Strategy_and_Unility

December 4, 2023

[1]:
```
import warnings
import math

import numpy as np
import pandas as pd
import datetime as dt
from datetime import date

import plotly.express as px

from tqdm import tqdm

warnings.filterwarnings('ignore')

from concurrent.futures import ThreadPoolExecutor

import statistics
```

1

, , : ,
.

1.1

[2]:
```
class MyPosition:

    def __init__(self, amount: float, price_current: float, short: bool) ->␣
 ↪None:
        """
                    MyPosition.

        :param amount:          a.
        :param price_current:             .
        :param acc_fees:                   .
        :param short:              (True,        ; False,        ).
        """
                    self._amount: float = amount
                    self._price_current: float = price_current
                    self._acc_fees: float = 0
                    self._short: bool = short

    def update_state(self, price: float) -> None:
        """
                        .


        :param price:            .
        """
                    self._price_current = price
                    if self._short:
                        #
                        transfer_fee = abs(self._amount) * self._price_current * 0.00065
                        self._acc_fees += transfer_fee

    def balance(self) -> float:
        """
                       .


        :return:           .
        """
                    return self._amount * self._price_current
```

: ### _position: ( ). _states: (, ,). _equity: . _margin_equity: . ### run . , , , . ### calc_upper_and_lower: . open_short: . open_long: . close_short: . close_long: . ### RISK: .

STD_COUNT_UP, STD_COUNT_DOWN: .

MA_COUNT: . ### , , , , , / .

[3]:
```
class Strategy:

    params = {'FEE': 0.0004}

    def __init__(self, states, start_equity):
        self._position = None
        self._states = states
        self._equity = start_equity
        self._margin_equity = 0

    def run(self, RISK, STD_COUNT_UP, STD_COUNT_DOWN, MA_COUNT):
        states_ma = self._states['pos'].rolling(window=MA_COUNT).mean()
        data = []

        for i in tqdm(range(len(self._states)), disable=True):
            if i < 10 * MA_COUNT:
                continue
            elif (i == 10 * MA_COUNT):
                state = self._states.loc[i]
                pos_ma = states_ma[i]
                prev_pos_ma = states_ma[i - 1]
                (pos_ma_upper, pos_ma_lower) = self.calc_upper_and_lower(i,␣
↪states_ma, STD_COUNT_UP, STD_COUNT_DOWN)
            else:
                state = self._states.loc[i]
                pos_ma = states_ma[i]
                prev_pos_ma = states_ma[i - 1]

            b = 0
            s = 0

            if self._position:
                if state['price'] != self._states['price'][i - 1]:
                    self._position.update_state(state['price'])

                #if prev_pos_ma > pos_ma_lower and pos_ma < pos_ma_lower and␣
↪self._position._short:
                if prev_pos_ma < pos_ma_upper and pos_ma > pos_ma_upper:
                    pos_ma_upper, pos_ma_lower = self.calc_upper_and_lower(i,␣
↪states_ma, STD_COUNT_UP, STD_COUNT_DOWN)
                    if self._position._short:
                        b = 1
                #if prev_pos_ma < pos_ma_upper and pos_ma > pos_ma_upper and␣
↪not self._position._short:
                if prev_pos_ma > pos_ma_lower and pos_ma < pos_ma_lower:
                    pos_ma_upper, pos_ma_lower = self.calc_upper_and_lower(i,␣
↪states_ma, STD_COUNT_UP, STD_COUNT_DOWN)
                    if not self._position._short:
                        s = 1

                data.append([
                    state['datetime'],
                    state['price'],
                    self._equity - self._position._acc_fees,
                    self._position._short,
                    pos_ma_upper,
                    pos_ma_lower,
                    pos_ma,
                    b,
                    s
                ])

                if b:
                    self.close_short()
                if s:
                    self.close_long()
            else:
                #if prev_pos_ma < pos_ma_lower and pos_ma > pos_ma_lower:
                if prev_pos_ma < pos_ma_upper and pos_ma > pos_ma_upper:
                    pos_ma_upper, pos_ma_lower = self.calc_upper_and_lower(i, states_ma, STD_COUNT_UP, STD_COUNT_DOWN)
                    self.open_long(state, RISK)
                    b = 1
                #if prev_pos_ma > pos_ma_upper and pos_ma < pos_ma_upper:
                if prev_pos_ma > pos_ma_lower and pos_ma < pos_ma_lower:
                    pos_ma_upper, pos_ma_lower = self.calc_upper_and_lower(i, states_ma, STD_COUNT_UP, STD_COUNT_DOWN)
                    self.open_short(state, RISK)
                    s = 1

                data.append([
                    state['datetime'],
                    state['price'],
                    self._equity,
                    0,
                    pos_ma_upper,
                    pos_ma_lower,
                    pos_ma,
                    b,
                    s
                ])

        return pd.DataFrame(data, columns=['datetime', 'price', 'equity', 'short', 'pos_ma_upper', 'pos_ma_lower', 'pos_ma', 'buy', 'sell'])

    def calc_upper_and_lower(self, i, states_ma, STD_COUNT_UP, STD_COUNT_DOWN):
        #mean_pos_ma = states_ma.head(i).mean()
        std_pos_ma = states_ma.head(i).std()
        pos_ma_upper = states_ma[i] + STD_COUNT_UP * std_pos_ma
        pos_ma_lower = states_ma[i] - STD_COUNT_DOWN * std_pos_ma
        return pos_ma_upper, pos_ma_lower

    def open_short(self, state, RISK):
        if self._position:
            raise Exception(f'Cannot open position, already have one {self.
↪_position}')

        self._margin_equity += (np.floor((1 - self.params['FEE']) * RISK * self.
↪_equity / state['price']) * state['price'] - self.params['FEE'] * self.
↪_equity)

        amount = (-1) * np.floor((1 - self.params['FEE']) * RISK * self._equity␣
↪/ state['price'])

        self._position = MyPosition(amount, state['price'], True)

    def open_long(self, state, RISK):
        if self._position:
            raise Exception(f'Cannot open position, already have one {self.
↪_position}')

        self._margin_equity -= (np.floor((1 - self.params['FEE']) * RISK * self.
↪_equity / state['price']) * state['price'] + self.params['FEE'] * RISK *␣
↪self._equity)

        amount = np.floor((1 - self.params['FEE']) * RISK * self._equity /␣
↪state['price'])

        self._position = MyPosition(amount, state['price'], False)

    def close_short(self):
        if self._position._short == 0:
            raise Exception(f'Cannot close short position, it is long')

        self._equity += (
                self._margin_equity + (1 - self.params['FEE']) * self._position.
↪balance() - self._position._acc_fees)
        self._margin_equity = 0
        self._position = None

    def close_long(self):
        if self._position._short == 1:
            raise Exception(f'Cannot close long position, it is short')

        self._equity += (self._margin_equity + (1 - self.params['FEE']) * self.
↪_position.balance() - self._position._acc_fees)
        self._margin_equity = 0
        self._position = None
```

[4]:

```
def run_strategy(all_data_copy, std_count_up, std_count_down, ma_count):
    strategy = Strategy(all_data_copy, 10000000)
    df = pd.DataFrame(strategy.run(1, std_count_up, std_count_down, ma_count))
    df['equity'] = df['equity'] / df['equity'].iloc[0]
    final_total_balance = df['equity'].iloc[-1]
    return final_total_balance

#

def train(all_data_copy1: pd.DataFrame) -> (float, float, float):
    #
    std_counts_up = np.arange(0.25, 2, 0.25)
    std_counts_down = np.arange(0.25, 2, 0.25)
    ma_counts = np.arange(500, 2500, 500)

    radius = 2

    parameters = []
    final_total_balances = []

    with ThreadPoolExecutor(max_workers=4) as executor:
        futures = []

        for std_count_up in std_counts_up:
            for std_count_down in std_counts_down:
                for ma_count in ma_counts:
                    #print(std_count_up, std_count_down, ma_count)
                    future = executor.submit(run_strategy, all_data_copy1,␣
↪std_count_up, std_count_down, ma_count)
                    futures.append((std_count_up, std_count_down, ma_count,␣
↪future))

        for std_count_up, std_count_down, ma_count, future in futures:
            final_total_balance = future.result()
            parameters.append((std_count_up, std_count_down, ma_count))
            final_total_balances.append(final_total_balance)

    data_parametrs = pd.DataFrame(parameters, columns=['std_count_up',␣
↪'std_count_down', 'ma_count'])
    data_parametrs['final_total_balance'] = final_total_balances

    array_utility = []
    for index_std_count_up in range(len(std_counts_up)):
        for index_std_count_down in range(len(std_counts_down)):
            for index_ma_count in range(len(ma_counts)):
```

```
            array_utility.append(Utility_assess(data_parametrs,␣
↪index_std_count_up, index_std_count_down, index_ma_count,␣
↪len(std_counts_up), len(std_counts_down), len(ma_counts), radius))

max_utility = max(array_utility)
max_utility_index = array_utility.index(max_utility)
corresponding_parameters = parameters[max_utility_index]

#
std_counts_up = np.arange(corresponding_parameters[0] - 0.1,␣
↪corresponding_parameters[0] + 0.2, 0.1)
std_counts_down = np.arange(corresponding_parameters[1] - 0.1,␣
↪corresponding_parameters[1] + 0.2, 0.1)
ma_counts = np.arange(corresponding_parameters[2] - 250,␣
↪corresponding_parameters[2] + 500, 250)

parameters = []
final_total_balances = []

with ThreadPoolExecutor(max_workers=4) as executor:
    futures = []

    for std_count_up in std_counts_up:
        for std_count_down in std_counts_down:
            for ma_count in ma_counts:
                future = executor.submit(run_strategy, all_data_copy1,␣
↪std_count_up, std_count_down, ma_count)
                futures.append((std_count_up, std_count_down, ma_count,␣
↪future))

    for std_count_up, std_count_down, ma_count, future in futures:
        final_total_balance = future.result()
        parameters.append((std_count_up, std_count_down, ma_count))
        final_total_balances.append(final_total_balance)

data_parametrs = pd.DataFrame(parameters, columns=['std_count_up',␣
↪'std_count_down', 'ma_count'])
data_parametrs['final_total_balance'] = final_total_balances

array_utility = []
for index_std_count_up in range(len(std_counts_up)):
    for index_std_count_down in range(len(std_counts_down)):
        for index_ma_count in range(len(ma_counts)):
            array_utility.append(Utility_assess(data_parametrs,␣
↪index_std_count_up, index_std_count_down, index_ma_count,␣
↪len(std_counts_up), len(std_counts_down), len(ma_counts), radius))
```

```
    max_utility = max(array_utility)
    max_utility_index = array_utility.index(max_utility)
    corresponding_parameters = parameters[max_utility_index]
    corresponding_final_total_balance = final_total_balances[max_utility_index]

    print(f"Maximum Total Balance: {corresponding_final_total_balance}")
    print(f"Maximum Utility: {corresponding_final_total_balance}")
    print(f"Corresponding Parameters (std_count, std_timerange):␣
 ↪{corresponding_parameters}")
    return corresponding_parameters

#

def test(all_data_copy2: pd.DataFrame, std_count_up: float, std_count_down:␣
 ↪float, ma_count: float) -> float:
    strategy = Strategy(all_data_copy2, 10000000)
    df = pd.DataFrame(strategy.run(1, std_count_up, std_count_down, ma_count))
    df['equity'] = df['equity'] / df['equity'].iloc[0]
    final_total_balance = df['equity'].iloc[-1]

    print(f'Test profit = {final_total_balance}')
    return final_total_balance
```

(,)

```
[5]: def clear_data(tiker):
    # Pos

    df = pd.read_csv(f'{tiker}_full_date.csv', sep=",")
    df = df.sort_values(by=['ticker', 'tradedate', 'tradetime']).
 ↪drop_duplicates().reset_index().drop('index', axis=1)
    df['datetime'] = pd.to_datetime(df['tradedate'] + ' ' + df['tradetime'])
    df = df.drop(['tradedate', 'tradetime'], axis=1)
    df = df[df['clgroup'] == 'YUR'].reset_index().drop('index', axis=1)
    df = df[['datetime', 'pos']]
    data_pos = df

    # Price

    df = pd.read_csv(f'{tiker}_full_date_price.csv', sep=",")
    df.reset_index(inplace=True)
    df.rename(columns={'TRADEDATE': 'datetime'}, inplace=True)
    df['datetime'] = pd.to_datetime(df['datetime'])
    df = df[df['BOARDID']=='TQBR']
    df = df[['datetime', 'WAPRICE']]
    df.rename(columns={'WAPRICE': 'price'}, inplace=True)
    data_price = df

    # All data

    price_dict = dict(zip(data_price['datetime'].dt.date, data_price['price']))

    data_pos['price'] = data_pos['datetime'].dt.date.map(price_dict)

    all_data = data_pos.sort_values('datetime').drop_duplicates().dropna().
 ↪reset_index().drop('index', axis=1)

    all_data_copy = all_data.copy()

    all_data_copy['date'] = all_data_copy['datetime'].dt.date

    # For train

    train_data = all_data_copy[all_data_copy['date'] < pd.
 ↪to_datetime('2022-02-20').date()].reset_index().drop('index', axis=1)

    # For test

    test_data = all_data_copy[all_data_copy['date'] > pd.
 ↪to_datetime('2022-12-31').date()].reset_index().drop('index', axis=1)

    return train_data, test_data
```

1.2

, 2,

125 $((2 * 2 + 1)^3)$.

[6]:
```
def Utility_fun(income, var):
    return income - 10*var

def Utility_assess(df_plotly, index_std_count_up, index_std_count_down,␣
 ↪index_ma_count, len_std_counts_up, len_std_counts_down, len_ma_counts,␣
 ↪radius):
    #        iloc
    income = df_plotly.loc[
        (df_plotly['std_count_up'] == df_plotly['std_count_up'].
 ↪iloc[index_std_count_up]) &
        (df_plotly['std_count_down'] == df_plotly['std_count_down'].
 ↪iloc[index_std_count_down]) &
        (df_plotly['ma_count'] == df_plotly['ma_count'].iloc[index_ma_count])
    ]['final_total_balance'].values[0]
    index_std_count_up -= radius
    index_std_count_down -= radius
    index_ma_count -= radius

    array_income = np.array([])
    for i in range(2 * radius + 1):
        for j in range(2 * radius + 1):
            for k in range(2 * radius + 1):
                if (
                    (index_std_count_up + i >= 0) and
                    (index_std_count_down + j >= 0) and
                    (index_ma_count + k >= 0) and
                    (index_std_count_up + i < len_std_counts_up) and
                    (index_std_count_down + j < len_std_counts_down) and
                    (index_ma_count + k < len_ma_counts)
                ):
                    array_income = np.append(
                        array_income,
                        df_plotly.loc[
                            (df_plotly['std_count_up'] ==␣
 ↪df_plotly['std_count_up'].iloc[index_std_count_up + i]) &
                            (df_plotly['std_count_down'] ==␣
 ↪df_plotly['std_count_down'].iloc[index_std_count_down + j]) &
                            (df_plotly['ma_count'] == df_plotly['ma_count'].
 ↪iloc[index_ma_count + k])
                        ]['final_total_balance'].values[0]
                    )

    std_of_income = statistics.stdev(array_income) if len(array_income) >= 2␣
 ↪else 0
    Utility = Utility_fun(income, std_of_income)
    return Utility
```

[7]:

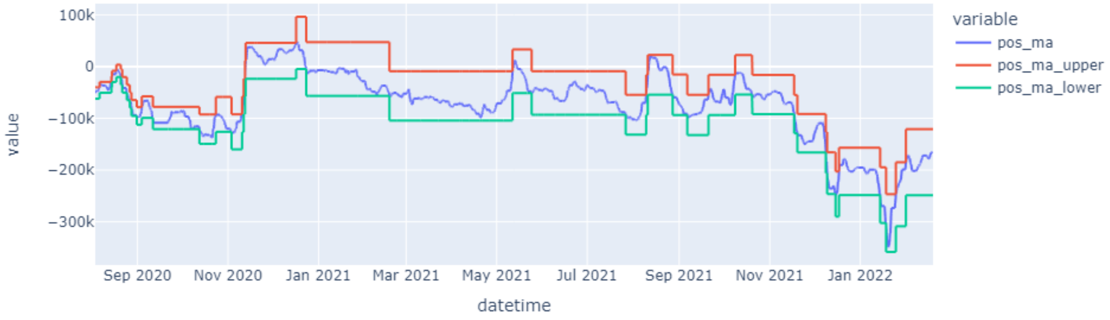
```
ticker_names=['sr', 'gz', 'lk', 'vb', 'rn', 'mn', 'af', 'al', 'sn', 'yn', 'tt',␣
 ↪'nm', 'hy', 'me', 'fv', 'gk', 'mg']

for ticker in ticker_names:

    train_data, test_data = clear_data(ticker)

    corresponding_parameters = []

    corresponding_parameters = train(train_data)
    result = test(test_data, corresponding_parameters[0],␣
 ↪corresponding_parameters[1], corresponding_parameters[2])
```

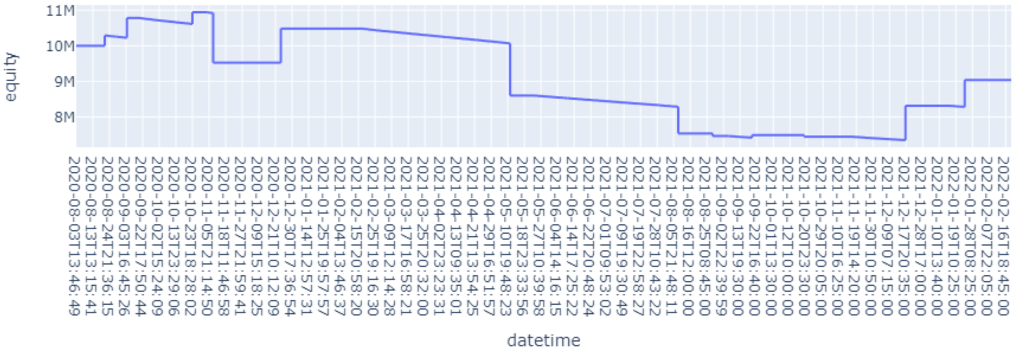
```
Maximum Total Balance: 0.6981924818628379
Maximum Utility: 0.6981924818628379
Corresponding Parameters (std_count, std_timerange): (0.65, 0.4, 750)
Test profit = 1.0633945176255282
Maximum Total Balance: 0.8612479478991492
Maximum Utility: 0.8612479478991492
Corresponding Parameters (std_count, std_timerange): (0.65, 1.75, 1750)
Test profit = 0.97207110060225
Maximum Total Balance: 0.6845689499363472
Maximum Utility: 0.6845689499363472
Corresponding Parameters (std_count, std_timerange): (0.65, 0.4, 750)
Test profit = 0.5946274646464171
Maximum Total Balance: 1.2474591283396346
Maximum Utility: 1.2474591283396346
Corresponding Parameters (std_count, std_timerange): (0.65, 1.65, 500)
Test profit = 1.271159448233706
Maximum Total Balance: 1.0763171320146696
Maximum Utility: 1.0763171320146696
Corresponding Parameters (std_count, std_timerange): (0.65, 0.4, 1500)
Test profit = 0.88623080875
Maximum Total Balance: 0.9064933091906588
Maximum Utility: 0.9064933091906588
Corresponding Parameters (std_count, std_timerange): (0.65, 0.4, 1000)
Test profit = 0.6894152605232093
Maximum Total Balance: 1.0
Maximum Utility: 1.0
Corresponding Parameters (std_count, std_timerange): (0.65, 1.9500000000000002,
1500)
Test profit = 1.1046274965023999
Maximum Total Balance: 0.9736834164028
Maximum Utility: 0.9736834164028
Corresponding Parameters (std_count, std_timerange): (0.65, 0.4, 2250)
Test profit = 0.8824816184874605
Maximum Total Balance: 1.0
Maximum Utility: 1.0
Corresponding Parameters (std_count, std_timerange): (0.65, 0.4, 1750)
Test profit = 1.1592918476134304
Maximum Total Balance: 1.0768127519197561
Maximum Utility: 1.0768127519197561
Corresponding Parameters (std_count, std_timerange): (0.65, 1.65, 250)
Test profit = 1.2163163318073906
Maximum Total Balance: 1.0330253413939434
Maximum Utility: 1.0330253413939434
Corresponding Parameters (std_count, std_timerange): (0.65, 0.4, 500)
Test profit = 0.6067048152710467
Maximum Total Balance: 1.0
Maximum Utility: 1.0
Corresponding Parameters (std_count, std_timerange): (0.65, 1.65, 1750)
```

```
Test profit = 1.3036761745830001
Maximum Total Balance: 1.0
Maximum Utility: 1.0
Corresponding Parameters (std_count, std_timerange): (0.65, 1.65, 1750)
Test profit = 1.2218503280421078
Maximum Total Balance: 0.8619265778361667
Maximum Utility: 0.8619265778361667
Corresponding Parameters (std_count, std_timerange): (0.65, 0.4, 750)
Test profit = 0.7269213448394016
Maximum Total Balance: 0.9269192867212758
Maximum Utility: 0.9269192867212758
Corresponding Parameters (std_count, std_timerange): (0.65, 0.4, 1750)
Test profit = 0.8248412285273055
Maximum Total Balance: 1.0
Maximum Utility: 1.0
Corresponding Parameters (std_count, std_timerange): (0.65, 0.4, 1750)
Test profit = 0.7977171618655284
Maximum Total Balance: 1.0
Maximum Utility: 1.0
Corresponding Parameters (std_count, std_timerange): (0.65, 1.65, 1750)
Test profit = 1.2477307323448001
```

[8]:

```
train_data, test_data = clear_data('sr')

strategy = Strategy(train_data, 10000000)
df = pd.DataFrame(strategy.run(1, 1, 1, 500))

px.line(df, x='datetime', y=['pos_ma', 'pos_ma_upper', 'pos_ma_lower']).show()

px.line(df, x='datetime', y='equity').update_xaxes(type='category').show()

up = pd.DataFrame(columns=['datetime', 'price'])
down = pd.DataFrame(columns=['datetime', 'price'])
for i in range(len(df['price'])):
    if df['buy'][i]:
        up.loc[len(up)] = df.iloc[i]
    elif df['sell'][i]:
        down.loc[len(down)] = df.iloc[i]
fig = px.line(df, x='datetime', y='price', title='      ',␣
 ↪labels={'datetime': '    ', 'price': ' '})
fig.add_trace(px.scatter(up, x='datetime', y='price').
 ↪update_traces(marker=dict(color='red')).data[0])
fig.add_trace(px.scatter(down, x='datetime', y='price').
 ↪update_traces(marker=dict(color='green')).data[0])
fig.update_xaxes(type='category')
fig.show()
```


variable
pos_ma
pos_ma_upper
pos_ma_lower
value
100k
0
−100k
−200k
−300k
Sep 2020
Nov 2020
Jan 2021
Mar 2021
May 2021
Jul 2021
Sep 2021
Nov 2021
Jan 2022
datetime


equity
11M
10M
9M
8M
2020-08-03T13:46:49
2020-08-13T13:15:41
2020-08-24T21:36:15
2020-09-03T16:45:26
2020-09-22T17:50:44
2020-10-02T15:24:09
2020-10-13T23:29:06
2020-10-23T18:28:02
2020-11-05T21:14:50
2020-11-18T11:46:58
2020-11-27T21:59:41
2020-12-09T15:18:25
2020-12-21T10:12:09
2020-12-30T17:36:54
2021-01-14T12:57:31
2021-01-25T19:57:57
2021-02-04T13:46:37
2021-02-15T20:58:20
2021-02-25T19:16:30
2021-03-09T12:14:28
2021-03-17T16:58:21
2021-03-25T20:32:00
2021-04-02T23:23:31
2021-04-13T09:35:01
2021-04-21T13:54:25
2021-04-29T16:51:57
2021-05-10T19:48:23
2021-05-18T23:33:56
2021-05-27T10:39:58
2021-06-04T14:16:15
2021-06-14T17:25:22
2021-06-22T20:48:24
2021-07-01T09:53:02
2021-07-09T19:30:49
2021-07-19T22:58:27
2021-07-28T10:43:22
2021-08-05T21:48:11
2021-08-16T12:00:00
2021-08-25T08:45:00
2021-09-02T22:39:59
2021-09-13T19:30:00
2021-09-22T16:40:00
2021-10-01T13:30:00
2021-10-12T10:30:00
2021-10-20T23:30:00
2021-10-29T20:05:00
2021-11-10T16:25:00
2021-11-19T14:20:00
2021-11-30T10:50:00
2021-12-09T07:15:00
2021-12-17T20:35:00
2021-12-28T17:00:00
2022-01-10T13:40:00
2022-01-19T10:25:00
2022-01-28T08:25:00
2022-02-07T22:05:00
2022-02-16T18:45:00
datetime

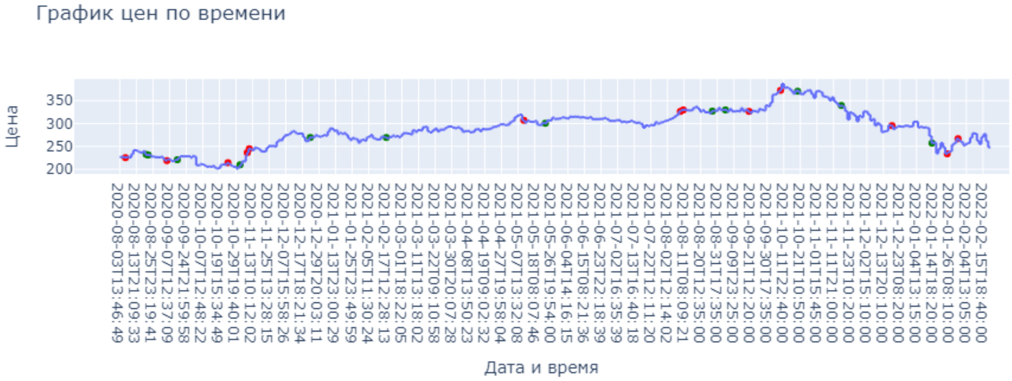


График цен по времени
Цена
350
300
250
200
2020-08-03T13:46:49
2020-08-13T21:09:33
2020-08-25T23:19:41
2020-09-07T12:37:09
2020-09-24T21:59:58
2020-10-07T12:48:22
2020-10-19T15:34:49
2020-10-29T19:40:01
2020-11-13T10:12:02
2020-11-25T13:28:15
2020-12-07T15:58:26
2020-12-17T18:21:34
2020-12-29T20:03:11
2021-01-13T23:00:29
2021-01-25T23:49:59
2021-02-05T11:30:24
2021-02-17T12:28:13
2021-03-01T18:22:05
2021-03-11T13:18:02
2021-03-22T09:10:58
2021-03-30T20:07:28
2021-04-08T13:50:23
2021-04-19T09:02:32
2021-04-27T19:58:04
2021-05-07T13:32:08
2021-05-18T08:07:46
2021-05-26T19:54:00
2021-06-04T14:16:15
2021-06-15T08:21:36
2021-06-23T22:18:39
2021-07-02T16:35:39
2021-07-13T16:40:18
2021-07-22T12:11:20
2021-08-02T12:14:02
2021-08-11T08:09:21
2021-08-20T12:35:00
2021-08-31T17:35:00
2021-09-09T23:25:00
2021-09-21T12:20:00
2021-09-30T17:35:00
2021-10-11T22:40:00
2021-10-21T10:50:00
2021-11-01T15:45:00
2021-11-11T21:00:00
2021-11-23T10:20:00
2021-12-02T15:10:00
2021-12-13T20:10:00
2021-12-23T08:20:00
2022-01-04T13:15:00
2022-01-14T18:20:00
2022-01-26T08:10:00
2022-02-04T13:05:00
2022-02-15T18:40:00
Дата и время

1.3

, 2,

125 $((2 * 2 + 1)^3)$.

[9]:

```
def Utility_fun(income, var):
    return income - 10*var
```

[18]:

```
import plotly.graph_objs as go
from plotly.subplots import make_subplots

step_std_counts_up = 0.15
step_std_counts_down = 0.15
step_ma_counts = 700
std_counts_up = np.arange(0.15, 1, step_std_counts_up)
std_counts_down = np.arange(0.15, 1, step_std_counts_down)
ma_counts = np.arange(500, 5000, step_ma_counts)

parameters = []
final_total_balances = []

with ThreadPoolExecutor(max_workers=4) as executor:
    futures = []

    for std_count_up in std_counts_up:
        for std_count_down in std_counts_down:
            for ma_count in ma_counts:
                future = executor.submit(run_strategy, train_data,␣
 ↪std_count_up, std_count_down, ma_count)
                futures.append((std_count_up, std_count_down, ma_count, future))

    for std_count_up, std_count_down, ma_count, future in futures:
        final_total_balance = future.result()
        parameters.append((std_count_up, std_count_down, ma_count))
        final_total_balances.append(final_total_balance)

#            DataFrame
df_plotly = pd.DataFrame(parameters, columns=['std_count_up', 'std_count_down',␣
 ↪'ma_count'])
df_plotly['final_total_balance'] = final_total_balances
print(df_plotly)


#                    Plotly
fig = make_subplots(rows=1, cols=1, specs=[[{'type': 'scatter3d'}]])

scatter = go.Scatter3d(
    x=df_plotly['std_count_up'],
    y=df_plotly['std_count_down'],
    z=df_plotly['ma_count'],
    mode='markers',
    marker=dict(
        size=8,
        color=df_plotly['final_total_balance'],
        colorscale='Viridis',
        colorbar=dict(title='Final Total Balance')
    )
)

fig.add_trace(scatter)

#
fig.update_layout(scene=dict(
                    xaxis_title='std_count_up',
                    yaxis_title='std_count_down',
                    zaxis_title='ma_count')
                  )

#
fig.show()
```

```
     std_count_up  std_count_down  ma_count  final_total_balance
0            0.15            0.15       500             0.905158
1            0.15            0.15      1200             0.744273
2            0.15            0.15      1900             0.786764
3            0.15            0.15      2600             0.826889
4            0.15            0.15      3300             1.063862
..            …              …        …                  …
247          0.90            0.90      1900             0.770576
248          0.90            0.90      2600             0.839360
249          0.90            0.90      3300             1.125710
250          0.90            0.90      4000             0.818123
251          0.90            0.90      4700             0.925350

[252 rows x 4 columns]
```

[14]:

```
radius = 2

array_utility = []
for index_std_count_up in range(len(std_counts_up)):
    for index_std_count_down in range(len(std_counts_down)):
        for index_ma_count in range(len(ma_counts)):
            array_utility.append(Utility_assess(df_plotly, index_std_count_up,␣
 ↪index_std_count_down, index_ma_count, len(std_counts_up),␣
 ↪len(std_counts_down), len(ma_counts), radius))
```

```
df_plotly['final_utility_balance'] = array_utility
print(np.max(df_plotly['final_total_balance']), np.max(array_utility))
```

```
1.1408370962580399 0.283375328345296
```

[17]:
```
import plotly.graph_objs as go
from plotly.subplots import make_subplots

#                   Plotly
fig = make_subplots(rows=1, cols=1, specs=[[{'type': 'scatter3d'}]])

scatter = go.Scatter3d(
    x=df_plotly['std_count_up'],
    y=df_plotly['std_count_down'],
    z=df_plotly['ma_count'],
    mode='markers',
    marker=dict(
        size=4,
        color=df_plotly['final_utility_balance'],
        colorscale='Viridis',
        colorbar=dict(title='Final Total Balance')
    ),
    text=df_plotly.apply(lambda row: f' final_utility_balance:␣
  ↪{row["final_utility_balance"]},  final_total_balance:␣
  ↪{row["final_total_balance"]}', axis=1)
)

fig.add_trace(scatter)

#
fig.update_layout(scene=dict(
                    xaxis_title='std_count_up',
                    yaxis_title='std_count_down',
                    zaxis_title='ma_count')
                 )

#
fig.show()
```

[ ]:
```

```