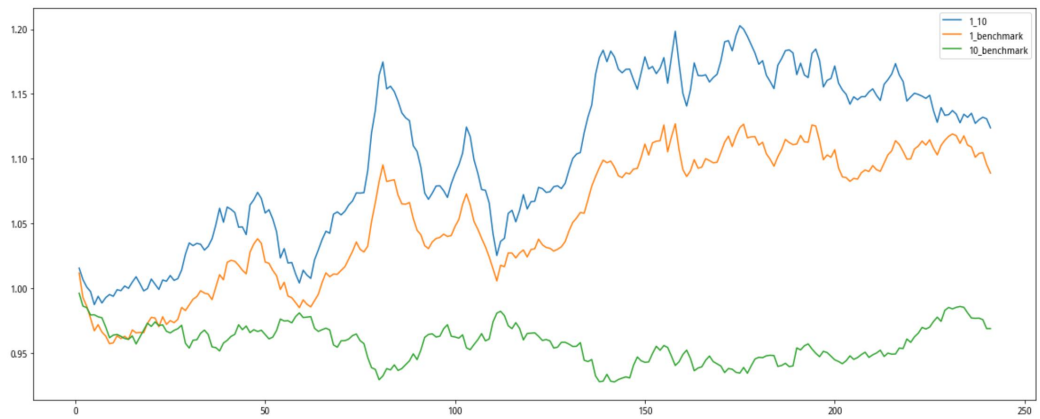
	factor	quantile	__version	algorithm_return	algorithm_volatility	alpha	annual_algo_return	annual_bm_return	avg_excess_return	avg_position_days
0	BP	(0, 10)	101	-0.025536	0.134467	0.0404928	-0.026369	-0.10697	0.000366423	174.27
1	BP	(10, 20)	101	0.0531229	0.129554	0.135871	0.0549264	-0.10697	0.000678345	131.415
2	BP	(20, 30)	101	0.0546153	0.12851	0.144611	0.0564708	-0.10697	0.000680389	105.756
3	BP	(30, 40)	101	0.061846	0.122107	0.147672	0.0639545	-0.10697	0.000707482	98.7292
4	BP	(40, 50)	101	0.0110379	0.133194	0.109669	0.0114049	-0.10697	0.000503765	93.1626
5	BP	(50, 60)	101	-0.0418053	0.124791	0.044797	-0.0431571	-0.10697	0.00028223	93.051
6	BP	(60, 70)	101	-0.0862141	0.125129	-0.00272728	-0.0889336	-0.10697	8.70619e-05	90.5816
7	BP	(70, 80)	101	-0.0952341	0.112399	-0.0254153	-0.0982225	-0.10697	4.61483e-05	94.2753
8	BP	(80, 90)	101	-0.143092	0.107965	-0.0848866	-0.147455	-0.10697	-0.000175688	106.236
9	BP	(90, 100)	101	-0.132049	0.097004	-0.0856166	-0.136103	-0.10697	-0.000121727	144.79



The original strategy was to sort the BP values from lowest to highest (positive order), and the images roughly indicate that the 10th group would have the highest BP values, with lower returns expected.



Overall the groupings with low scores (i.e., low BP) had better returns and risk-adjusted



returns, with smaller retracements

The blue curve is the net value curve of group I divided by group X. The curve shows a general upward trend, indicating that it is effective to go long in group I and short in group X

algorithmic factors	the full changeover days factor
A detailed description of the intuition behind the single-factor algorithm that	
This strategy incorporates an ROE-based stock selection methodology with a full changeover days factor 1,The strategy starts with an initial screening based on ROE.Companies with higher ROE usually have stronger profitability, and are more likely to be more profitable than those with higher ROE. 2. The strategy ranks the full turnover days factor in ascending order, i.e., stocks with a cumulative turnover rate rapidly reaching 100% are considered to be more favored by the market. Then, based on a set quartile (e.g., the top 10 percent), a subset of stocks with the smallest factor values are selected, and these are the stocks that are considered to be active in the market and have a high level of investor interest. 3. The final list of stocks is based on ROE and is further filtered by the full turnover days factor. The strategy will buy and sell within a set period, utilizing the combination of these two factors to capture potential excess returns. Risk management: control of positions, risk control signals to liquidate positions in a timely manner, and avoidance of repeat purchases of stocks already held	

Below is the code for the implemented algorithm

```
from kuanke.wizard import *
from jqdata import *
from jqfactor import get_factor_values
import numpy as np
import pandas as pd
import talib
import datetime

## Initialize functions, set stocks, benchmarks, etc., to be manipulated
def initialize(context):
set_params() # set policy parameters
# Setting the baseline
   set_benchmark('000906.XSHG')
# Setting the slippage point
   set_slippage(FixedSlippage(0.02))
# True to enable dynamic reweighting mode, using real price trading
   set_option('use_real_price', True)
# Setting volume ratios
   set_option('order_volume_ratio', 1)
# The stock class transaction fee is: buy commission 3/10,000, sell commission 3/10,000 plus 1/1,000 stamp duty, each transaction commission minimum deduction of 5 dollars, # The stock class transaction fee is: buy commission 3/10,000, sell commission 3/10,000 plus 1/1,000 stamp duty, each transaction commission minimum deduction of 5 bucks
   set_order_cost(OrderCost(open_tax=0, close_tax=0.001, open_commission=0.0003, close_commission=0.0003, min_commission=5), type='stock')
# Maximum position weight of individual stocks
   g.security_max_proportion = 1
# Frequency of stock selection
   g.check_stocks_refresh_rate = 5
# Buying frequency
   g.buy_refresh_rate = 1
# Selling frequency
```

```
    g.sell_refresh_rate = 1
# Maximum number of positions
    g.max_hold_stocknum = 20

# stock picking frequency counters
    g.check_stocks_days = 0
# Frequency counters for purchase and sale transactions
    g.buy_trade_days=0
    g.sell_trade_days=0
# Getting unsold shares
    g.open_sell_securities = []
# Sell the stock dict
    g.selled_security_list={}

# stock screening initialization function
    check_stocks_initialize()
# stock filter sort initialization function
    check_stocks_sort_initialize()
# out of the field initialization function
    sell_initialize()
# entry initialization function
    buy_initialize()
# The wind control initialization function
    risk_management_initialize()

# Close the cue
    log.set_level('order', 'info')

# Run the function
run_daily(sell_every_day,'open') #Sell shares that were not sold successfully
run_daily(risk_management, 'every_bar') #risk_control
run_daily(check_stocks, 'open') #pick_stocks
run_daily(trade, 'open') #trade
run_daily(sold_security_list_count, 'after_close') #sold_stock_date_count

def set_params():
g.factor = 'full_turnover_days' # current backtested single factor
g.factors = {'full_turnover_days': False} # False means the factors are in ascending order
    g.sort_rank = True
g.quantile = (0, 10) # Focus on stocks that are in the top 10% (0 to 10th percentile) of the sorted order

## Stock screening initialization function
def check_stocks_initialize():
# Whether or not to filter stops
    g.filter_paused = True
# Whether or not to filter delisting
    g.filter_delisted = True
# Is it only ST
    g.only_st = False
# whether to filter ST
    g.filter_st = True
# Stock pools
    g.security_universe_index = ["000300.XSHG","000905.XSHG"]
    g.security_universe_user_securities = []
# Industry listings
    g.industry_list =
["801010","801020","801030","801040","801050","801080","801110","801120","801130","801140","801150","801160","8011
```

```
70","801180","801200","801210","801230","801710","801720","801730","801740","801750","801760","801770","801780","80
1790","801880","801890"]
# List of concepts
    g.concept_list = []

## Stock filter sort initialization function
def check_stocks_sort_initialize():
# General ordering criteria: desc-desc, asc-asc
    g.check_out_lists_ascending = 'desc'

## appearing in the initialization function
def sell_initialize():
# Sets whether to sell stocks in buy_lists
    g.sell_will_buy = False

# Fix the number or percentage of positions to be exited
    g.sell_by_amount = None
    g.sell_by_percent = None

## Entry initialization function
def buy_initialize():
# Whether or not repeat purchases can be made
    g.filter_holded = False

# Delegation type
    g.order_style_str = 'by_cap_mean'
    g.order_style_value = 100

## The wind control initialization function
def risk_management_initialize():
# Strategic wind signaling
    g.risk_management_signal = True

# The strategy triggers a wind-down signal on the same day
    g.daily_risk_management = True

# Maximum number or amount of shares bought in a single
    g.max_buy_value = None
    g.max_buy_amount = None


## Selling unsold shares
def sell_every_day(context):
    g.open_sell_securities = list(set(g.open_sell_securities))
    open_sell_securities = [s for s in context.portfolio.positions.keys() if s in g.open_sell_securities]
    if len(open_sell_securities)>0:
        for stock in open_sell_securities:
            order_target_value(stock, 0)
    g.open_sell_securities = [s for s in g.open_sell_securities if s in context.portfolio.positions.keys()]
    return

## Wind control
def risk_management(context):
### _Windshield Function Screening-Start ###
### _Windshield function screening-end ###
    return

## Stock screening
```

```
def check_stocks(context):
    if g.check_stocks_days%g.check_stocks_refresh_rate != 0:
# Counter plus one
        g.check_stocks_days += 1
        return
# Stock pool assignments
    g.check_out_lists = get_security_universe(context, g.security_universe_index, g.security_universe_user_securities)
# Industry filtration
    g.check_out_lists = industry_filter(context, g.check_out_lists, g.industry_list)
# Conceptual filtering
    g.check_out_lists = concept_filter(context, g.check_out_lists, g.concept_list)
# Filtering ST Stocks
    g.check_out_lists = st_filter(context, g.check_out_lists)
# Filtering delisted stocks
    g.check_out_lists = delisted_filter(context, g.check_out_lists)
# Financial screening
    g.check_out_lists = financial_statements_filter(context, g.check_out_lists)
# Market Screening
    g.check_out_lists = situation_filter(context, g.check_out_lists)
# Technical indicators screening
    g.check_out_lists = technical_indicators_filter(context, g.check_out_lists)
# Morphological indicator screening functions
    g.check_out_lists = pattern_recognition_filter(context, g.check_out_lists)
# other screening functions
    g.check_out_lists = other_func_filter(context, g.check_out_lists)

# Sort of
    input_dict = get_check_stocks_sort_input_dict()
    g.check_out_lists = check_stocks_sort(context,g.check_out_lists,input_dict,g.check_out_lists_ascending)

# Counter normalization
    g.check_stocks_days = 1
    return

## Transaction functions
def trade(context):
# Initialize the buy list
    buy_lists = []

# Buy stock screening
    if g.buy_trade_days%g.buy_refresh_rate == 0:
# Get a list of buy_lists
        buy_lists = g.check_out_lists
# Filtering ST Stocks
        buy_lists = st_filter(context, buy_lists)
# Filtering suspended stocks
        buy_lists = paused_filter(context, buy_lists)
# Filtering delisted stocks
        buy_lists = delisted_filter(context, buy_lists)
# Filtering for stopping stocks
        buy_lists = high_limit_filter(context, buy_lists)

### _Admission Function Filter-Start ###
### _Admission Function Filter-End ###

# Selling operations
    if g.sell_trade_days%g.sell_refresh_rate != 0:
# Counter plus one
```

```
        g.sell_trade_days += 1
    else:
# Sell the stock
        sell(context, buy_lists)
# Counter normalization
        g.sell_trade_days = 1

# Buying operations
    if g.buy_trade_days%g.buy_refresh_rate != 0:
# Counter plus one
        g.buy_trade_days += 1
    else:
# Sell the stock
        buy(context, buy_lists)
# Counter normalization
        g.buy_trade_days = 1

## Sell stock date count
def selled_security_list_count(context):
    g.daily_risk_management = True
    if len(g.selled_security_list)>0:
        for stock in g.selled_security_list.keys():
            g.selled_security_list[stock] += 1

################################## Stock Pickers Group ##################################

## Financial indicator screening functions
def financial_statements_filter(context, security_list):
### _Financial Indicator Screening Functions-Start ###
    security_list = financial_data_filter_dayu(security_list, indicator.inc_revenue_year_on_year, 0)
    security_list = financial_data_filter_qujian(security_list, valuation.pe_ratio, (0,100))
    security_list = financial_data_filter_qujian(security_list, valuation.pb_ratio, (0,20))
### _Financial Indicator Screening Function-End ###

# Return to list
    return security_list

## The ticker screening function
def situation_filter(context, security_list):
### _Quote Filtering Functions-Start ###
### _Quote Filter Function - End ###

# Return to list
    return security_list

## Technical indicator screening functions
def technical_indicators_filter(context, security_list):
### _Technical Indicator Screening Functions-Start ###
### _Technical Indicator Screening Functions-End ###

# Return to list
    return security_list

## Morphological indicator screening functions
def pattern_recognition_filter(context, security_list):
### _Pattern Indicator Screening Functions - Getting Started ###
### _Pattern Indicator Screening Functions - End ###
```

```
# Return to list
    return security_list

## Other ways to filter functions
def other_func_filter(context, security_list):
### _Other ways of filtering functions-begin ###
### _Other ways of filtering functions-end ###

# Return to list
    return security_list

# Get the input_dict of the sorted stock picks
def get_check_stocks_sort_input_dict():
    input_dict = {
        indicator.roe:('desc',1),
        }
# return results
    return input_dict

################################## Trading Function Group ##################################
# Trading Functions - Exit
def sell(context, buy_lists):
# Get a list of sell_lists
    init_sl = context.portfolio.positions.keys()
    sell_lists = context.portfolio.positions.keys()

# Determine whether to sell stocks in buy_lists
    if not g.sell_will_buy:
        sell_lists = [security for security in sell_lists if security not in buy_lists]

### _Out function screening-start ###
### _Out function screening-end ###

# Sell the stock
    if len(sell_lists)>0:
        for stock in sell_lists:
            sell_by_amount_or_percent_or_none(context,stock, g.sell_by_amount, g.sell_by_percent, g.open_sell_securities)

# Get the sold stocks, and add them to g.sold_security_list
    selled_security_list_dict(context,init_sl)

    return

# Trading Functions - Entry
def buy(context, buy_lists):
# Wind control signaling
    if not g.risk_management_signal:
        return

# Determine whether a windfall liquidation stop loss has been triggered for the day
    if not g.daily_risk_management:
        return
# Determining whether a buy-in is repeatable
    buy_lists = holded_filter(context,buy_lists)

# Get the final list of buy_lists
    Num = g.max_hold_stocknum - len(context.portfolio.positions)
    buy_lists = buy_lists[:Num]
```

```
# Buying stock
    if len(buy_lists)>0:
# Allocating funds
        result = order_style(context,buy_lists,g.max_hold_stocknum, g.order_style_str, g.order_style_value)
        for stock in buy_lists:
            if len(context.portfolio.positions) < g.max_hold_stocknum:
# Access to funds
                Cash = result[stock]
# Determining the maximum weighting of individual stocks
                value = judge_security_max_proportion(context,stock,Cash,g.security_max_proportion)
# Determine the maximum number of shares or amount of money that can be bought in a single
                amount = max_buy_value_or_amount(stock,value,g.max_buy_value,g.max_buy_amount)
# Place your order
                order(stock, amount, MarketOrderStyle())
    return

################################## Public Function Group ##################################
## Sort of
def check_stocks_sort(context,security_list,input_dict,ascending='desc'):
    if (len(security_list) == 0) or (len(input_dict) == 0):
        return security_list
    else:
# Generate a list of keys
        idk = list(input_dict.keys())
# Generate the matrix
        a = pd.DataFrame()
        for i in idk:
            b = get_sort_dataframe(security_list, i, input_dict[i])
            a = pd.concat([a,b],axis = 1)
# Generate the score column
        a['score'] = a.sum(1,False)
# Sorted by score
if ascending == 'asc':# ascending
            if hasattr(a, 'sort'):
                a = a.sort(['score'],ascending = True)
            else:
                a = a.sort_values(['score'],ascending = True)
elif ascending == 'desc':# descending
            if hasattr(a, 'sort'):
                a = a.sort(['score'],ascending = False)
            else:
                a = a.sort_values(['score'],ascending = False)
# return results
        return list(a.index)

## Filtering the same underlying to not buy it again for N days after it was last sold
def filter_n_tradeday_not_buy(security, n=0):
    try:
        if (security in g.selled_security_list.keys()) and (g.selled_security_list[security]<n):
            return False
        return True
    except:
        return True

## Whether or not it's a repeatable purchase
def holded_filter(context,security_list):
    if not g.filter_holded:
```

```
        security_list = [stock for stock in security_list if stock not in context.portfolio.positions.keys()]
# return results
    return security_list

## Sell the stock and add it to the dict. ##
def selled_security_list_dict(context,security_list):
    selled_sl = [s for s in security_list if s not in context.portfolio.positions.keys()]
    if len(selled_sl)>0:
        for stock in selled_sl:
            g.selled_security_list[stock] = 0

## Filtering suspended stocks
def paused_filter(context, security_list):
    if g.filter_paused:
        current_data = get_current_data()
        security_list = [stock for stock in security_list if not current_data[stock].paused]
# return results
    return security_list

## Filtering delisted stocks
def delisted_filter(context, security_list):
    if g.filter_delisted:
        current_data = get_current_data()
security_list = [stock for stock in security_list if not (('retreat' in current_data[stock].name) or ('*' in current_data[stock].name))]
# return results
    return security_list


## Filtering ST stocks
def st_filter(context, security_list):
    if g.only_st:
        current_data = get_current_data()
        security_list = [stock for stock in security_list if current_data[stock].is_st]
    else:
        if g.filter_st:
            current_data = get_current_data()
            security_list = [stock for stock in security_list if not current_data[stock].is_st]
# return results
    return security_list

# Filtering for stopping stocks
def high_limit_filter(context, security_list):
    current_data = get_current_data()
    security_list = [stock for stock in security_list if not (current_data[stock].day_open >= current_data[stock].high_limit)]
# return results
    return security_list

# Get stock stock pools
def get_security_universe(context, security_universe_index, security_universe_user_securities):
    temp_index = []
    for s in security_universe_index:
        if s == 'all_a_securities':
            temp_index += list(get_all_securities(['stock'], context.current_dt.date()).index)
        else:
            temp_index += get_index_stocks(s)
    for x in security_universe_user_securities:
        temp_index += x
    return  sorted(list(set(temp_index)))
```

```
# Industry filtration
def industry_filter(context, security_list, industry_list):
    if len(industry_list) == 0:
# return to the stock list
        return security_list
    else:
        securities = []
        for s in industry_list:
            temp_securities = get_industry_stocks(s)
            securities += temp_securities
        security_list = [stock for stock in security_list if stock in securities]
# return to the stock list
        return security_list

# Conceptual filtering
def concept_filter(context, security_list, concept_list):
    if len(concept_list) == 0:
        return security_list
    else:
        securities = []
        for s in concept_list:
            temp_securities = get_concept_stocks(s)
            securities += temp_securities
        security_list = [stock for stock in security_list if stock in securities]
# return to the stock list
        return security_list
# Customized functions
# Customize the function that calculates the full changeover days factor
def cpt_stk_allturn_days(stk, dt):
    df_turn = get_valuation([stk], end_date=dt, fields=['turnover_ratio'], count=300)
    df_ = df_turn.sort_values(by='day', ascending=False)
    df_['no'] = range(len(df_))
    df_['cum_turn'] = df_['turnover_ratio'].cumsum()
    try:
        all_turn_days = df_.loc[df_['cum_turn'] > 100].iloc[0]['no']
    except:
        all_turn_days = np.nan
    return all_turn_days


# Polygraphic wide factor screening
def jq_factor_filter(context, security_list):
print("================================== Full Changeover Days Factor Screening Begins
==================================")
    if len(security_list) < g.max_hold_stocknum * 2:
print("The number of stocks in the pool does not meet the requirement, and the full turnover days factor is not screened.")
        return security_list

    full_turnover_days = {}
    for stock in security_list:

        full_turnover_days[stock] = cpt_stk_allturn_days(stock, context.current_dt.date())

# Filter out stocks that are calculated as nan
    filtered_stocks = {stk: days for stk, days in full_turnover_days.items() if not np.isnan(days)}

# Sorted in ascending order of days to full turnover
```

```
    sorted_stocks = [stock for stock, days in sorted(filtered_stocks.items(), key=lambda x: x[1])]

# Determine the range of stocks to pick based on g.quantile
    quantile_start = int(len(sorted_stocks) * g.quantile[0] / 100)
    quantile_end = int(len(sorted_stocks) * g.quantile[1] / 100)
    selected_stocks = sorted_stocks[quantile_start:quantile_end]

print("Number of stocks retained after full changeover days factor screening:%d" % len(selected_stocks))
    return selected_stocks
```

Condition settings and results of experiments with the full changeover days factorization algorithm

1. Sample screening

Sample range: CSI 800 Index Constituents
Test Sample Period:2019-01-01 to 2020-01-01
(1) Eliminate stocks with ST/PT on the stock selection date.
(2) Exclusion of stocks listed for less than one year.
(3) Excluding stocks that cannot be purchased on the stock selection date due to suspension of trading or other reasons.

2. Data cleansing

The winsorize shrinkage process was used to take a 95% confidence interval, and outliers were defined as outliers. For outliers, the value is changed to the boundary of the 95% confidence interval.

3. factor standardization

Rank standardized processing factor data.

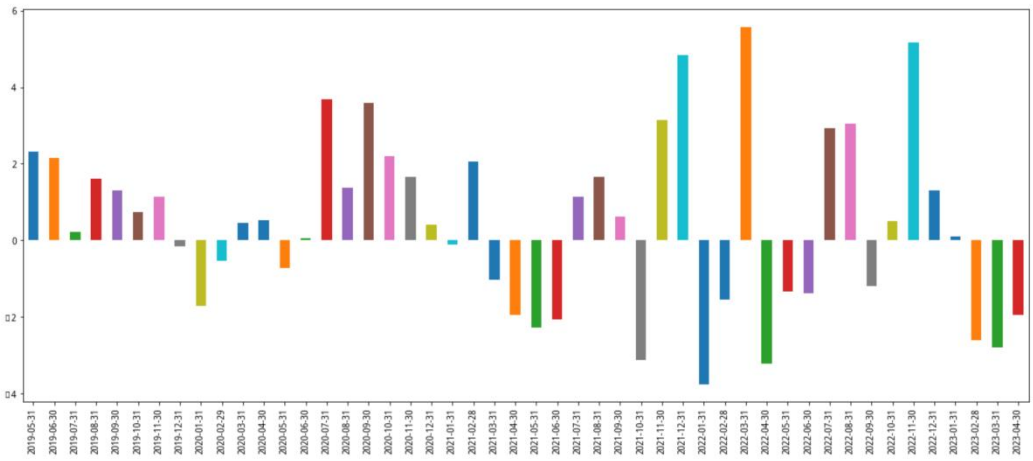
4. factor validity analysis

(1) t-test

Inspection time period:2019/05/31-2023/05/31

t值序列绝对值平均值——判断因子的显著性（即预测是否有效） 1.852532814886456
t值序列绝对值大于1.96的占比——判断因子的显著性是否稳定 0.3958333333333333

<matplotlib.axes._subplots.AxesSubplot at 0x7f7455d73908>

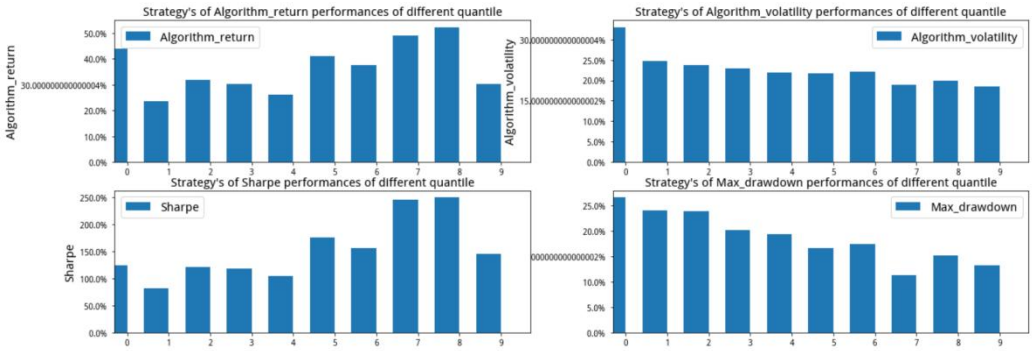


The mean absolute value of the t-value was 1.8523, which did not exceed 2, indicating that the factor was not statistically significant.

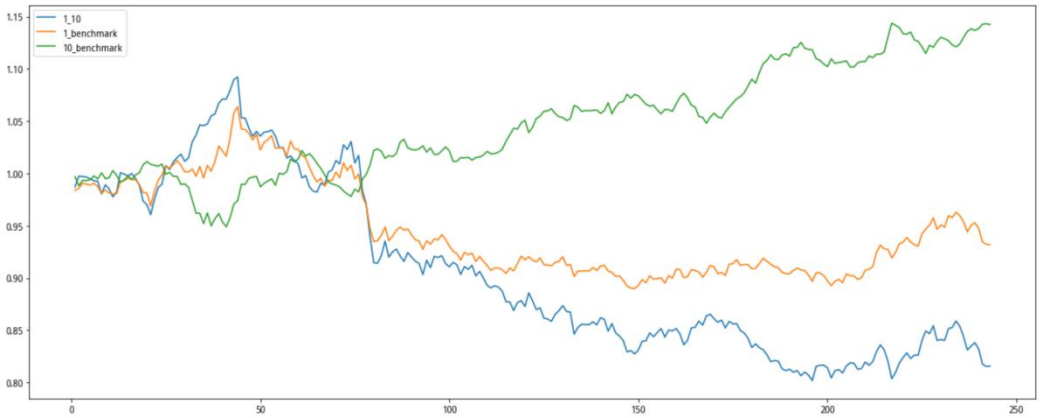
(2) factorial grouping tests

Test Sample Period:2019-01-01 to 2020-01-01

	factor	quantile	__version	algorithm_return	algorithm_volatility	alpha	annual_algo_return	annual_bm_return	avg_excess_return
0	full_turnover_days	(0, 10)	101	0.438835	0.330393	-0.0220825	0.451765	0.34665	0.000370582
1	full_turnover_days	(10, 20)	101	0.235426	0.248147	-0.142051	0.241866	0.34665	-0.000301231
2	full_turnover_days	(20, 30)	101	0.317932	0.236639	-0.0447091	0.326909	0.34665	-4.12666e-05
3	full_turnover_days	(30, 40)	101	0.302668	0.228609	-0.0514135	0.311166	0.34665	-9.16726e-05
4	full_turnover_days	(40, 50)	101	0.26233	0.219049	-0.0768392	0.269582	0.34665	-0.000220489
5	full_turnover_days	(50, 60)	101	0.41005	0.217235	0.076658	0.422015	0.34665	0.00023192
6	full_turnover_days	(60, 70)	101	0.37703	0.221656	0.0345162	0.387906	0.34665	0.000134447
7	full_turnover_days	(70, 80)	101	0.490523	0.188822	0.197147	0.505224	0.34665	0.00045617
8	full_turnover_days	(80, 90)	101	0.523633	0.199293	0.220853	0.539492	0.34665	0.00054876
9	full_turnover_days	(90, 100)	101	0.302337	0.185847	0.00838033	0.310824	0.34665	-9.63039e-05



Returns fluctuate as the number of days to full turnover increases, but are highest in the (70, 80) and (80, 90) ranges, suggesting that stocks with low turnover have better returns during these periods. Volatility is lower. Taken together, stocks with long days to full turnover show higher returns, lower volatility, and provide higher excess returns during this period.



The blue line represents a declining net worth curve for group 1 divided by group 10, the orange line represents a declining net worth curve for group 1 divided by the benchmark group, and the green line represents an increasing net worth curve for group 10 divided by the benchmark group. Over the time period examined, stocks with long full turnover days (group 10) not only outperform stocks with short full turnover days (group 1), but also outperform the market as a whole.

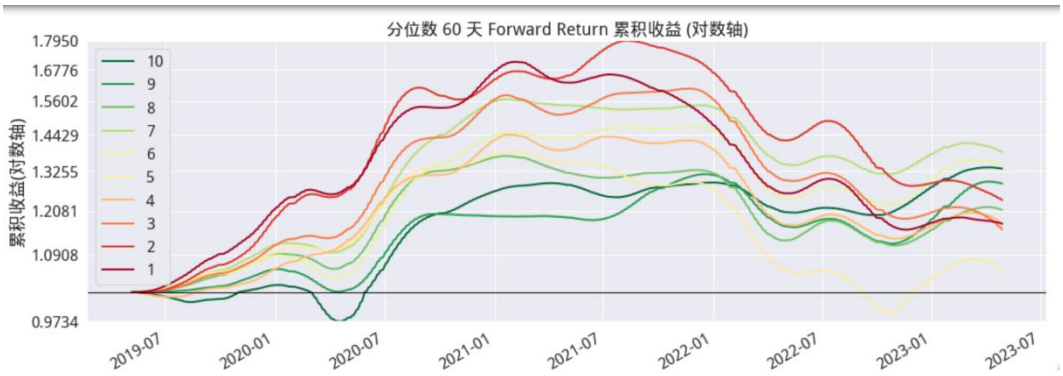
(3) single factor tests

Sample range: CSI 300 Index Constituents

Test Sample Period:2019-05-01 to 2023-05-01

Excluding ST/PT on the stock selection date, listed for less than one year, during the momentum calculation period, and suspended on the stock selection date

Factor preprocessing and neutralization, by aggregated broad level industry and market capitalization

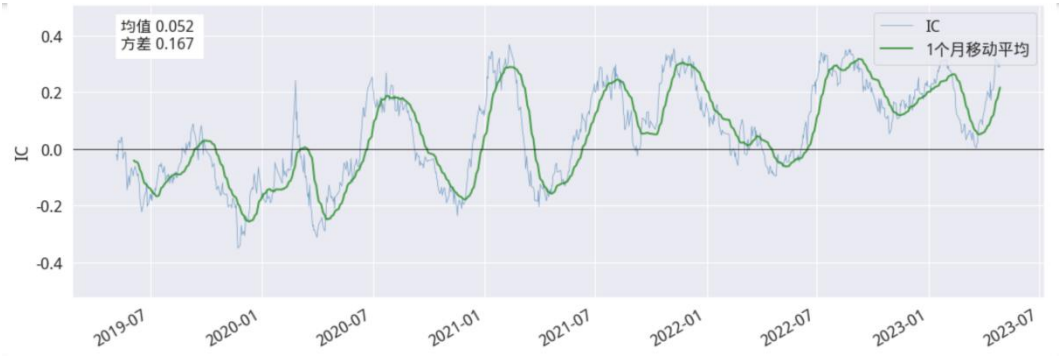


The chart shows the cumulative returns of the 60-day, 10-quartile position adjustment. During the bull market (mid-2019 to early 2020), low turnover day stocks (i.e., the low quartile group) outperformed, probably because the market was liquid and actively traded, and investors tended to chase those stocks that were frequently traded and actively performing. The market has since become entangled, with no main direction, and appears to be on the verge of reversing in 2023.

During the period from the second half of 2019 to the beginning of 2020, the performance of the portfolio was declining when going long the stocks with the highest factor scores (largest quartile) while shorting the stocks with the lowest scores (smallest quartile). This may indicate that stocks with high factor scores underperformed stocks with low scores during that period.

The performance of the left and right combinations leveled off in mid-2020, which may indicate a reduction in the performance difference between factor scores

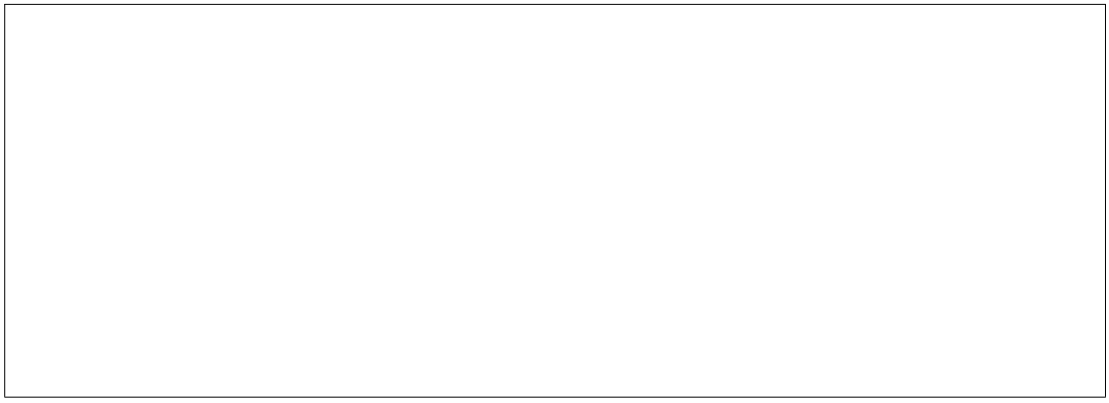
The strategy begins to show positive cumulative returns from 2021 through May 2023, indicating that during this period, stocks with high factor scores begin to regain performance and outperform those with low scores, resulting in positive returns for the hedging strategy.



Negative IC value in the first period, gradually turning to positive value in the later period, which may be a reversal factor

3. the two-factor algorithm

Algorithmic bifactorization	BP factor with an unweighted price factor
A detailed description of the intuition for the two-factor algorithm, the	
The BP factor is used to assess whether the intrinsic value of a stock is undervalued by the market. A high BP value usually indicates that a stock may be undervalued and has investment value. The DWP factor, on the other hand, directly reflects the market price movement of a stock, which captures the market's immediate evaluation of the stock and the price trend, with a lower DWP indicating that the stock is undervalued. The combination of these two factors provides a relatively comprehensive view that can help identify stocks that are both undervalued and show potential for price appreciation Execution logic. 1. Setting the basic parameters of the strategy, including the selected factor (BP and non-reweighted price), the number of days to observe the factor, the percentage of stocks in the portfolio, the benchmark index, the weights of the factor, the ranking of the stocks, and the interquartile range selected. 2. Before the opening of the market on each trading day, according to the set conditions to determine whether it is a transfer day. If it is, then determine the pool of stocks that can be traded on that day, which includes screening out non-suspended, non-ST stocks and further screening based on data from the past period. 3. For each stock in the pool, the BP value is calculated, which is usually used to measure the value of a stock, and a high BP value may indicate that the stock is undervalued. 4. The current unweighted price data is obtained directly as the value of the second factor. The uncompounded price is usually used to observe the historical changes in stock prices without taking into account the effects of stock dividends, stock bonuses and other factors. 5. Factor standardization and scoring: The two factor values are standardized, and then a composite score is synthesized based on the set weights, and this score is used to rank the stocks. 6. Stocks to be bought are selected based on the stock's overall score and a set interquartile range. On each trading day, the system automatically adjusts the position by buying stocks with high scores and selling stocks that are not in the buy list. 7. During the trading day, the actual buying and selling operations are carried out on the basis of the previously prepared and generated trading signals.	



Below is the code for the implemented algorithm

```
import datetime
import numpy as np
import pandas as pd
import time
from jqdata import *
from pandas import Series, DataFrame
import statsmodels.api as sm
from jqfactor import get_factor_values

'''
================================================================================
Before the overall backtesting, the
================================================================================
'''

#Things to do before overall backtesting
def initialize(context):
set_params() #1 sets the curated parameters
set_variables() #2 sets the intermediate variables, the
set_backtest() #3 set the backtest condition that

#1 Set the policy parameters
def set_params():
g.factor1 = 'BP' # the first factor
g.factor2 = 'price_no_fq' # second factor, non-reweighting factor
g.shift1 = 21 # Number of days of observation for the BP factor
# g.shift2 = 21 # the number of days of observation of other factors, here do not reweight the factor using real-time updated data, do not set the window period
g.percent = 0.10 # Positions as a percentage of the optional stock pool, the
g.index = '000906.XSHG' # define stock pool, CSI 800

g.factor1_weight = 0.5 # BP factor weight, default value
g.factor2_weight = 0.5 # weight of uncompounded factor, default value

    g.sort_rank = True
g.quantile = (0, 10) # Focus on stocks that are in the top 10% (0 to 10th percentile) of the sorted order

#2 Setting of intermediate variables
def set_variables():
g.feasible_stocks = [] # the current pool of tradable stocks
g.if_trade = False # Whether or not the day is traded
g.num_stocks = 0 # Set the number of stocks to hold
```

```
#3 Setting up backtesting conditions
def set_backtest():
set_benchmark('000906.XSHG') # set as benchmark
set_option('use_real_price', True) # Trade at the real price
log.set_level('order', 'error') # Set the error reporting level of the

'''
================================================================================
Before the opening of the market each day
================================================================================
'''
#Things to do every day before the opening bell
def before_trading_start(context):
# Get the current date, #
    day = context.current_dt.day
    yesterday = context.previous_date
    rebalance_day = shift_trading_day(yesterday, 1)
    if yesterday.month != rebalance_day.month:
        if yesterday.day > rebalance_day.day:
            g.if_trade = True
#5 Set feasible stock pools: get the current open stock pool and exclude stocks that are currently or during the calculation sample
period suspended from trading
            g.feasible_stocks = set_feasible_stocks(get_index_stocks(g.index), g.shift1,context)
            #g.feasible_stocks = set_feasible_stocks(get_index_stocks(g.index), max(g.shift1, g.shift2), context)
                    #6 Setting of slippage and handling charges
            set_slip_fee(context)
# Purchase stocks in proportion to the pool of viable stocks
            g.num_stocks = int(len(g.feasible_stocks) * g.percent)

#4

def shift_trading_day(date,shift):
# Get all trading days, return a list of all trading days with datetime.date type.
    tradingday = get_all_trade_days()
# Get the line number in the list for the day after date shift day Returns a number
    shiftday_index = list(tradingday).index(date)+shift
# Returns the date of the day based on the line number as a datetime.date type
    return tradingday[shiftday_index]

#5
# Setting up a pool of viable stocks

def set_feasible_stocks(stock_list,days,context):
# dataframe with information about whether the license is suspended or not, 1 for suspended, 0 for not suspended.
    suspened_info_df = get_price(list(stock_list),
                    start_date=context.current_dt,
                    end_date=context.current_dt,
                    frequency='daily',
                    fields='paused'
    )['paused'].T
# Filter for suspended stocks return dataframe
    unsuspened_index = suspened_info_df.iloc[:,0]<1
# Get the codes of the stocks that are not suspended for the day list.
    unsuspened_stocks = suspened_info_df[unsuspened_index].index
# Further, screen the list of stocks that have not been suspended in the last few days.
    feasible_stocks = []
    current_data = get_current_data()
```

```
    for stock in unsuspened_stocks:
        if sum(attribute_history(stock,
                                 days,
                                 unit = '1d',
                                 fields = ('paused'),
                                 skip_paused = False
                                )
               )[0] == 0:
            feasible_stocks.append(stock)
# Excluding ST shares
    st_data = get_extras('is_st', feasible_stocks, end_date = context.previous_date, count = 1)
    stockList = [stock for stock in feasible_stocks if not st_data[stock][0]]
    return stockList

#6 Slippage and handling fees based on different time periods
def set_slip_fee(context):
# Set the slippage point to 0
    set_slippage(FixedSlippage(0))
# Setting handling fees based on different time periods
    dt=context.current_dt

    if dt>datetime.datetime(2013,1, 1):
        set_commission(PerTrade(buy_cost=0.0003,
                                sell_cost=0.0013,
                                min_cost=5))

    elif dt>datetime.datetime(2011,1, 1):
        set_commission(PerTrade(buy_cost=0.001,
                                sell_cost=0.002,
                                min_cost=5))

    elif dt>datetime.datetime(2009,1, 1):
        set_commission(PerTrade(buy_cost=0.002,
                                sell_cost=0.003,
                                min_cost=5))

    else:
        set_commission(PerTrade(buy_cost=0.003,
                                sell_cost=0.004,
                                min_cost=5))
'''
==========================================================================
At the time of each day's trading, the
==========================================================================
'''
def handle_data(context, data):
# If it's a trading day
    if g.if_trade == True:
# 7 Get buy and sell signals, input context, output stock list list
# The corresponding default value in the dictionary is false holding_list is filtered to true, then the factor with the highest score is
selected
        holding_list = get_stocks(g.feasible_stocks, context, asc=g.sort_rank)
# new additions, calculating holding_list length, # new additions, calculating holding_list length
        total_number = len(holding_list)
        # print 'feasible_stocks is %d, holding is %d' % (len(g.feasible_stocks), total_number)
# Extract the required quantile information
        (start_q, end_q) = g.quantile
# 8 Rebalance position, enter context, use signal result holding_list
```

```
        rebalance(context, holding_list, start_q, end_q, total_number)
g.if_trade = False # Assignment


#7 Raw Data Re-Extraction Factor Scoring Ranking

def get_stocks(stocks_list, context, asc):
# Obtain BP factor values and preprocess them
    df_bp = get_df_BP(stocks_list, context, g.shift1)

# Perform depolarization
    #df_BP['BP'] = winsorize(df_BP, 'BP', std=3, have_negative=True)
# Normalize the BP factors
    #df_bp['BP'] = standardize(df_bp, 'BP', ty=2)['BP']

# Use get_factor_values to get unweighted price factor data
# Direct access to current unweighted price factor data
    end_date = context.current_dt.strftime('%Y-%m-%d')
    factor_data = get_factor_values(securities=stocks_list, factors=[g.factor2], start_date=end_date, end_date=end_date)

# Directly using current unweighted price data as factor values
# Assuming that the DataFrame contained in factor_data is sorted by date, the most recent data is used here
    df_price_no_fq = factor_data[g.factor2].iloc[-1]

# Perform depolarization
    #df_price_no_fq = winsorize(df_price_no_fq, 'price_no_fq', std=3, have_negative=True)

# Standardized
    #df_price_no_fq = standardize(df_price_no_fq.to_frame('price_no_fq'), 'price_no_fq', ty=2)['price_no_fq']

# Ensure that df_bp contains BP values
    df_bp = df_bp[df_bp['BP'].notnull()]

# Combine two factors for a weighted score
    df_combined = pd.DataFrame(index=df_bp.index)
    df_combined['score'] = g.factor1_weight * df_bp['BP'] + g.factor2_weight * df_price_no_fq

# Ranked on the basis of a composite score
    df_combined = df_combined.sort_values(by='score', ascending=asc)
    return list(df_combined.index)
#8
# Get a list of stocks to buy based on the strategy's buy signals

def rebalance(context, holding_list, start_q, end_q, total_number):
    if end_q == 100:
        end_q = 100
# Amount purchased per stock
    every_stock = context.portfolio.portfolio_value/g.num_stocks
# Empty positions only buy operations
    if len(list(context.portfolio.positions.keys()))==0:
# The original reset scort starts with a return-related scoring calculation, where returns are sorted in ascending order
        # for stock_to_buy in holding_list[int(round(start_q * total_number / 100)) : int(round(end_q * total_number / 100))]:
        for stock_to_buy in holding_list[int(round(start_q * total_number / 100)) : int(round(end_q * total_number / 100))]:
            order_target_value(stock_to_buy, every_stock)
    else :
# Not a short position to first sell shares held but not on the buy list
        for stock_to_sell in list(context.portfolio.positions.keys()):
            if stock_to_sell not in holding_list[int(round(start_q * total_number / 100)) : int(round(end_q * total_number / 100))]:
```

```
            order_target_value(stock_to_sell, 0)
# Because the order function is adjusted to the order of adjustment, in order to prevent the first line to adjust the position of the stock due to the latter line to adjust the position of the stock accounted for the amount of too large can not be adjusted in place at once, here to run twice to solve this problem
        for stock_to_buy in holding_list[int(round(start_q * total_number / 100)) : int(round(end_q * total_number / 100))]:
            order_target_value(stock_to_buy, every_stock)
        for stock_to_buy in holding_list[int(round(start_q * total_number / 100)) : int(round(end_q * total_number / 100))]:
            order_target_value(stock_to_buy, every_stock)

# BP
# Get a dataframe: containing the ticker symbol, book-to-market ratio BP and corresponding rank BP_sorted_rank
# Default date = the day before context.current_dt
def get_df_BP(stock_list, context, asc):
    df_BP = get_fundamentals(query(valuation.code, valuation.pb_ratio
                    ).filter(valuation.code.in_(stock_list)))
# Get the pb countdown, #
    df_BP['BP'] = df_BP['pb_ratio'].apply(lambda x: 1/x)



# Delete nan in case an item in the data does not generate nan
    df_BP = df_BP[pd.notnull(df_BP['BP'])]
# Generate ranking ordinal numbers
    df_BP['BP_sorted_rank'] = df_BP['BP'].rank(ascending = asc, method = 'dense')
# Use the ticker symbol as the index
    df_BP.index = df_BP.code
# Delete useless data
    del df_BP['code']
    print (df_BP)
    return df_BP



# Depolarization function (3x standard deviation depolarization)
def winsorize(factor_data, factor, std=3,  have_negative = True):
    '''
The depolarization function , the
factor: Series with stock code as index and factor value as value
std is the number of times the standard deviation, have_negative is a boolean, whether or not to include negative values
Output Series
    '''
    r=factor_data[factor]
    if have_negative == False:
        r = r[r>=0]
    else:
        pass
# Taking extreme values
    edge_up = r.mean()+std*r.std()
    edge_low = r.mean()-std*r.std()
    r[r>edge_up] = edge_up
    r[r<edge_low] = edge_low
    r = pd.DataFrame(r)
    return r


# z-score standardized function.
def standardize(factor_data,factor,ty=2):
```

```
temp=factor_data[factor]
if int(ty)==1:
    re = (temp - temp.min())/(temp.max() - temp.min())
elif ty==2:
    re = (temp - temp.mean())/temp.std()
elif ty==3:
    re = temp/10**np.ceil(np.log10(temp.abs().max()))
return pd.DataFrame(re)
```

conditioning and results of algorithmic experiments

1. Sample screening

Sample range: CSI 800 Index Constituents

Test Sample Period:2023-01-01 to 2024-01-01

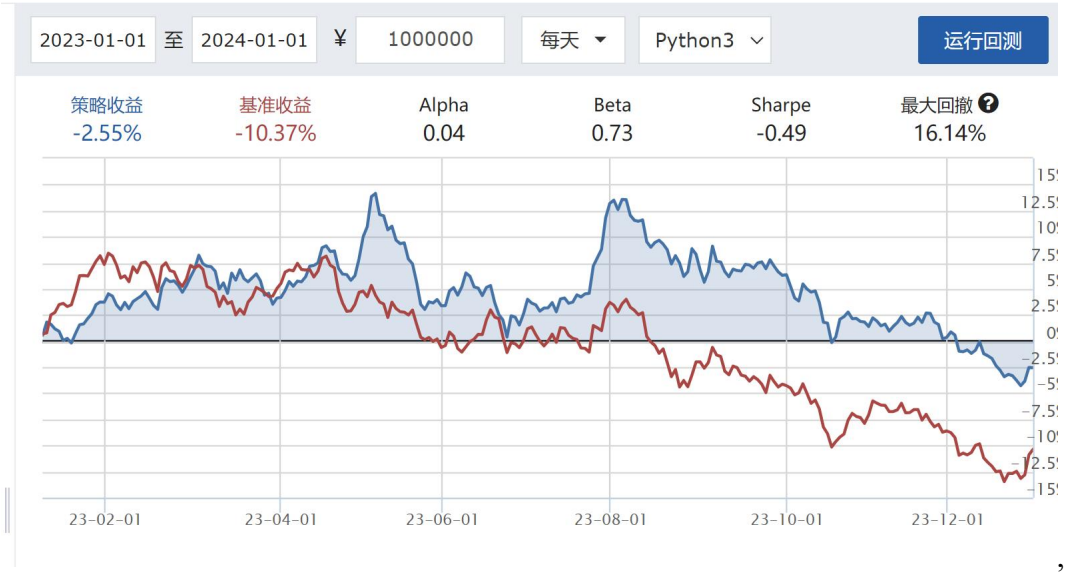
(1) Eliminate stocks with ST/PT on the stock selection date.

(2) Exclusion of stocks listed for less than one year.

(3) Excluding stocks that cannot be purchased on the stock selection date due to suspension of trading or other reasons.

2. one-factor vs. two-factor

2.1 Backtesting data using a single factor (BP)



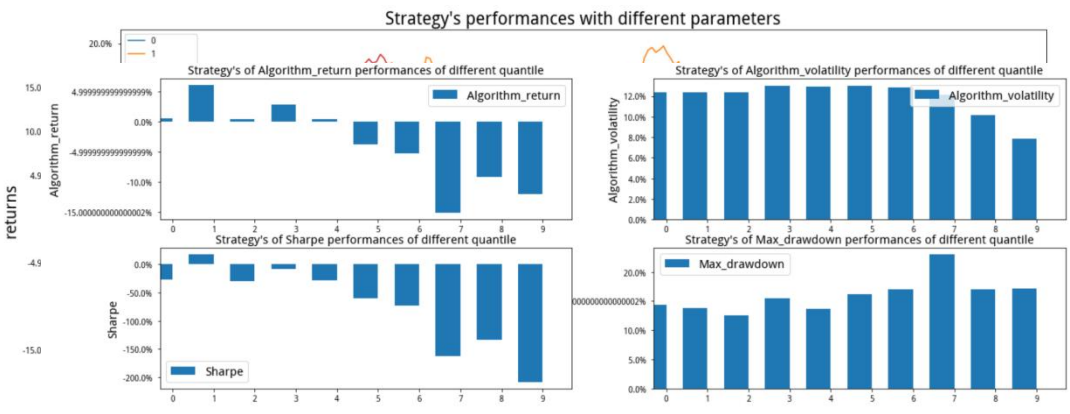
, the

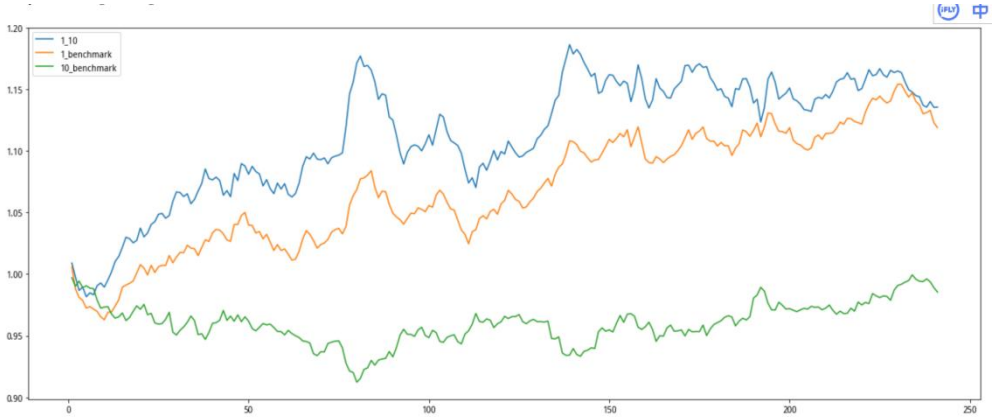
2.2 Using two-factor (BP with unweighted factor) backtesting data, the

The two-factor strategy with the addition of the non-complex-weighted price factor outperforms the single BP-factor strategy in terms of return, risk-adjusted return, and maximum retracement, which suggests that the overall performance and risk tolerance of the strategy can be improved by combining the two factors in this case.

3. Two-factor strategy group backtesting

	quantile	__version	algorithm_return	algorithm_volatility	alpha	annual_algo_return	annual_bm_return	avg_excess_return	avg_position_days	a
0	(0, 10)	101	0.00553701	0.123574	0.0754845	0.00572058	-0.10697	0.000489252	176.833	
1	(10, 20)	101	0.0600489	0.123953	0.139703	0.0620944	-0.10697	0.000704156	146.769	
2	(20, 30)	101	0.00315704	0.123935	0.0816669	0.00326157	-0.10697	0.000475705	124.258	
3	(30, 40)	101	0.028299	0.130275	0.122137	0.029248	-0.10697	0.00057464	115.41	
4	(40, 50)	101	0.00313396	0.12895	0.0982949	0.00323773	-0.10697	0.000470559	107.888	
5	(50, 60)	101	-0.0376755	0.130288	0.0569281	-0.0388965	-0.10697	0.00029921	107.676	
6	(60, 70)	101	-0.0523507	0.128215	0.0366482	-0.0540337	-0.10697	0.000236965	104.489	
7	(70, 80)	101	-0.151916	0.121251	-0.0712652	-0.156524	-0.10697	-0.000222715	115.693	
8	(80, 90)	101	-0.092443	0.101094	-0.0353281	-0.0953485	-0.10697	5.99052e-05	127.496	
9	(90, 100)	101	-0.120001	0.0786436	-0.0893069	-0.123712	-0.10697	-6.38208e-05	145.939	





The upward trend of the curve indicates that it is valid to go long in the first group (with high composite scores, i.e., high BP values and low unweighted prices) and short in the last group (with low composite scores, i.e., low BP values and high unweighted prices)

4. Two-factor algorithm in-depth research and analysis that

(1) The first dimension: changing the scoring/grouping method of

i. The shift from scoring in the order of factor values to scoring by volume

Z-score Standardized volume-based scoring still uses tertile grouping (10th percentile)

```
import datetime

import numpy as np
import pandas as pd
import time
from jqdata import *
from pandas import Series, DataFrame
import statsmodels.api as sm
from jqfactor import get_factor_values
from sklearn.cluster import KMeans


'''
================================================================================
Before the overall backtesting, the
================================================================================
'''

#Things to do before overall backtesting
def initialize(context):
```

```
set_params() #1 sets the curated parameters
set_variables() #2 sets the intermediate variables, the
set_backtest() #3 set the backtest condition that

#1 Set the policy parameters
def set_params():
g.factor1 = 'BP' # the first factor
g.factor2 = 'price_no_fq' # second factor, non-reweighting factor
g.shift1 = 21 # Number of days of observation for the BP factor
# g.shift2 = 21 # the number of days of observation of other factors, here do not reweight the factor using real-time updated data, do not set the window period
g.percent = 0.10 # Positions as a percentage of the optional stock pool, the
g.index = '000906.XSHG' # define stock pool, CSI 800

g.factor1_weight = 0.5 # BP factor weight, default value
g.factor2_weight =0.5 # weight of uncompounded factor, default value

    #g.sort_rank = True
g.quantile = (0, 10) # Focus on stocks that are in the top 10% (0 to 10th percentile) of the sorted order
#g.k_clusters = 10 # Add: number of clusters, for adaptive grouping
#g.cluster_method = 'kmeans' # New: clustering method
#g.target_group_index = 0 # Specify the index of the group you want to backtest, starting with 0 (0 is the highest scoring group).

#2 Setting of intermediate variables
def set_variables():
g.feasible_stocks = [] # the current pool of tradable stocks
g.if_trade = False # Whether or not the day is traded
g.num_stocks = 0 # Set the number of stocks to hold

#3 Setting up backtesting conditions
def set_backtest():
set_benchmark('000906.XSHG') # set as benchmark
set_option('use_real_price', True) # Trade at the real price
log.set_level('order', 'error') # Set the error reporting level of the

'''
================================================================================
Before the opening of the market each day
================================================================================
'''
#Things to do every day before the opening bell
def before_trading_start(context):
# Get the current date, #
    day = context.current_dt.day
    yesterday = context.previous_date
    rebalance_day = shift_trading_day(yesterday, 1)
    if yesterday.month != rebalance_day.month:
        if yesterday.day > rebalance_day.day:
            g.if_trade = True
#5 Set feasible stock pools: get the current open stock pool and exclude stocks that are currently or during the calculation sample period suspended from trading
            g.feasible_stocks = set_feasible_stocks(get_index_stocks(g.index), g.shift1,context)
            #g.feasible_stocks = set_feasible_stocks(get_index_stocks(g.index), max(g.shift1, g.shift2), context)
                #6 Setting of slippage and handling charges
            set_slip_fee(context)
# Purchase stocks in proportion to the pool of viable stocks
            g.num_stocks = int(len(g.feasible_stocks) * g.percent)
```

```
#4

def shift_trading_day(date,shift):
# Get all trading days, return a list of all trading days with datetime.date type.
    tradingday = get_all_trade_days()
# Get the line number in the list for the day after date shift day Returns a number
    shiftday_index = list(tradingday).index(date)+shift
# Returns the date of the day based on the line number as a datetime.date type
    return tradingday[shiftday_index]

#5
# Setting up a pool of viable stocks

def set_feasible_stocks(stock_list,days,context):
# dataframe with information about whether the license is suspended or not, 1 for suspended, 0 for not suspended.
    suspened_info_df = get_price(list(stock_list),
                    start_date=context.current_dt,
                    end_date=context.current_dt,
                    frequency='daily',
                    fields='paused'
    )['paused'].T
# Filter for suspended stocks return dataframe
    unsuspened_index = suspened_info_df.iloc[:,0]<1
# Get the codes of the stocks that are not suspended for the day list.
    unsuspened_stocks = suspened_info_df[unsuspened_index].index
# Further, screen the list of stocks that have not been suspended in the last few days.
    feasible_stocks = []
    current_data = get_current_data()
    for stock in unsuspened_stocks:
        if sum(attribute_history(stock,
                            days,
                            unit = '1d',
                            fields = ('paused'),
                            skip_paused = False
                           )
            )[0] == 0:
            feasible_stocks.append(stock)
# Excluding ST shares
    st_data = get_extras('is_st', feasible_stocks, end_date = context.previous_date, count = 1)
    stockList = [stock for stock in feasible_stocks if not st_data[stock][0]]
    return stockList

#6 Slippage and handling fees based on different time periods
def set_slip_fee(context):
# Set the slippage point to 0
    set_slippage(FixedSlippage(0))
# Setting handling fees based on different time periods
    dt=context.current_dt

    if dt>datetime.datetime(2013,1, 1):
        set_commission(PerTrade(buy_cost=0.0003,
                            sell_cost=0.0013,
                            min_cost=5))

    elif dt>datetime.datetime(2011,1, 1):
        set_commission(PerTrade(buy_cost=0.001,
                            sell_cost=0.002,
                            min_cost=5))
```

```
    elif dt>datetime.datetime(2009,1, 1):
        set_commission(PerTrade(buy_cost=0.002,
                                sell_cost=0.003,
                                min_cost=5))

    else:
        set_commission(PerTrade(buy_cost=0.003,
                                sell_cost=0.004,
                                min_cost=5))
'''
================================================================================
At the time of each day's trading, the
================================================================================
'''
def handle_data(context, data):
    if g.if_trade:
# Get buy and sell signals, input context, output stock list list
        holding_list = get_stocks(g.feasible_stocks, context)
# Rebalance positions, enter context, use signal result holding_list
        rebalance(context, holding_list)
        g.if_trade = False

#7 Raw Data Re-Extraction Factor Scoring Ranking

#7 Raw Data Re-Extraction Factor Scoring Ranking

def get_stocks(stocks_list, context):
# Obtain BP factor values and preprocess them
    df_bp = get_df_BP(stocks_list, context, g.shift1)


# Direct access to current unweighted price factor data
    end_date = context.current_dt.strftime('%Y-%m-%d')
    factor_data = get_factor_values(securities=stocks_list, factors=[g.factor2], start_date=end_date, end_date=end_date)

# Directly using current unweighted price data as factor values
# Assuming that the DataFrame contained in factor_data is sorted by date, the most recent data is used here
    df_price_no_fq = factor_data[g.factor2].iloc[-1]

# And then depolarization

# Ensure that df_bp contains BP values
    df_bp = df_bp[df_bp['BP'].notnull()

# Combine the two factors for weighted scores, with z-score standardization first
    df_combined = pd.DataFrame(index=df_bp.index)
    df_combined['BP_zscore'] = (df_bp['BP'] - df_bp['BP'].mean()) / df_bp['BP'].std()
    df_combined['price_no_fq_zscore'] = (df_price_no_fq - df_price_no_fq.mean()) / df_price_no_fq.std()

# In order for both factors to "score as high as possible", the z-score for the non-reweighted price factor is taken to be negative
(taking into account that low-priced stocks should receive high scores).
# Here the scores of the unweighted price factors are inverted so that low prices get high scores
    df_combined['score'] = g.factor1_weight * df_combined['BP_zscore'] - g.factor2_weight * df_combined['price_no_fq_zscore']

# Sorting stocks based on scores
    df_combined['score_rank'] = df_combined['score'].rank(ascending=False)
    total_stocks = len(df_combined)
```

```
# Calculate the number of stocks corresponding to the quartile
    start_q, end_q = g.quantile
    quantile_start = int(total_stocks * start_q / 100)
    quantile_end = int(total_stocks * end_q / 100)

# Selection of stocks based on quartiles
    selected_stocks = df_combined[(df_combined['score_rank'] > quantile_start) & (df_combined['score_rank'] <=
quantile_end)].index.tolist()

    return selected_stocks
#8

def rebalance(context, holding_list):
# Amount purchased per stock
    every_stock_value = context.portfolio.portfolio_value / len(holding_list)

# Sell stocks that are not in the holding_list first
    for stock in context.portfolio.positions:
        if stock not in holding_list:
            order_target_value(stock, 0)

# Buy or adjust stock positions in holding_list
    for stock in holding_list:
        order_target_value(stock, every_stock_value)

# BP
# Get a dataframe: containing the ticker symbol, book-to-market ratio BP and corresponding rank BP_sorted_rank
# Default date = the day before context.current_dt
def get_df_BP(stock_list, context, asc):

    df_BP = get_fundamentals(query(valuation.code, valuation.pb_ratio
                    ).filter(valuation.code.in_(stock_list)))
# Get the pb countdown, #
    df_BP['BP'] = df_BP['pb_ratio'].apply(lambda x: 1/x)

# Delete nan in case an item in the data does not generate nan
    df_BP = df_BP[pd.notnull(df_BP['BP'])]
# Generate ranking ordinal numbers
    #df_BP['BP_sorted_rank'] = df_BP['BP'].rank(ascending = asc, method = 'dense')
    df_BP['BP_sorted_rank'] = df_BP['BP'].rank(ascending=False, method='dense')
# Use the ticker symbol as the index
    df_BP.index = df_BP.code
# Delete useless data
    del df_BP['code']
    print (df_BP)
    return df_BP
```

The results of the backtesting data were not as good as the ranked scoring after the change to volume scoring

1.1 Explore further causes.

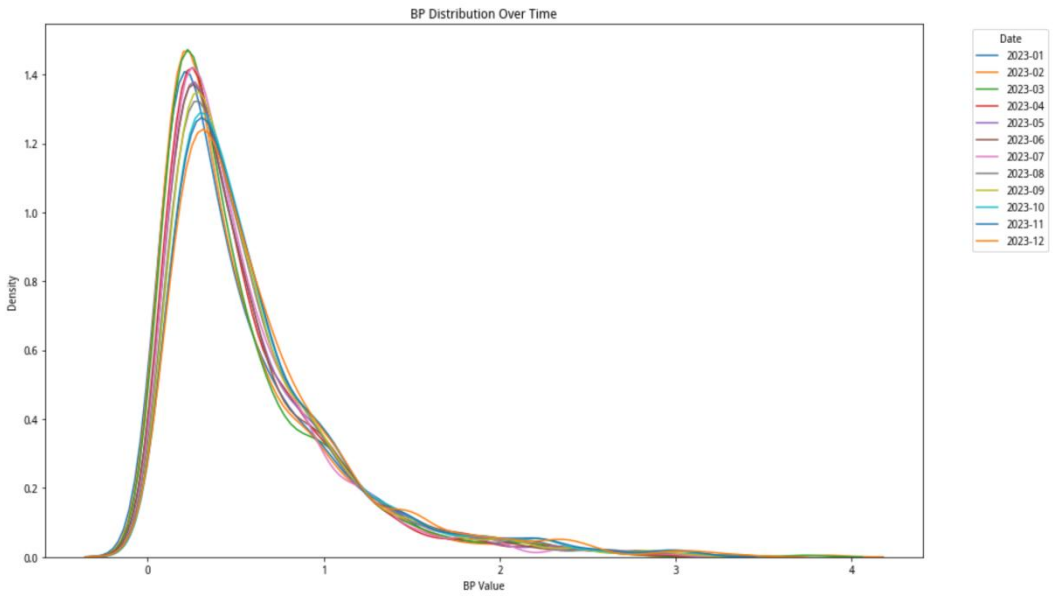
Possible Cause I: Factor Distribution and Extreme Value Treatment

Sorting and scoring: The main focus is on the relative position of the factor values rather than the absolute values. Even if the factor values of two stocks are very different, they may still be ranked next to each other, resulting in close scores.

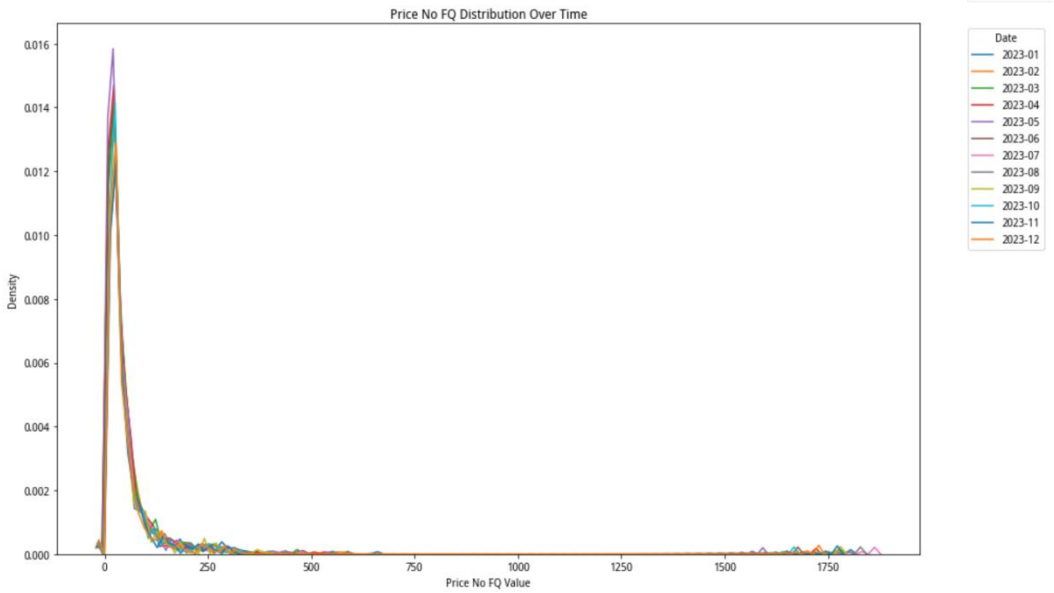
Z-score Standardized: Scoring is done by calculating the number of standard deviations of each stock's factor value from the mean, which better reflects the absolute differences in factor values. However, it is sensitive to extreme values

Visualize the distribution of factor values (sample data used here is from January 1, 2023 - January 1, 2024)

BP Factor Distribution: The distribution of BP factor values is characterized by a relatively constant



Unweighted Price Factor Distribution: Stock prices have a heavy-tailed distribution, and extreme price values may have a large impact on the overall score.



1.2 Improvement initiatives

1.2.1 Limiting Extremes by Percentiles

def winsorize_series(series, limits=[0.35, 0.65]):

The following results are obtained for depolarization at different percentiles for only the ends of the uncompounded weighting factor: the

limits=[0.30, 0.70]

limits=[0.35, 0.65], this percentile goes down to the extremes best, but there may be a risk of overfitting, the

limits=[0.4, 0.6]

1.2.2 Asymmetric depolarization methods

By observing the distribution of factor values during the sample period, it was found that there were more extreme values at the right end of the two factors, so the right-end depolarization winsorize_series_right_tail(series, upper_limit=0.35) was used.

upper_limit=0.4

upper_limit=0.35 The unweighted price factor is optimal when it retains about 35% of the data at the left end, the

upper_limit=0.3

Right-end depolarization of the BP factor based on taking 0.3 for the right-end depolarization of the uncompounded weighting factor.

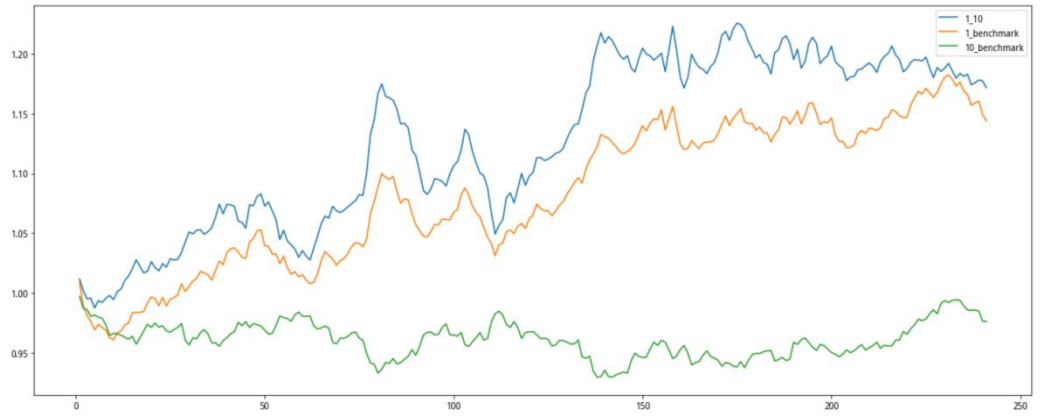
upper_limit=0.7

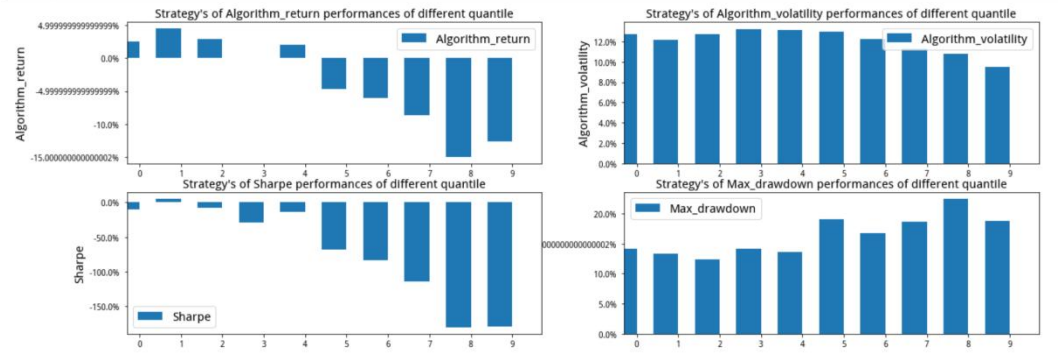
upper_limit=0.75 The BP factor retains approximately 75% of the data at the left end of the BP factor based on a depolarization of the right end of the uncompounded weighting factor to 0.3

upper_limit=0.8

Results of group backtesting after changing the scoring method.

	quantile	__version	algorithm_return	algorithm_volatility	alpha	annual_algo_return	annual_bm_return	avg_excess_return	avg_position_days
0	(0, 10)	101	0.0255351	0.126927	0.0926546	0.0263902	-0.10697	0.000573935	168.035
1	(10, 20)	101	0.0451459	0.121715	0.117314	0.0466726	-0.10697	0.000647692	134.752
2	(20, 30)	101	0.0290082	0.126677	0.115267	0.0299814	-0.10697	0.000579125	123.627
3	(30, 40)	101	0.000698307	0.132187	0.097064	0.000721399	-0.10697	0.000461585	111.783
4	(40, 50)	101	0.0208582	0.130927	0.11741	0.0215551	-0.10697	0.00054363	101.697
5	(50, 60)	101	-0.0473919	0.129535	0.0441098	-0.0489197	-0.10697	0.000258143	93.7107
6	(60, 70)	101	-0.0603086	0.121925	0.0211353	-0.062239	-0.10697	0.000202322	92.6754
7	(70, 80)	101	-0.0865203	0.112866	-0.0163995	-0.089249	-0.10697	8.5942e-05	91.875
8	(80, 90)	101	-0.150655	0.108125	-0.0936967	-0.155227	-0.10697	-0.000211765	102.569
9	(90, 100)	101	-0.126114	0.094965	-0.081824	-0.13	-0.10697	-9.32956e-05	142.753





Changing the way scoring is done Conclusion.

- The refinement of the scoring methodology resulted in a more even performance of returns across multiple quartiles, reducing the impact of extreme values.
- Optimized volatility is reduced, especially at the higher quartiles, showing better management of risk.
- The increase in the Sharpe ratio suggests that a more refined approach to scoring risk-adjusted returns may be more effective.
- The improved scoring method showed smaller maximum retracements at most points, indicating improved risk control.

A comprehensive comparison of the two scoring methods shows that volumetric scoring (the second scoring method) is more robust in terms of Sharpe ratio and maximum retracement, and although it does not show a significant increase in returns, risk control is improved. The sequential scoring method shows higher returns in the first quartile, but its volatility and maximum retracement indicators are higher, indicating higher risk exposure.

2. Changes in grouping based on quantitative scoring

Z-score standardized quantitative scoring followed by a clustering algorithm that dynamically adjusts the grouping thresholds


```
# Bifactor-BP with no reweighting factor clustering grouping 1
import datetime
import numpy as np
import pandas as pd
import time
from jqdata import *
from pandas import Series, DataFrame
import statsmodels.api as sm
from jqfactor import get_factor_values
```

```
from sklearn.cluster import KMeans


'''
================================================================================
Before the overall backtesting, the
================================================================================
'''

#Things to do before overall backtesting
def initialize(context):
set_params() #1 sets the curated parameters
set_variables() #2 sets the intermediate variables, the
set_backtest() #3 set the backtest condition that

#1 Set the policy parameters
def set_params():
g.factor1 = 'BP' # the first factor
g.factor2 = 'price_no_fq' # second factor, non-reweighting factor
g.shift1 = 21 # Number of days of observation for the BP factor
# g.shift2 = 21 # the number of days of observation of other factors, here do not reweight the factor using real-time updated data,
do not set the window period
g.percent = 0.10 # Positions as a percentage of the optional stock pool, the
g.index = '000906.XSHG' # define stock pool, CSI 800

g.factor1_weight = 0.5 # BP factor weight, default value
g.factor2_weight =0.5 # weight of uncompounded factor, default value

    #g.sort_rank = True
#g.quantile = (0, 10) # Focus on stocks that are in the top 10% (0 to 10th percentile) of the sorted list
g.k_clusters = 10 # add: number of clusters, for adaptive grouping
g.cluster_method = 'kmeans' # new:clustering_method
g.target_group_index = 0 # Specify the index of the group you want to backtest, starting with 0 (0 is the highest scoring group).

#2 Setting of intermediate variables
def set_variables():
g.feasible_stocks = [] # the current pool of tradable stocks
g.if_trade = False # Whether or not the day is traded
g.num_stocks = 0 # Set the number of stocks to hold


#3 Setting up backtesting conditions
def set_backtest():
set_benchmark('000906.XSHG') # set as benchmark
set_option('use_real_price', True) # Trade at the real price
log.set_level('order', 'error') # Set the error reporting level of the

'''
================================================================================
Before the opening of the market each day
================================================================================
'''
#Things to do every day before the opening bell
def before_trading_start(context):
# Get the current date, #
    day = context.current_dt.day
    yesterday = context.previous_date
    rebalance_day = shift_trading_day(yesterday, 1)
```

```
    if yesterday.month != rebalance_day.month:
        if yesterday.day > rebalance_day.day:
            g.if_trade = True
#5 Set feasible stock pools: get the current open stock pool and exclude stocks that are currently or during the calculation sample period suspended from trading
            g.feasible_stocks = set_feasible_stocks(get_index_stocks(g.index), g.shift1,context)
            #g.feasible_stocks = set_feasible_stocks(get_index_stocks(g.index), max(g.shift1, g.shift2), context)
                    #6 Setting of slippage and handling charges
            set_slip_fee(context)
# Purchase stocks in proportion to the pool of viable stocks
            g.num_stocks = int(len(g.feasible_stocks) * g.percent)

#4

def shift_trading_day(date,shift):
# Get all trading days, return a list of all trading days with datetime.date type.
    tradingday = get_all_trade_days()
# Get the line number in the list for the day after date shift day Returns a number
    shiftday_index = list(tradingday).index(date)+shift
# Returns the date of the day based on the line number as a datetime.date type
    return tradingday[shiftday_index]

#5
# Setting up a pool of viable stocks

def set_feasible_stocks(stock_list,days,context):
# dataframe with information about whether the license is suspended or not, 1 for suspended, 0 for not suspended.
    suspened_info_df = get_price(list(stock_list),
                    start_date=context.current_dt,
                    end_date=context.current_dt,
                    frequency='daily',
                    fields='paused'
    )['paused'].T
# Filter for suspended stocks return dataframe
    unsuspened_index = suspened_info_df.iloc[:,0]<1
# Get the codes of the stocks that are not suspended for the day list.
    unsuspened_stocks = suspened_info_df[unsuspened_index].index
# Further, screen the list of stocks that have not been suspended in the last few days.
    feasible_stocks = []
    current_data = get_current_data()
    for stock in unsuspened_stocks:
        if sum(attribute_history(stock,
                            days,
                            unit = '1d',
                            fields = ('paused'),
                            skip_paused = False
                           )
            )[0] == 0:
            feasible_stocks.append(stock)
# Excluding ST shares
    st_data = get_extras('is_st', feasible_stocks, end_date = context.previous_date, count = 1)
    stockList = [stock for stock in feasible_stocks if not st_data[stock][0]]
    return stockList

#6 Slippage and handling fees based on different time periods
def set_slip_fee(context):
# Set the slippage point to 0
    set_slippage(FixedSlippage(0))
```

```
# Setting handling fees based on different time periods
    dt=context.current_dt

    if dt>datetime.datetime(2013,1, 1):
        set_commission(PerTrade(buy_cost=0.0003,
                                sell_cost=0.0013,
                                min_cost=5))

    elif dt>datetime.datetime(2011,1, 1):
        set_commission(PerTrade(buy_cost=0.001,
                                sell_cost=0.002,
                                min_cost=5))

    elif dt>datetime.datetime(2009,1, 1):
        set_commission(PerTrade(buy_cost=0.002,
                                sell_cost=0.003,
                                min_cost=5))

    else:
        set_commission(PerTrade(buy_cost=0.003,
                                sell_cost=0.004,
                                min_cost=5))
'''
================================================================================
At the time of each day's trading, the
================================================================================
'''
def handle_data(context, data):
# If it's a trading day
    #if g.if_trade == True:
# 7 Get buy and sell signals, input context, output stock list list
# The corresponding default value in the dictionary is false holding_list is filtered to true, then the factor with the highest score is
selected
        #holding_list = get_stocks(g.feasible_stocks, context, asc=g.sort_rank)
# new additions, calculating holding_list length, # new additions, calculating holding_list length
        #total_number = len(holding_list)
        # print 'feasible_stocks is %d, holding is %d' % (len(g.feasible_stocks), total_number)
# Extract the required quantile information
        #(start_q, end_q) = g.quantile
# 8 Rebalance position, enter context, use signal result holding_list
        #rebalance(context, holding_list, start_q, end_q, total_number)
#g.if_trade = False # Assignment
# If it's a trading day
    if g.if_trade:
# Get buy and sell signals, input context, output stock list list
        holding_list = get_stocks(g.feasible_stocks, context)

# Rebalance positions, enter context, use signal result holding_list
        rebalance(context, holding_list)

# Reset the transaction flag
        g.if_trade = False

#7 Raw Data Re-Extraction Factor Scoring Ranking

#7 Raw Data Re-Extraction Factor Scoring Ranking

def get_stocks(stocks_list, context):
```

```
# Obtain BP factor values and preprocess them
    df_bp = get_df_BP(stocks_list, context, g.shift1)

# Perform depolarization
    #df_BP['BP'] = winsorize(df_BP, 'BP', std=3, have_negative=True)
# Normalize the BP factors
    #df_bp['BP'] = standardize(df_bp, 'BP', ty=2)['BP']

# Use get_factor_values to get unweighted price factor data
# Direct access to current unweighted price factor data
    end_date = context.current_dt.strftime('%Y-%m-%d')
    factor_data = get_factor_values(securities=stocks_list, factors=[g.factor2], start_date=end_date, end_date=end_date)

# Directly using current unweighted price data as factor values
# Assuming that the DataFrame contained in factor_data is sorted by date, the most recent data is used here
    df_price_no_fq = factor_data[g.factor2].iloc[-1]




# And then depolarization

# Perform depolarization
    #df_price_no_fq = winsorize(df_price_no_fq, 'price_no_fq', std=3, have_negative=True)

# Standardized
    #df_price_no_fq = standardize(df_price_no_fq.to_frame('price_no_fq'), 'price_no_fq', ty=2)['price_no_fq']

# Ensure that df_bp contains BP values
    df_bp = df_bp[df_bp['BP'].notnull()]


# Combine two factors for a weighted score
    #df_combined = pd.DataFrame(index=df_bp.index)
    #df_combined['score'] = g.factor1_weight * df_bp['BP'] + g.factor2_weight * df_price_no_fq

# Ranked on the basis of a composite score
    #df_combined = df_combined.sort_values(by='score', ascending=asc)
    #return list(df_combined.index)

# Combine the two factors for weighted scores, with z-score standardization first
    df_combined = pd.DataFrame(index=df_bp.index)
    df_combined['BP_zscore'] = (df_bp['BP'] - df_bp['BP'].mean()) / df_bp['BP'].std()
    df_combined['price_no_fq_zscore'] = (df_price_no_fq - df_price_no_fq.mean()) / df_price_no_fq.std()

# In order for both factors to "score as high as possible", the z-score for the non-reweighted price factor is taken to be negative
(taking into account that low-priced stocks should receive high scores).
# Invert the scores of the unweighted price factors so that low prices receive high scores
    df_combined['score'] = g.factor1_weight * df_combined['BP_zscore'] - g.factor2_weight * df_combined['price_no_fq_zscore']

# Adaptive grouping using KMeans clustering
    kmeans = KMeans(n_clusters=g.k_clusters, random_state=0).fit(df_combined[['score']])
    df_combined['cluster'] = kmeans.labels_

# Selection of specific clusters based on cluster scores
    group_scores = df_combined.groupby('cluster')['score'].mean().sort_values(ascending=False)

# Identify specific groups to be selected
```

```
    if g.target_group_index < len(group_scores):
        target_cluster = group_scores.index[g.target_group_index]
        selected_stocks = df_combined[df_combined['cluster'] == target_cluster].index.tolist()
    else:
        selected_stocks = []

    return selected_stocks
#8
# Get a list of stocks to buy based on the strategy's buy signals

#def rebalance(context, holding_list, start_q, end_q, total_number):
    #if end_q == 100:
        #end_q = 100
# Amount purchased per stock
    #every_stock = context.portfolio.portfolio_value/g.num_stocks
# Empty positions only buy operations
    #if len(list(context.portfolio.positions.keys()))==0:
# The original reset scort starts with a return-related scoring calculation, where returns are sorted in ascending order
        # for stock_to_buy in holding_list[int(round(start_q * total_number / 100)) : int(round(end_q * total_number / 100))]:
        #for stock_to_buy in holding_list[int(round(start_q * total_number / 100)) : int(round(end_q * total_number / 100))]:
            #order_target_value(stock_to_buy, every_stock)
    #else :
# Not a short position to first sell shares held but not on the buy list
        #for stock_to_sell in list(context.portfolio.positions.keys()):
            #if stock_to_sell not in holding_list[int(round(start_q * total_number / 100)) : int(round(end_q * total_number / 100))]:
                #order_target_value(stock_to_sell, 0)
# Because the order function is adjusted to the order of adjustment, in order to prevent the first line to adjust the position of the stock due to the latter line to adjust the position of the stock accounted for the amount of too large can not be adjusted in place at once, here to run twice to solve this problem
        #for stock_to_buy in holding_list[int(round(start_q * total_number / 100)) : int(round(end_q * total_number / 100))]:
            #order_target_value(stock_to_buy, every_stock)
        #for stock_to_buy in holding_list[int(round(start_q * total_number / 100)) : int(round(end_q * total_number / 100))]:
            #order_target_value(stock_to_buy, every_stock)

def rebalance(context, holding_list):
# Amount purchased per stock
    every_stock_value = context.portfolio.portfolio_value / len(holding_list)

# Sell stocks that are not in the holding_list first
    for stock in context.portfolio.positions:
        if stock not in holding_list:
            order_target_value(stock, 0)

# Buy or adjust stock positions in holding_list
    for stock in holding_list:
        order_target_value(stock, every_stock_value)



# BP
# Get a dataframe: containing the ticker symbol, book-to-market ratio BP and corresponding rank BP_sorted_rank
# Default date = the day before context.current_dt
def get_df_BP(stock_list, context, asc):

    df_BP = get_fundamentals(query(valuation.code, valuation.pb_ratio
                ).filter(valuation.code.in_(stock_list)))
# Get the pb countdown, #
    df_BP['BP'] = df_BP['pb_ratio'].apply(lambda x: 1/x)
```

```
# Delete nan in case an item in the data does not generate nan
    df_BP = df_BP[pd.notnull(df_BP['BP'])]
# Generate ranking ordinal numbers
    #df_BP['BP_sorted_rank'] = df_BP['BP'].rank(ascending = asc, method = 'dense')
    df_BP['BP_sorted_rank'] = df_BP['BP'].rank(ascending=False, method='dense')
# Use the ticker symbol as the index
    df_BP.index = df_BP.code
# Delete useless data
    del df_BP['code']
    print (df_BP)
    return df_BP



# Depolarization function (3x standard deviation depolarization)
def winsorize(factor_data, factor, std=3,  have_negative = True):
    '''
The depolarization function , the
factor: Series with stock code as index and factor value as value
std is the number of times the standard deviation, have_negative is a boolean, whether or not to include negative values
Output Series
    '''
    r=factor_data[factor]
    if have_negative == False:
        r = r[r>=0]
    else:
        pass
# Taking extreme values
    edge_up = r.mean()+std*r.std()
    edge_low = r.mean()-std*r.std()
    r[r>edge_up] = edge_up
    r[r<edge_low] = edge_low
    r = pd.DataFrame(r)
    return r


# z-score standardized function.
def standardize(factor_data,factor,ty=2):

    temp=factor_data[factor]
    if int(ty)==1:
        re = (temp - temp.min())/(temp.max() - temp.min())
    elif ty==2:
        re = (temp - temp.mean())/temp.std()
    elif ty==3:
        re = temp/10**np.ceil(np.log10(temp.abs().max()))
    return pd.DataFrame(re)
```

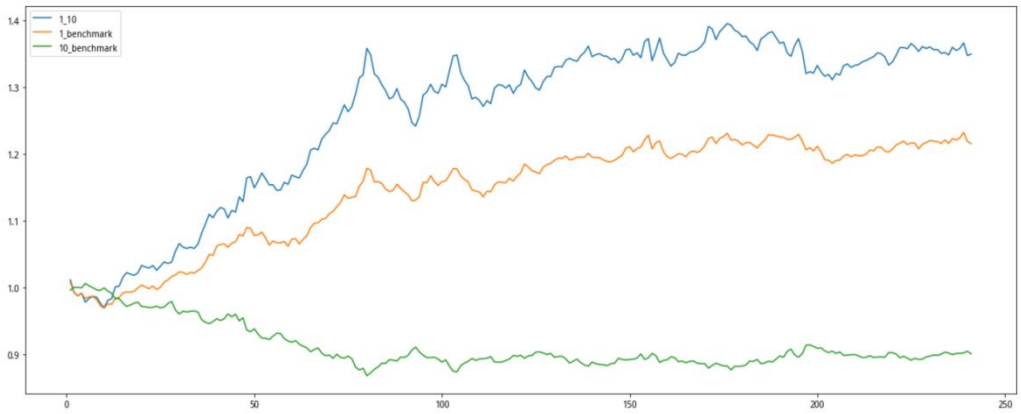
(2) The second dimension: adjusting the parameters of the

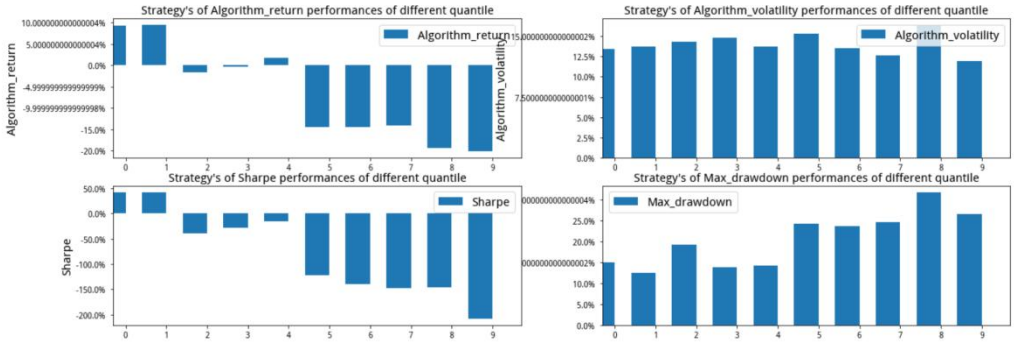
i. change of subject matter

Replaced CSI 800 with subdivided CSI 300 and CSI 500 respectively.

CSI 300 (000300.XSHG) backtest results

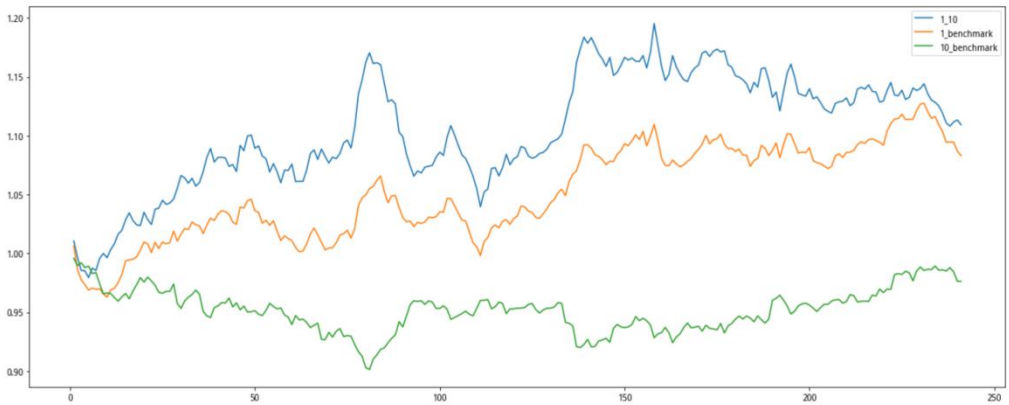
	quantile	__version	algorithm_return	algorithm_volatility	alpha	annual_algo_return	annual_bm_return	avg_excess_return	avg_position_days
0	(0, 10)	101	0.0929203	0.134153	0.174137	0.0961352	-0.10697	0.00083547	186.154
1	(10, 20)	101	0.0947914	0.136807	0.175603	0.098074	-0.10697	0.000844175	148.163
2	(20, 30)	101	-0.0164351	0.142852	0.0829807	-0.0169738	-0.10697	0.000394356	135.038
3	(30, 40)	101	-0.00235189	0.148267	0.108253	-0.00242954	-0.10697	0.000451353	122.424
4	(40, 50)	101	0.0183662	0.137701	0.121444	0.018979	-0.10697	0.000534132	101.942
5	(50, 60)	101	-0.144375	0.153387	-0.0273636	-0.148774	-0.10697	-0.000185086	112.531
6	(60, 70)	101	-0.145154	0.135557	-0.0489842	-0.149574	-0.10697	-0.000189576	101.969
7	(70, 80)	101	-0.141341	0.12586	-0.0586152	-0.145655	-0.10697	-0.000170084	117.706
8	(80, 90)	101	-0.193055	0.1635	-0.0844074	-0.198757	-0.10697	-0.00041752	127.885
9	(90, 100)	101	-0.202126	0.118951	-0.138389	-0.20806	-0.10697	-0.000468807	169.029

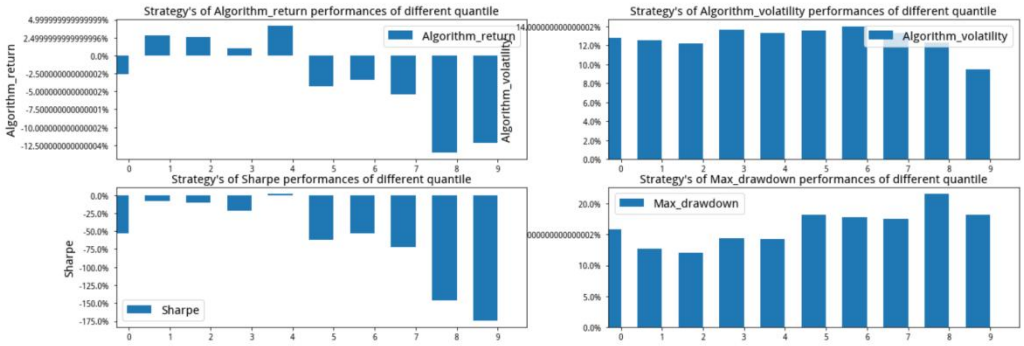




CSI 500 (000905.XSHG) backtest results

	quantile	__version	algorithm_return	algorithm_volatility	alpha	annual_algo_return	annual_bm_return	avg_excess_return	avg_position_days
0	(0, 10)	101	-0.0265675	0.127849	0.042262	-0.0274336	-0.10697	0.000357279	182.785
1	(10, 20)	101	0.028517	0.125363	0.102704	0.0294735	-0.10697	0.000581996	134.523
2	(20, 30)	101	0.0256752	0.121922	0.0992317	0.0265352	-0.10697	0.000569118	118.802
3	(30, 40)	101	0.0104612	0.136674	0.103698	0.0108089	-0.10697	0.000505714	105.439
4	(40, 50)	101	0.0414561	0.133007	0.133966	0.0428555	-0.10697	0.000629472	101.534
5	(50, 60)	101	-0.0434252	0.135917	0.0488864	-0.0448281	-0.10697	0.000278392	90.8702
6	(60, 70)	101	-0.033701	0.140429	0.0617399	-0.0347954	-0.10697	0.00032138	91.2756
7	(70, 80)	101	-0.0543349	0.133236	0.0253264	-0.0560798	-0.10697	0.000235125	98.6726
8	(80, 90)	101	-0.135762	0.122556	-0.0662488	-0.139921	-0.10697	-0.000138875	111.645
9	(90, 100)	101	-0.122236	0.095209	-0.0819422	-0.126011	-0.10697	-7.30349e-05	137.107





Conclusion.

- In all three pools, returns in quintile 1 were generally higher than those in quintile 10, in line with the strategy's expectations, indicating that stocks with high long scores (i.e., quintile 1) and low short scores (i.e., quintile 10) were able to generate positive returns.
- The Sharpe ratios varied across quartiles, but in general, quartile 1 had higher Sharpe ratios, suggesting that, after adjusting for risk, this group was able to generate better returns.
- The maximum retracement data show that, while quartile 1 may be subject to higher downside risk during certain periods, it is more stable over the long term.
- The NAV graph shows the performance of the strategy over time for different pools of stocks. The return curves of the CSI 800 and the CSI 300 are close to each other, while the CSI 500 underperforms on a relative basis.

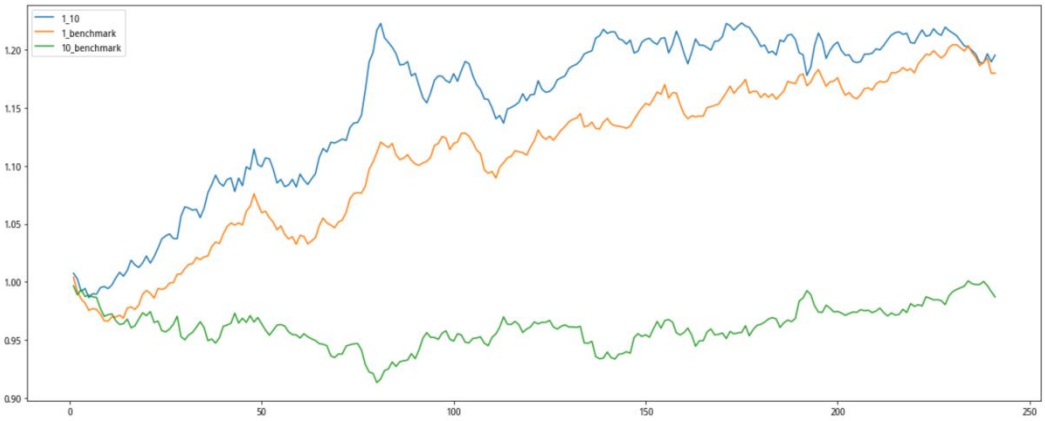
Comparing the backtest results of different stock indices, it can be seen that the CSI 300 strategy profitability is arranged in a regular manner with the quartile value from high to low, and the effect of applying this two-factor strategy is better than that of the CSI 500 and the CSI 800. In large-cap stocks, the combination of the BP factor and the non-compounded price factor may be more capable of capturing solid profitability opportunities, whereas in the mid- and small-cap stocks, which usually have a higher growth rate, is accompanied by higher market volatility risk, which may be the reason why the strategy shows greater volatility in these indices. This may be the reason for the strategy's more volatile performance on these indices.

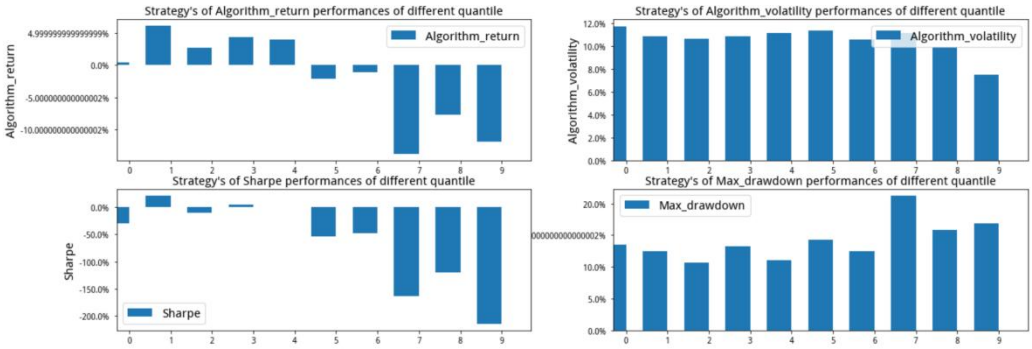
ii. changing the frequency of transactions

The underlying is still CSI 800, and the frequency of trading has been changed from daily to weekly or monthly.

Trading on a weekly basis

	quantile	__version	algorithm_return	algorithm_volatility	alpha	annual_algo_return	annual_bm_return	avg_excess_return	avg_position_days
0	(0, 10)	101	0.00450328	0.117792	0.0696528	0.00465249	-0.10697	0.000484399	177.02
1	(10, 20)	101	0.0613375	0.109023	0.125033	0.0634282	-0.10697	0.000709566	148.965
2	(20, 30)	101	0.0273328	0.106721	0.0883471	0.028249	-0.10697	0.000574608	135.32
3	(30, 40)	101	0.0441652	0.109184	0.114196	0.0456581	-0.10697	0.000639011	127.531
4	(40, 50)	101	0.0393147	0.111931	0.113901	0.0406405	-0.10697	0.000618836	113.461
5	(50, 60)	101	-0.0215031	0.114108	0.0554393	-0.022206	-0.10697	0.000368517	122.481
6	(60, 70)	101	-0.0111331	0.105694	0.053969	-0.011499	-0.10697	0.000413979	113.121
7	(70, 80)	101	-0.138254	0.111528	-0.0675328	-0.142482	-0.10697	-0.000156495	127.511
8	(80, 90)	101	-0.0772231	0.0995185	-0.0208487	-0.0796714	-0.10697	0.00012856	138.86
9	(90, 100)	101	-0.118459	0.0755487	-0.0914628	-0.122126	-0.10697	-5.58327e-05	146.873

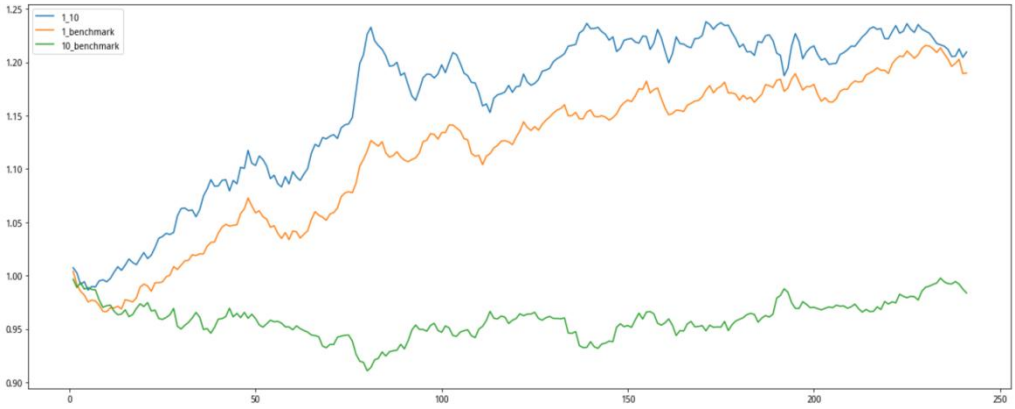


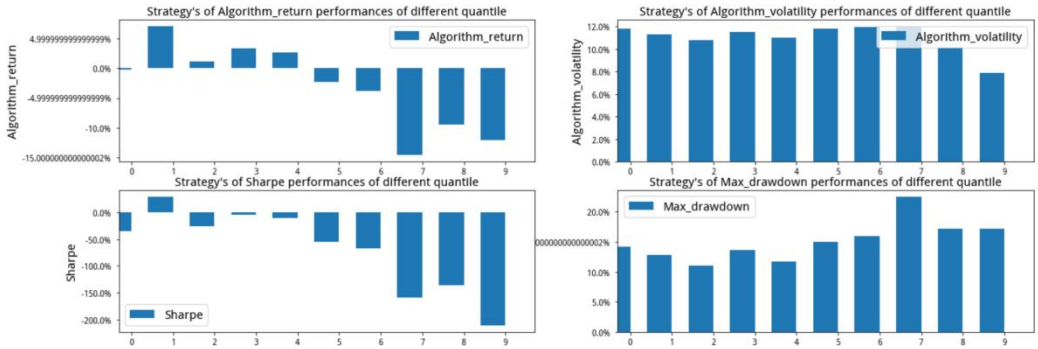


Ttest_1sampResult(statistic=34.46555890573732, pvalue=6.509015017963465e-95)

monthly transactions

	quantile	__version	algorithm_return	algorithm_volatility	alpha	annual_algo_return	annual_bm_return	avg_excess_return	avg_position_days
0	(0, 10)	101	-0.00210573	0.11829	0.0623002	-0.00217527	-0.10697	0.000457578	173.396
1	(10, 20)	101	0.0702159	0.112687	0.138105	0.0726194	-0.10697	0.000743857	142.426
2	(20, 30)	101	0.0111085	0.108176	0.0735108	0.0114779	-0.10697	0.000508536	117.8
3	(30, 40)	101	0.0332882	0.1154	0.110106	0.0344074	-0.10697	0.000595219	111.913
4	(40, 50)	101	0.0265269	0.109744	0.101208	0.0274157	-0.10697	0.000566442	101.73
5	(50, 60)	101	-0.0237551	0.118385	0.0575594	-0.0245307	-0.10697	0.00035894	103.259
6	(60, 70)	101	-0.0388231	0.11987	0.0413453	-0.0400804	-0.10697	0.000295649	101.781
7	(70, 80)	101	-0.146103	0.119694	-0.0670193	-0.15055	-0.10697	-0.000194453	114.849
8	(80, 90)	101	-0.094003	0.100968	-0.0370516	-0.0969549	-0.10697	5.27984e-05	127.496
9	(90, 100)	101	-0.121384	0.0785431	-0.0910571	-0.125134	-0.10697	-7.02036e-05	145.394






```
Ttest_1sampResult(statistic=34.21461717481858, pvalue=2.79843954270221e-94)
```

Conclusion.

- based on an algorithmic intuitive analysis

The backtesting results show that the weekly trading strategy outperforms the daily and monthly trading frequency in terms of key metrics such as return, maximum retracement, and Sharpe ratio. This may be related to the market dynamics captured by the BP factor, which represents the market's assessment of a stock's value over the long term, and the DWP factor, which captures short-term market volatility and sentiment. In weekly trading, the strategy may better balance the assessment of long-term value with the reaction to short-term market dynamics, resulting in better risk-adjusted returns. In contrast, trading on a daily basis may

overemphasize short-term market fluctuations, increasing volatility and affecting the stability of the strategy. Monthly trading may not capture valid short-term market information in a timely manner, resulting in the loss of potential return opportunities.

- Based on parametric sensitivity analysis, the

The backtesting results show that the trading frequency has a high sensitivity to the strategy, especially considering the changes in volatility and maximum retracement. This effect is manifested in the volatility of the backtest curves, which are more volatile for high-frequency trades, reflecting the faster uptake and release of information from the market, necessitating a careful choice of trading frequency.

The robustness of the strategy needs further analysis, especially the ability to maintain a stable performance under parameter variations.

Single-factor optimal trading frequency.

1,BP single factor

Trading on a daily basis

Trading on a weekly basis

Monthly

2, Unweighted price factor

Trading on a daily basis

Trading on a weekly basis

monthly transactions

We can find that both two-factor (non-duplicated price factor and BP factor) and one-factor (non-duplicated price factor and BP factor alone, respectively) strategies show optimal performance for trading on a weekly basis in experiments that vary the frequency of trading

Conclusion.
Both the one-factor and two-factor strategies are optimized for weekly trading at different trading frequencies, which shows the stability and reliability of the strategies in the face of market movements and increases the robustness of the strategies. This shows the stability and reliability of the strategy in the face of market movements, which increases the robustness of the strategy. Because the weekly trading frequency may provide the strategy with a buffer against market volatility, it also reduces the cost of frequent trading and the potential for execution error, which is beneficial for improving long-term investment returns. Combining the strengths of both factors captures short-term market dynamics while taking into account the fundamental value of the stock.

iii. changing factor weights

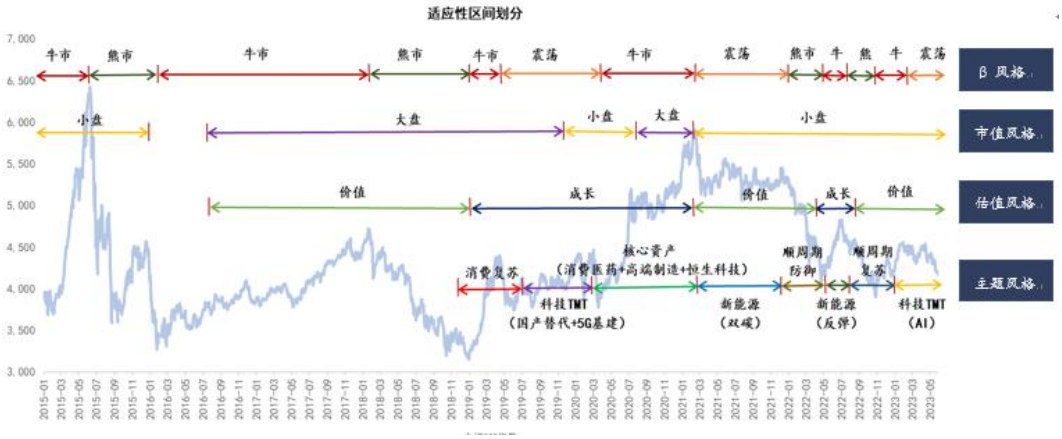
The BP factor weights are adjusted from 0 to 1 in steps of 0.1, while the weights of the uncompounded price factor are adjusted from 1 to 0 in steps of 0.1, and the best results are found when the BP factor weights are 0.7 and the uncompounded price factor weights are 0.3.

This may indicate that value investing (BP factor) contributes more to the strategy than market sentiment (unweighted price factor) over the time horizon and market conditions tested This weighted portfolio may be the optimal portfolio for the current market conditions, balancing the long-term value of the market with short-term price dynamics. If this portfolio performs well over multiple historical cycles and in different market environments, then it may point to a more robust strategy allocation. Therefore, it is further analyzed by changing the market cycle below.

(iii) The third dimension: robustness judgments under different market cycles

Roughly based on market cycles (source: Leadership Institute, Wind).

市场情景划分	划分具体参考指标	细分情形	划分区间个数
市场行情 β	中证 800 指数涨跌幅	牛市	6
		熊市	4
		震荡市	3
市值风格	(沪深 300/中证 1000) 收益率	大盘	2
		小盘	3
估值风格	(全指成长/全指价值) 收益率	成长	2
		价值	3
市场主线	利得主题涨跌幅极差	消费复苏、科技 TMT (国产替代+5G 新基建)、核心资产 (消费医药+高端制造+恒生科技)、新能源 (反弹)、顺周期复苏、科技 TMT (AI)	8

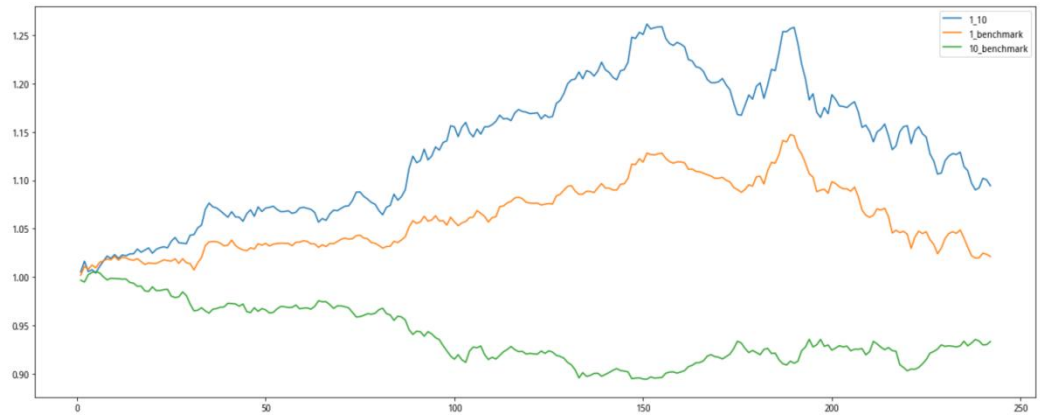


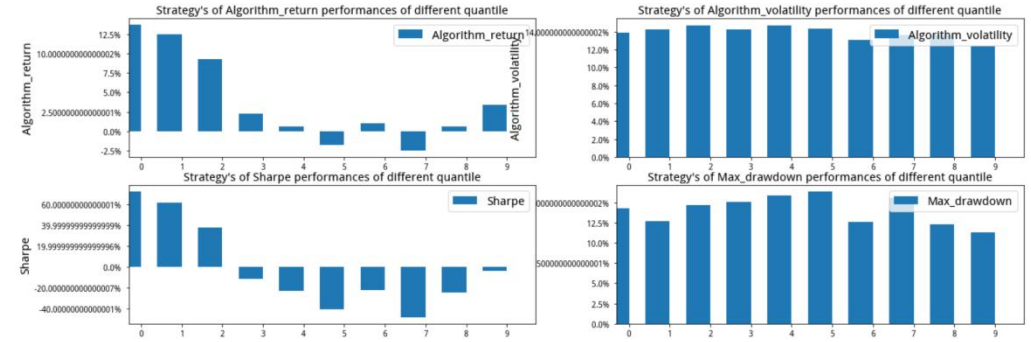
The bull market of CSI 800 index is from July 2016 to July 2017; the bear market is from January 2018 to January 2019; and the oscillator market is from March 2021 to December 2021, and the bull market of CSI 800 index is from July 2016 to July 2017; the bear market is from January 2018 to January 2019; and the oscillator market is from March 2021 to December 2021

The original backtesting period was January 1, 2023 - January 1, 2024, and the
The results of the backtests based on the CSI 800 stock pool, varying the market cycle, are as follows.

1. Bull market (July 2016-July 2017)

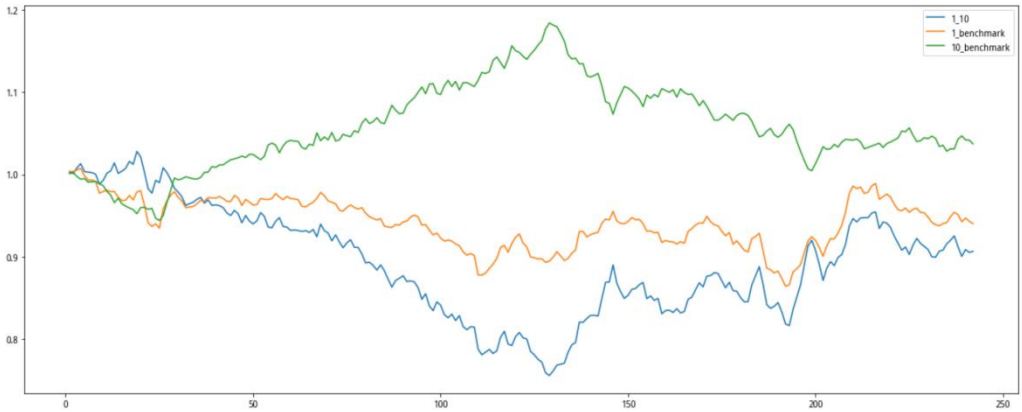
	quantile	__version	algorithm_return	algorithm_volatility	alpha	annual_algo_return	annual_bm_return	avg_excess_return	avg_position_days
0	(0, 10)	101	0.136955	0.138613	0.0180173	0.141167	0.117247	9.53825e-05	165.769
1	(10, 20)	101	0.124426	0.142728	-0.00254523	0.128231	0.117247	4.73722e-05	109.607
2	(20, 30)	101	0.0930486	0.147256	-0.0361302	0.0958536	0.117247	-6.72021e-05	97.1685
3	(30, 40)	101	0.0229987	0.142362	-0.10579	0.023669	0.117247	-0.000340935	84.1878
4	(40, 50)	101	0.00624949	0.146527	-0.126628	0.0064301	0.117247	-0.000408665	84.5681
5	(50, 60)	101	-0.0176492	0.143436	-0.147245	-0.018153	0.117247	-0.000506851	81.1847
6	(60, 70)	101	0.0102364	0.1309	-0.114379	0.0105328	0.117247	-0.000396026	88.5971
7	(70, 80)	101	-0.0253135	0.136031	-0.152312	-0.0260331	0.117247	-0.000541491	93.2132
8	(80, 90)	101	0.00597415	0.135751	-0.120901	0.00614678	0.117247	-0.000412128	99.6793
9	(90, 100)	101	0.0343096	0.123546	-0.0794836	0.0353152	0.117247	-0.000296669	147.919

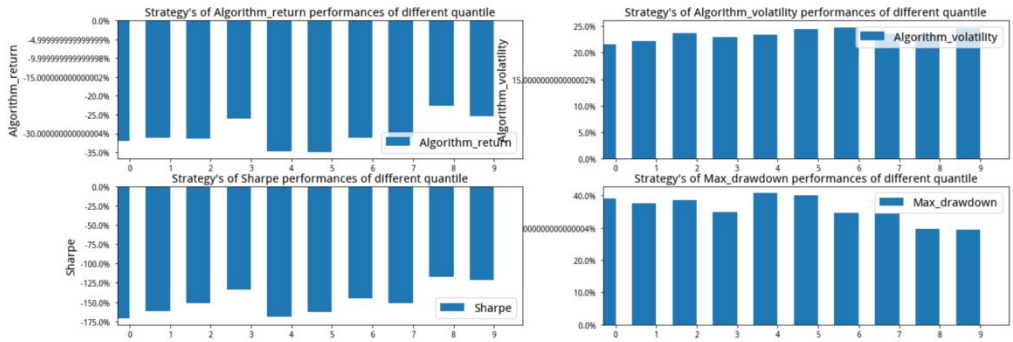




2. Bear Market (January 2018-January 2019)

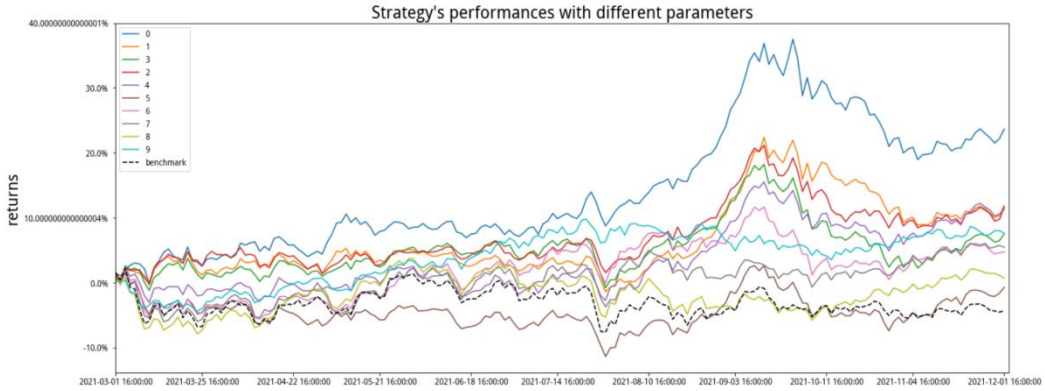
	quantile	__version	algorithm_return	algorithm_volatility	alpha	annual_algo_return	annual_bm_return	avg_excess_return	avg_position_days
0	(0, 10)	101	-0.319913	0.215262	-0.0928828	-0.327425	-0.280496	-0.00024188	149.377
1	(10, 20)	101	-0.310946	0.222118	-0.0631547	-0.318299	-0.280496	-0.000193639	111.92
2	(20, 30)	101	-0.312837	0.237271	-0.0402928	-0.320224	-0.280496	-0.000205001	94.1443
3	(30, 40)	101	-0.259681	0.228569	0.00773439	-0.266065	-0.280496	9.67576e-05	85.5822
4	(40, 50)	101	-0.346625	0.233306	-0.0734875	-0.354587	-0.280496	-0.000417	85.0093
5	(50, 60)	101	-0.349788	0.244464	-0.0635781	-0.357801	-0.280496	-0.000433718	84.8981
6	(60, 70)	101	-0.312018	0.248078	-0.0129158	-0.31939	-0.280496	-0.000204972	91.4356
7	(70, 80)	101	-0.310578	0.235829	-0.0288249	-0.317924	-0.280496	-0.000198471	99.9086
8	(80, 90)	101	-0.227079	0.23338	0.0550807	-0.232792	-0.280496	0.000270297	112.271
9	(90, 100)	101	-0.253188	0.246301	0.0391328	-0.259443	-0.280496	0.000135138	146.1

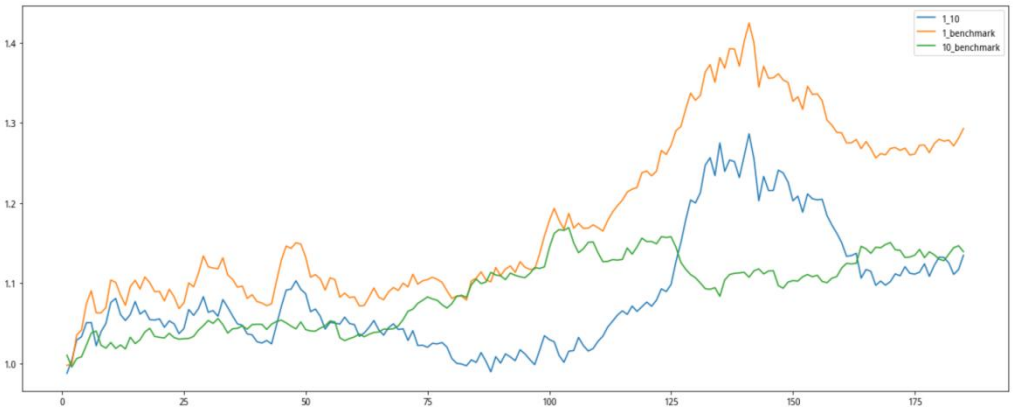


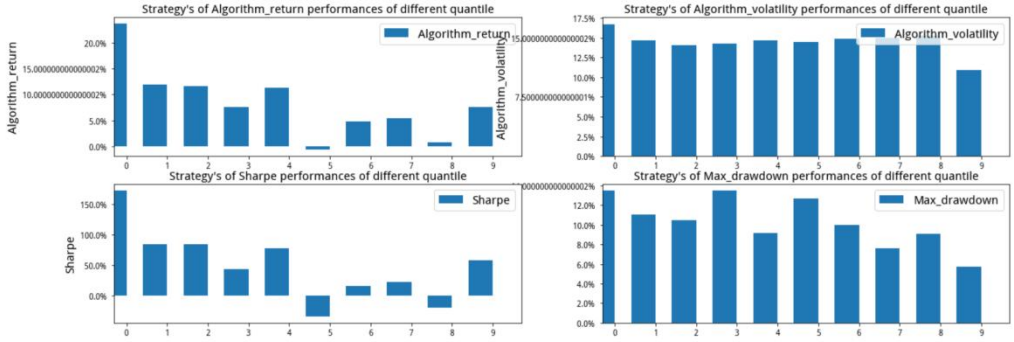


3. Oscillatory market (March 2021-December 2021)

	quantile	__version	algorithm_return	algorithm_volatility	alpha	annual_algo_return	annual_bm_return	avg_excess_return	avg_position_days
0	(0, 10)	101	0.23659	0.167389	0.328746	0.33033	-0.0568648	0.00144397	121.434
1	(10, 20)	101	0.119076	0.146452	0.161801	0.163246	-0.0568648	0.000893171	92.8025
2	(20, 30)	101	0.116567	0.140712	0.165306	0.159742	-0.0568648	0.000870145	82.764
3	(30, 40)	101	0.0753523	0.14244	0.11677	0.102572	-0.0568648	0.000659343	79.3422
4	(40, 50)	101	0.112956	0.146271	0.180435	0.154704	-0.0568648	0.00083372	72.6453
5	(50, 60)	101	-0.00668919	0.144929	0.0171775	-0.00898048	-0.0568648	0.00022066	77.8466
6	(60, 70)	101	0.047528	0.148851	0.0980794	0.0643988	-0.0568648	0.000500606	84.7126
7	(70, 80)	101	0.0548267	0.149338	0.113181	0.0743788	-0.0568648	0.000532584	87.3293
8	(80, 90)	101	0.0072388	0.153087	0.0510349	0.00974167	-0.0568648	0.000283744	95.7114
9	(90, 100)	101	0.0763001	0.108779	0.112601	0.103878	-0.0568648	0.000653008	103.827







Conclusion.

- In the bull market, two-factor strategies (especially the first group) showed strong investment value in the initial phase, but the growth was not sustained in the later phase.
- In a bear market, there is a lack of consistency with expectations. The first quartile (the group expected to perform best) did not actually deliver the highest returns. Instead, their performance did not seem to follow a clear, consistent trend between quartiles.
- In an oscillating market, strategy performance is volatile, and this market uncertainty can have a large impact on the strategy.

The analysis of three market cycles shows that the effectiveness of this two-factor strategy is highly dependent on market conditions. At the beginning of a bull market, the strategy may identify stocks that are truly undervalued and have upside potential. However, in a bear market, the overall market sentiment is pessimistic, which may cause investors to withdraw from the stock market, even for stocks that appear to be reasonably valued. Second, strategies based on BP and non-weighted prices may not take into account the impact of macroeconomic factors and market liquidity, which have a significant impact on stock prices during bear markets. Strategies may no longer be effective in oscillating markets, as market sentiment and volatility may cause the fundamental analysis of stocks to be less accurate.

To summarize

The two-factor strategy combines the value assessment of the BP factor with the dynamic monitoring of the market price of the uncompounded price factor, and through this combination, it is possible to identify stocks that are undervalued by the market and have the potential to rise in price. However, its performance is limited by the following market conditions and strategies.

1. Optimal application period and stock pool selection.

    a. At the beginning of the bull market, choosing large-cap stocks (e.g. CSI 300 Index) as the main pool of stocks can realize better returns by taking advantage of the BP factor and the non-duplicated price factor.
    b. During bear or shock markets, the weighting of small and mid-cap stocks (e.g. CSI 500 Index) can be reduced or shifted to more stable large-cap stocks, while the weighting of the BP factor can be increased to better protect against market volatility and capture undervalued investment opportunities.

The direction of other adjustment strategies is as follows: (1) In bear or shock markets, introduce a dynamic adjustment mechanism to adjust factor weights according to changes in market volatility and investor sentiment. For example, when the market volatility increases, the weight of BP factor can be increased because value investment is more important in this

environment. (2) Introduce additional factors such as market sentiment indicators, liquidity indicators or volatility factors to help better understand the short-term dynamics of the market, thereby improving the performance of the strategy in unstable market environments.

2. Possible trade setups.

a. Trading Frequency: Weekly trading shows better performance than daily and monthly trading because it better balances the assessment of long-term value with the response to short-term market dynamics.
b. Factor weights.
- During periods of market instability or economic uncertainty, a value investing strategy that increases the weight of the BP factor can help investors capitalize on opportunities that are mispriced by the market. This is because investors tend to look for stocks that are financially sound and reasonably valued. In addition, because the BP Factor reflects a long-term assessment of value, it is less sensitive to short-term market fluctuations, which can help stabilize the strategy in volatile markets.
- For the January 1, 2023 to January 1, 2024 testing period, a BP factor weighting of 0.7 and an uncompounded price factor weighting of 0.3 show the best risk-adjusted returns. This weighting is suitable for the current market environment and can effectively capture the long-term value and short-term price dynamics. However, machine learning methods should be used later to optimize the factor weights and stock selection process, which can help to make more accurate predictions in an uncertain market environment.

3. Risk Warning.

a. In the exploration of improved scoring methods (Z-score standardized scoring), the threshold adjustment of the extremes of the percentile limits is discussed in terms of optimization based on data from the current backtesting period, and there may be a risk of future functioning and overfitting.

b. The parameters (e.g. factor weights) and models used in the strategy are not applicable to all market conditions. Parameters and models should be regularly evaluated and adjusted according to market changes and actual performance. Blind reliance on a single parameter setting or model forecast is not appropriate.

Appendix.

1,Parameter sensitivity analysis

```
#1 Pre-load the required program package
import datetime
import numpy as np
import pandas as pd
import time
from jqdata import *
from pandas import Series, DataFrame
import matplotlib.pyplot as plt
import seaborn as sns
import itertools
import copy
import pickle

# Define the class 'Parameter Analysis'
class parameter_analysis(object):

    # define different variables in the function
    def __init__(self, algorithm_id=None):
        self.algorithm_id = algorithm_id # backtest id

        self.params_df = pd.DataFrame() # backtest the contents of all the alternative values of the parameter, the column name is the name of the corresponding modification surface, corresponding to the backtest in the g.XXXX, the contents of all the alternative values of the parameter, the column name is the name of the corresponding modification surface, corresponding to the backtest in the g.XXXX
        self.results = {} # The return rate of the backtest results, key is the row number of params_df, value is, and the return rate is the number of rows in the params_df
        self.evaluations = {} # The metrics of the backtest results, key is the row number of params_df, value is a dataframe, self.evaluations = {} # The metrics of the backtest results, key is the row number of params_df, and value is a dataframe
        self.backtest_ids = {} # The ids of the backtest results

        # The backtest result id of the newly added benchmark, which can default to null '', then the benchmark set in the backtest is used
        self.benchmark_id = 'f16629492d6b6f4040b2546262782c78'

        self.benchmark_returns = [] # The backtested returns of the newly added benchmark, the
        self.returns = {} # Record all the returns, the
        self.excess_returns = {} # Record the excess returns, the
        self.log_returns = {} # Log the value of the returns
        self.log_excess_returns = {} # Log the value of the excess returns
        self.dates = [] # backtest all the dates corresponding to the
        self.excess_max_drawdown = {} # Calculate the maximum drawdown for excess returns, the
```

```
        self.excess_annual_return = {} # Calculate the annualized metric for excess return, the
        self.evaluations_df = pd.DataFrame() # Record the various backtesting metrics, except for the daily returns
        self.failed_list= []

    # Define the queue to run the multi-parameter backtest function
    def run_backtest(self,                          #
                     algorithm_id=None, # backtest strategy id
                     running_max=10, # Maximum number of backtests that can be run at the same time in a backtest
                     start_date='2006-01-01', # The start date of the backtest, the
                     end_date='2016-11-30', # The end date of the backtest, the
                     frequency='day', # how often the backtest was run
                     initial_cash='1000000', # the initial position amount for the backtest
                     param_names=[], # Variables involved in adjusting parameters in the backtest
                     param_values=[], # alternative parameter values for each variable in the backtest
                     python_version = 2, # python version for backtesting
                     use_credit =False # Whether to allow consumption of credits to continue backtesting, the
                     ):
        # When the id of the strategy to be backtested here is not given, call the strategy id entered by the class
        if algorithm_id == None: algorithm_id=self.algorithm_id

        # Generate all parameter combinations and load them into df
        # A list of tuples containing a set of parameters in a permutation of specific alternative values for different parameters
        param_combinations = list(itertools.product(*param_values))
        # Generate a dataframe, with columns corresponding to the variables for each of the callbacks, each value being an alternative value for the callbacks
        to_run_df = pd.DataFrame(param_combinations,dtype='object')
        # Modify the name of the column called the marshaling variable
        to_run_df.columns = param_names

        # Setting the run start time and save format
        start = time.time()
        # Recorded end-of-run backtests
        finished_backtests = {}
        # Documentation of running backtests
        running_backtests = {}
        # Counters
        pointer = 0
        # Total number of backtests run, equal to the number of elements in the permutation
        total_backtest_num = len(param_combinations)
        # Record the rate of return on the results of backtesting
        all_results = {}
        # Indicators documenting the results of backtesting
        all_evaluations = {}
```

```
    # Displayed at the start of the run
    print(('[Completed|Running|To be run]:'), end=' ')

    cannot_get = {} # Failed to get , retry plus one

    # When the runback test starts, if it doesn't run all the way through.
    while len(finished_backtests)<total_backtest_num:
        # Displaying the number of backtests running, completed and pending
        print(('[%s|%s|%s].' % (len(finished_backtests),
                        len(running_backtests),
                        (total_backtest_num-len(finished_backtests)-len(running_backtests)) )), end=' ')
        # Record the number of empty spaces in the current run
        to_run = min(running_max-len(running_backtests), total_backtest_num-len(running_backtests)-len(finished_backtests))
        # Run backtests of available openings
        for i in range(pointer, pointer+to_run):
            # Alternative parameter permutations of row i in the df into dict, each key is a column name, value is the corresponding
value in the df
            params = to_run_df.iloc[i].to_dict()
            # Record the id of the strategy backtest result, adjust the parameter extras to use the contents of params
            backtest = create_backtest(algorithm_id = algorithm_id,
                                start_date = start_date,
                                end_date = end_date,
                                frequency = frequency,
                                initial_cash = initial_cash,
                                extras = params,
                                # Re-test the results of the parameterization with a name that contains the values of all the variables
involved
                                name = str(params),
                                python_version = python_version,
                                use_credit = use_credit
                                )
            # Record the backtest id of the running i backtest
            running_backtests[i] = backtest
        # The counter counts the number of runs completed , the
        pointer = pointer+to_run

        # Getting backtest results
        failed = []
        finished = []

        # For a running backtest, the key is the ordinal number of all permutations in to_run_df
        for key in list(running_backtests.keys()):
            # study the results of the call backtests, running_backtests[key] is the id of the result saved in the run
            try :
```

```
                back_id = running_backtests[key]
                bt = get_backtest(back_id)
                # Get the status of the results of the run backtest, both success and failure need to be returned at the end of the run, if not returned then the run is not finished
                status = bt.get_status()
                # When running a backtest fails
                if status == 'failed':
                    # Record the corresponding backtest result id in the failure list
                    print('')
                    print(('Backtest failed : https://www.joinquant.com/algorithm/backtest/detail?backtestId='+back_id))
                    failed.append(key)
                # When running a successful backtest
                elif status == 'done':
                    # Successful list Record the corresponding backtest result id, finish Record only the successful run
                    finished.append(key)
                    # return return record corresponding to the return dict, key to_run_df all permutations in the ordinal, value is the dict of the return, # return return record corresponding to the return dict, key to_run_df all permutations in the ordinal, value is the dict of the return
                    # each value a list each object is a dict containing the time, daily return and base return
                    all_results[key] = bt.get_results()
                    # backtest return record corresponding to the backtest result indicator dict, key to_run_df all permutations in the ordinal, value is the dataframe of the backtest result indicator
                    all_evaluations[key] = bt.get_risk()
                        # Remove failed runs from the list of backtest result ids in the run
            except Exception as e:
                count = cannot_get.get( key)
                if not count :
                    cannot_get[key] = 1
                    count = 0
                elif count < 5:
                    cannot_get[key] = count +1
                else :
                    failed.append(key)
                print("Failed to get backtest information {}, retried {} times , total {} times \n {} ".format(back_id, count ,5 ,e) )
                time.sleep(2)
        # Remove failed runs from the list of backtest result ids in the run
        for key in failed:
            finished_backtests[key] = running_backtests.pop(key)
        # Record the id of the successfully run backtest in the end backtest result dict, and delete the backtest from the running record
        for key in finished:
            finished_backtests[key] = running_backtests.pop(key)
#               print (finished_backtests)
        # Report time when a set of simultaneously running backtests ends
```

```
            if len(finished_backtests) != 0 and len(finished_backtests) % running_max == 0 and to_run !=0:
                # Recording the time
                middle = time.time()
                # Calculate the remaining time, assuming no workload time is equal
                remain_time = (middle - start) * (total_backtest_num - len(finished_backtests)) / len(finished_backtests)
                # print the current runtime
                print((('[Don't close the browser when %s has been used and %s is left].')) % (str(round((middle - start) / 60.0 / 60.0,3)),
                                    str(round(remain_time / 60.0 / 60.0,3)))), end=' ')
            self.failed_list  += failed
            # 5 seconds and then a little run
            time.sleep(5)
        # Record end time
        end = time.time()
        print('')
        print((('[Backtest completed] total time:%s seconds (i.e. %s hours).') % (str(int(end-start)),
                                    str(round((end-start)/60.0/60.0,2)))), end=' ')
#       print (to_run_df,all_results,all_evaluations,finished_backtests)
        # Correspondence modifies the internal correspondence of the class
#       to_run_df = {key:value for key,value in returns.items() if key not in faild}
        self.params_df = to_run_df
#       all_results = {key:value for key,value in all_results.items() if key not in faild}
        self.results = all_results
#       all_evaluations = {key:value for key,value in all_evaluations.items() if key not in faild}
        self.evaluations = all_evaluations
#       finished_backtests = {key:value for key,value in finished_backtests.items() if key not in faild}
        self.backtest_ids = finished_backtests


    #7 Maximum retracement calculation methodology
    def find_max_drawdown(self, returns):
        # Variables defining the maximum retracement of
        result = 0
        # Highest point of return on record
        historical_return = 0
        # Iterate through all the dates
        for i in range(len(returns)):
            # The highest rate of return on record
            historical_return = max(historical_return, returns[i])
            # Maximum retracements recorded
            drawdown = 1-(returns[i] + 1) / (historical_return + 1)
            # Record maximum retracements
            result = max(drawdown, result)
        # Returns the maximum retraction value
        return result
```

```
    # log return, excess return under the new benchmark and maximum retracement relative to the new benchmark
    def organize_backtest_results(self, benchmark_id=None):
        # If the backtest result id of the new benchmark is not given, the
        if benchmark_id==None:
            # Using the default benchmark return, which is set in the backtesting strategy
            self.benchmark_returns = [x['benchmark_returns'] for x in self.results[0]]
        # When new baseline indicators are given
        else:
            # Benchmarks use the results of newly added benchmark backtests
            self.benchmark_returns = [x['returns'] for x in get_backtest(benchmark_id).get_results()]
        # The date of the backtest is the date corresponding to the first item recorded in the results
        self.dates = [x['time'] for x in self.results[0]]

        # Generate new data corresponding to the order (key) of each backtest among all alternative backtests, # Generate new data
corresponding to the order (key) of each backtest among all alternative backtests
        # by {key:{u'benchmark_returns': 0.022480100091729405,
        #          u'returns': 0.03184566700000002,
        # u'time': u'2006-02-14'}} The format is converted to.
        # In {key: []} format, where list is a list of returns for the corresponding date
        for key in list(self.results.keys()):
            self.returns[key] = [x['returns'] for x in self.results[key]]
        # Generate an excess return over the benchmark (or new benchmark)
        for key in list(self.results.keys()):
            self.excess_returns[key] = [(x+1)/(y+1)-1 for (x,y) in zip(self.returns[key], self.benchmark_returns)]
        # Generate a rate of return in log form, #
        for key in list(self.results.keys()):
            self.log_returns[key] = [log(x+1) for x in self.returns[key]]
        # Generate the log form of the excess return
        for key in list(self.results.keys()):
            self.log_excess_returns[key] = [log(x+1) for x in self.excess_returns[key]]
        # Maximum retracement for generating excess return
        for key in list(self.results.keys()):
            self.excess_max_drawdown[key] = self.find_max_drawdown(self.excess_returns[key])
        # Generate annualized excess returns
        for key in list(self.results.keys()):
            self.excess_annual_return[key] = (self.excess_returns[key][-1]+1)**(252./float(len(self.dates)))-1
        # Merge the parameter combination df from the parameterization data with the corresponding result df
        self.evaluations_df = pd.concat([self.params_df, pd.DataFrame(self.evaluations).T], axis=1)
#       self.evaluations_df =

    # Obtain aggregate analytical data, call the queuing backtest function and the function that organizes the data, #
    def get_backtest_data(self,
                          algorithm_id=None, # backtest strategy id
```

```
                benchmark_id=None, # new benchmark backtest result id
                file_name='results.pkl', # name of the pickle file where the results are stored
                running_max=10, # Maximum number of simultaneous running backtests
                start_date='2006-01-01', # start time of the backtest
                end_date='2016-11-30', # end_date_of_backtest
                frequency='day', # how often the backtest was run
                initial_cash='1000000', # backtesting initial position funding
                param_names=[], # backtest the variables that need to be tested
                param_values=[], # alternative parameters corresponding to each variable
                python_version = 2,
                use_credit = False
                ):
# Call the queuing backtest function, passing in the corresponding parameters
self.run_backtest(algorithm_id=algorithm_id,
                running_max=running_max,
                start_date=start_date,
                end_date=end_date,
                frequency=frequency,
                initial_cash=initial_cash,
                param_names=param_names,
                param_values=param_values,
                python_version = python_version,
                use_credit = use_credit,
                )
# Inclusion of indicators such as log return and excess return in the metrics for backtesting results
self.organize_backtest_results(benchmark_id)
# Generate dict to save all results.
results = {'returns':self.returns,
        'excess_returns':self.excess_returns,
        'log_returns':self.log_returns,
        'log_excess_returns':self.log_excess_returns,
        'dates':self.dates,
        'benchmark_returns':self.benchmark_returns,
        'evaluations':self.evaluations,
        'params_df':self.params_df,
        'backtest_ids':self.backtest_ids,
        'excess_max_drawdown':self.excess_max_drawdown,
        'excess_annual_return':self.excess_annual_return,
        'evaluations_df':self.evaluations_df,
         "failed_list" : self.failed_list}
# Save the pickle file
pickle_file = open(file_name, 'wb')
pickle.dump(results, pickle_file)
pickle_file.close()
```

```
# Read the saved pickle file and assign the object name of the class to the saved contents
def read_backtest_data(self, file_name='results.pkl'):
    pickle_file = open(file_name, 'rb')
    results = pickle.load(pickle_file)
    self.returns = results['returns']
    self.excess_returns = results['excess_returns']
    self.log_returns = results['log_returns']
    self.log_excess_returns = results['log_excess_returns']
    self.dates = results['dates']
    self.benchmark_returns = results['benchmark_returns']
    self.evaluations = results['evaluations']
    self.params_df = results['params_df']
    self.backtest_ids = results['backtest_ids']
    self.excess_max_drawdown = results['excess_max_drawdown']
    self.excess_annual_return = results['excess_annual_return']
    self.evaluations_df = results['evaluations_df']
    self.failed_list =  results['failed_list']
# A line graph of the rate of return
def plot_returns(self):
    # Specify the width and height of the drawing object in inches using the figsize parameter.
    fig = plt.figure(figsize=(20,8))
    ax = fig.add_subplot(111)
    # Charting
    for key in list(self.returns.keys()):
        ax.plot(list(range(len(self.returns[key]))), self.returns[key], label=key)
    # Setting the benchmark curve and labeling it
    ax.plot(list(range(len(self.benchmark_returns))), self.benchmark_returns, label='benchmark', c='k', linestyle='--')
    ticks = [int(x) for x in np.linspace(0, len(self.dates)-1, 11)]
    plt.xticks(ticks, [self.dates[i] for i in ticks])
    # set the legend style
    ax.legend(loc = 2, fontsize = 10)
    # Setting the y-tag style
    ax.set_ylabel('returns',fontsize=20)
    # set the x-tag style
    ax.set_yticklabels([str(x*100)+'% 'for x in ax.get_yticks()])
    # Setting the picture title style
    ax.set_title("Strategy's performances with different parameters", fontsize=21)
    plt.xlim(0, len(self.returns[0]))

# Excess yield charts
def plot_excess_returns(self):
    # Specify the width and height of the drawing object in inches using the figsize parameter.
    fig = plt.figure(figsize=(20,8))
```

```
        ax = fig.add_subplot(111)
        # Charting
        for key in list(self.returns.keys()):
            ax.plot(list(range(len(self.excess_returns[key]))), self.excess_returns[key], label=key)
        # Setting the benchmark curve and labeling it
        ax.plot(list(range(len(self.benchmark_returns))), [0]*len(self.benchmark_returns), label='benchmark', c='k', linestyle='--')
        ticks = [int(x) for x in np.linspace(0, len(self.dates)-1, 11)]
        plt.xticks(ticks, [self.dates[i] for i in ticks])
        # set the legend style
        ax.legend(loc = 2, fontsize = 10)
        # Setting the y-tag style
        ax.set_ylabel('excess returns',fontsize=20)
        # set the x-tag style
        ax.set_yticklabels([str(x*100)+'% 'for x in ax.get_yticks()])
        # Setting the picture title style
        ax.set_title("Strategy's performances with different parameters", fontsize=21)
        plt.xlim(0, len(self.excess_returns[0]))

    # log return chart
    def plot_log_returns(self):
        # Specify the width and height of the drawing object in inches using the figsize parameter.
        fig = plt.figure(figsize=(20,8))
        ax = fig.add_subplot(111)
        # Charting
        for key in list(self.returns.keys()):
            ax.plot(list(range(len(self.log_returns[key]))), self.log_returns[key], label=key)
        # Setting the benchmark curve and labeling it
        ax.plot(list(range(len(self.benchmark_returns))), [log(x+1) for x in self.benchmark_returns], label='benchmark', c='k',
linestyle='--')
        ticks = [int(x) for x in np.linspace(0, len(self.dates)-1, 11)]
        plt.xticks(ticks, [self.dates[i] for i in ticks])
        # set the legend style
        ax.legend(loc = 2, fontsize = 10)
        # Setting the y-tag style
        ax.set_ylabel('log returns',fontsize=20)
        # Setting the picture title style
        ax.set_title("Strategy's performances with different parameters", fontsize=21)
        plt.xlim(0, len(self.log_returns[0]))

    # A log plot of excess return
    def plot_log_excess_returns(self):
        # Specify the width and height of the drawing object in inches using the figsize parameter.
        fig = plt.figure(figsize=(20,8))
        ax = fig.add_subplot(111)
```

```
    # Charting
    for key in list(self.returns.keys()):
        ax.plot(list(range(len(self.log_excess_returns[key]))), self.log_excess_returns[key], label=key)
    # Setting the benchmark curve and labeling it
    ax.plot(list(range(len(self.benchmark_returns))), [0]*len(self.benchmark_returns), label='benchmark', c='k', linestyle='--')
    ticks = [int(x) for x in np.linspace(0, len(self.dates)-1, 11)]
    plt.xticks(ticks, [self.dates[i] for i in ticks])
    # set the legend style
    ax.legend(loc = 2, fontsize = 10)
    # Setting the y-tag style
    ax.set_ylabel('log excess returns',fontsize=20)
    # Setting the picture title style
    ax.set_title("Strategy's performances with different parameters", fontsize=21)
    plt.xlim(0, len(self.log_excess_returns[0]))


# 4 key metrics for backtesting, including total return, maximum retracement Sharpe ratio and volatility
def get_eval4_bar(self, sort_by=[]):

    sorted_params = self.params_df
    for by in sort_by:
        sorted_params = sorted_params.sort(by)
    indices = sorted_params.index
    indices = set(sorted_params.index)-set(self.failed_list)
    fig = plt.figure(figsize=(20,7))

    # Define the location
    ax1 = fig.add_subplot(221)
    # Setting the horizontal axis to the corresponding quantile and the vertical axis to the corresponding indicator
    ax1.bar(list(range(len(indices))),
            [self.evaluations[x]['algorithm_return'] for x in indices], 0.6, label = 'Algorithm_return')
    plt.xticks([x+0.3 for x in range(len(indices))], indices)
    # set the legend style
    ax1.legend(loc='best',fontsize=15)
    # Setting the y-tag style
    ax1.set_ylabel('Algorithm_return', fontsize=15)
    # Setting the y-tag style
    ax1.set_yticklabels([str(x*100)+'% 'for x in ax1.get_yticks()])
    # Setting the picture title style
    ax1.set_title("Strategy's of Algorithm_return performances of different quantile", fontsize=15)
    # x-axis range
    plt.xlim(0, len(indices))

    # Define the location
```

```
ax2 = fig.add_subplot(224)
# Setting the horizontal axis to the corresponding quantile and the vertical axis to the corresponding indicator
ax2.bar(list(range(len(indices))),
    [self.evaluations[x]['max_drawdown'] for x in indices], 0.6, label = 'Max_drawdown')
plt.xticks([x+0.3 for x in range(len(indices))], indices)
# set the legend style
ax2.legend(loc='best',fontsize=15)
# Setting the y-tag style
ax2.set_ylabel('Max_drawdown', fontsize=15)
# set the x-tag style
ax2.set_yticklabels([str(x*100)+'% 'for x in ax2.get_yticks()])
# Setting the picture title style
ax2.set_title("Strategy's of Max_drawdown performances of different quantile", fontsize=15)
# x-axis range
plt.xlim(0, len(indices))

# Define the location
ax3 = fig.add_subplot(223)
# Setting the horizontal axis to the corresponding quantile and the vertical axis to the corresponding indicator
ax3.bar(list(range(len(indices))),
    [self.evaluations[x]['sharpe'] for x in indices], 0.6, label = 'Sharpe')
plt.xticks([x+0.3 for x in range(len(indices))], indices)
# set the legend style
ax3.legend(loc='best',fontsize=15)
# Setting the y-tag style
ax3.set_ylabel('Sharpe', fontsize=15)
# set the x-tag style
ax3.set_yticklabels([str(x*100)+'% 'for x in ax3.get_yticks()])
# Setting the picture title style
ax3.set_title("Strategy's of Sharpe performances of different quantile", fontsize=15)
# x-axis range
plt.xlim(0, len(indices))

# Define the location
ax4 = fig.add_subplot(222)
# Setting the horizontal axis to the corresponding quantile and the vertical axis to the corresponding indicator
ax4.bar(list(range(len(indices))),
    [self.evaluations[x]['algorithm_volatility'] for x in indices], 0.6, label = 'Algorithm_volatility')
plt.xticks([x+0.3 for x in range(len(indices))], indices)
# set the legend style
ax4.legend(loc='best',fontsize=15)
# Setting the y-tag style
ax4.set_ylabel('Algorithm_volatility', fontsize=15)
# set the x-tag style
```

```
    ax4.set_yticklabels([str(x*100)+'% 'for x in ax4.get_yticks()])
    # Setting the picture title style
    ax4.set_title("Strategy's of Algorithm_volatility performances of different quantile", fontsize=15)
    # x-axis range
    plt.xlim(0, len(indices))

#14 Annualized returns and maximum retracements, plus or minus two-color representation
def get_eval(self, sort_by=[]):

    sorted_params = self.params_df
    for by in sort_by:
        sorted_params = sorted_params.sort(by)
    indices = sorted_params.index
    indices = set(sorted_params.index)-set(self.failed_list)
    # size
    fig = plt.figure(figsize = (20, 8))
    # Figure 1 location
    ax = fig.add_subplot(111)
    # Maximum retracement for generating graph excess returns
    ax.bar([x+0.3 for x in range(len(indices))],
           [-self.evaluations[x]['max_drawdown'] for x in indices], color = '#32CD32',
               width = 0.6, label = 'Max_drawdown', zorder=10)
    # Charting annualized excess returns
    ax.bar([x for x in range(len(indices))],
           [self.evaluations[x]['annual_algo_return'] for x in indices], color = 'r',
               width = 0.6, label = 'Annual_return')
    plt.xticks([x+0.3 for x in range(len(indices))], indices)
    # set the legend style
    ax.legend(loc='best',fontsize=15)
    # Baseline
    plt.plot([0, len(indices)], [0, 0], c='k',
             linestyle='--', label='zero')
    # set the legend style
    ax.legend(loc='best',fontsize=15)
    # Setting the y-tag style
    ax.set_ylabel('Max_drawdown', fontsize=15)
    # set the x-tag style
    ax.set_yticklabels([str(x*100)+'% 'for x in ax.get_yticks()])
    # Setting the picture title style
    ax.set_title("Strategy's performances of different quantile", fontsize=15)
    # Set the length of the x-axis
    plt.xlim(0, len(indices))
```

```
    #14 Annualized returns and maximum retracements of excess returns
    # After joining the new benchmark excess returns and
    def get_excess_eval(self, sort_by=[]):

        sorted_params = self.params_df
        for by in sort_by:
            sorted_params = sorted_params.sort(by)
        indices = sorted_params.index
        indices = set(sorted_params.index)-set(self.failed_list)
        # size
        fig = plt.figure(figsize = (20, 8))
        # Figure 1 location
        ax = fig.add_subplot(111)
        # Maximum retracement for generating graph excess returns
        ax.bar([x+0.3 for x in range(len(indices))],
               [-self.excess_max_drawdown[x] for x in indices], color = '#32CD32',
                   width = 0.6, label = 'Excess_max_drawdown')
        # Charting annualized excess returns
        ax.bar([x for x in range(len(indices))],
               [self.excess_annual_return[x] for x in indices], color = 'r',
                   width = 0.6, label = 'Excess_annual_return')
        plt.xticks([x+0.3 for x in range(len(indices))], indices)
        # set the legend style
        ax.legend(loc='best',fontsize=15)
        # Baseline
        plt.plot([0, len(indices)], [0, 0], c='k',
                 linestyle='--', label='zero')
        # set the legend style
        ax.legend(loc='best',fontsize=15)
        # Setting the y-tag style
        ax.set_ylabel('Max_drawdown', fontsize=15)
        # set the x-tag style
        ax.set_yticklabels([str(x*100)+'% 'for x in ax.get_yticks()])
        # Setting the picture title style
        ax.set_title("Strategy's performances of different quantile", fontsize=15)
        # Set the length of the x-axis
        plt.xlim(0, len(indices))

#2 Set the policy id for backtesting
pa = parameter_analysis('2e90ddc1230314ee5e4a00675c55ecd3')
#3 Run backtests
pa.get_backtest_data(file_name = 'results.pkl', # name of the Pickle file that holds the results of the backtest
                     running_max = 2, # Maximum number of simultaneous backtests, can be redeemed through the Points Store
                     benchmark_id = None, # the benchmark's backtest ID, note that it is the backtest ID and not the strategy ID, and
```

when it is None, it is the benchmark used in the strategy

start_date = '2016-02-01', # backtest start time, the

end_date = '2018-11-01', #Backtest end date that

frequency = 'day', #Test frequency, supports day, minute, tick

initial_cash = '2000000', # initial_cash

param_names = ['factor', 'quantile'], #variable_names

param_values = [['BP'], tuple(zip(range(0,100,10), range(10,101,10)))], #variable corresponding to parameter

python_version = 3, # backtest python version

use_credit = True # Whether or not to allow the use of credit to continue the test when the available time is insufficient, insufficient credit will lead to an error, please pay attention to the credit balance, please use_credit = True # If the available time is insufficient, please allow the use of credit to continue the test

)

2, the way the factors are assigned

```
#Step 1: Obtain factorization data
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from jqdata import *

# Getting BP factor data
def get_bp_factor_data(stock_list, start_date, end_date):
    dates = pd.date_range(start=start_date, end=end_date, freq='M') # the last working day of the month as an observation point
    bp_data = pd.DataFrame()

    for date in dates:
        q = query(valuation.code, valuation.pb_ratio).filter(valuation.code.in_(stock_list))
        df = get_fundamentals(q, date=date)
        df['BP'] = 1 / df['pb_ratio']
        bp_data[date.strftime('%Y-%m')] = df.set_index('code')['BP']

    return bp_data

# Obtaining data on non-weighted price factors
def get_price_no_fq_factor_data(stock_list, start_date, end_date):
    dates = pd.date_range(start=start_date, end=end_date, freq='M')
    price_no_fq_data = pd.DataFrame()
```

```
for date in dates:
    price_data = get_price(stock_list, end_date=date, count=1, fields=['close'])['close'].T
    price_no_fq_data[date.strftime('%Y-%m')] = price_data.iloc[:, 0]

return price_no_fq_data

import seaborn as sns

def plot_factor_distribution_over_time(factor_data, title):
    plt.figure(figsize=(14, 8))
    for date in factor_data.columns:
        #sns.kdeplot(factor_data[date].dropna(), label=date, fill=True)
        sns.kdeplot(factor_data[date].dropna(), label=date) # code for more compatibility
    plt.title(f'{title} Distribution Over Time')
    plt.xlabel(f'{title} Value')
    plt.ylabel('Density')
    plt.legend(title='Date', bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.tight_layout()
    plt.show()

#Step 3: Implementation analysis
# Define the pool of stocks to be analyzed and the time horizon
stock_list = get_index_stocks('000906.XSHG') # CSI 800 Index
start_date = '2023-01-01'
end_date = '2024-01-01'

# Getting factorized data
bp_factor_data = get_bp_factor_data(stock_list, start_date, end_date)
price_no_fq_factor_data = get_price_no_fq_factor_data(stock_list, start_date, end_date)

# Visualization factor distribution
plot_factor_distribution_over_time(bp_factor_data, 'BP')
plot_factor_distribution_over_time(price_no_fq_factor_data, 'Price No FQ')
```