ProSpace Assignment Quantitative Trading Report

Akshat Thakur, 22B2110

June 10, 2024

Contents

1 Description

The assignment required us to create a backtesting engine, following a class-based implementation. This is a comprehensive report of the same, detailing my thought process throughout the project as well as explaining the code at places.
The dataset contained two columns: 'price' and 'alpha'. We were required to **find the optimum static liquidate and build thresholds** to **maximize PnL** at the end of all trades. No null values were detected, negating the need to employ data-cleaning methods.

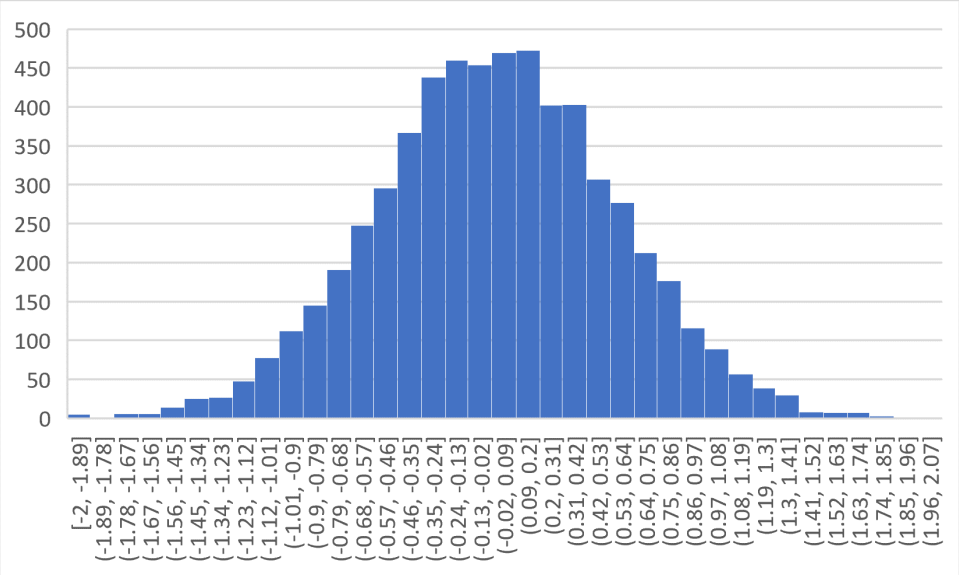

Figure 1: Distribution of Alpha values

As can be seen, most of the alpha values lie between ± 0.75, hinting towards a **possible upper bound** for the build threshold. The final values will be determined later through a brute-force algorithm.

2 Making the Strategy Class

As mentioned above, a class-based implementation of the strategy and backtesting engine was followed. The following methods (apart from the dunder __init__) were used, along with the rationale behind the code:

2.1 append_posn*(self, liq, build)*

This function was used to implement the first task of the problem statement, that is, appending a column containing values 0, 1 and -1 denoting liquidate, long and short positions respectively.
The key point here lies in **realizing that the position values depended not only on the range in which the current alpha value lied, but also on the previous position value**. This can be illustrated by the below-shown hysteresis-like diagram.

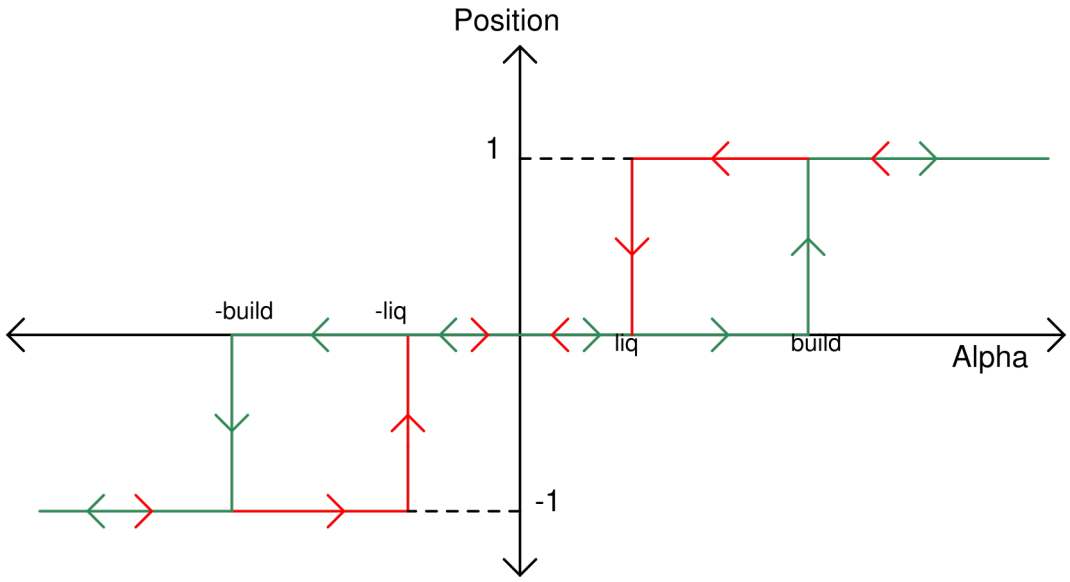

Figure 2: Assigning Position Values

It can be realized that the **current position can be decided by a boolean logic** of the range in which current alpha lies, and the previous position value. Thus, **five zone flags** (z1 to z5) and **3 condition flags** (c0 to c2) were coded and their boolean logic was used as conditioning statements for the 'if' block. The flags also took care of the edge case that liq=build.

```
if (liq != build):
    z1 = 1 if curr_alpha <= (-1*build) else 0
    z5 = 1 if curr_alpha >= build else 0
else:
    z1 = 1 if curr_alpha < (-1*build) else 0
    z5 = 1 if curr_alpha > build else 0
z2 = 1 if ((-1*build) < curr_alpha and curr_alpha < (-1*liq)) else 0
z3 = 1 if ((-1*liq) <= curr_alpha and curr_alpha <= liq) else 0
z4 = 1 if (liq < curr_alpha and curr_alpha < build) else 0

c0 = 1 if prev_posn == 0 else 0
c1 = 1 if prev_posn == 1 else 0
c2 = 1 if prev_posn == -1 else 0
```

This significantly reduced the amount of code and allowed to **condense the entire logic into a single line**. Logic for current position:

- 1, if α in z5, or α in z4 and prev_posn=1
- -1, α in z1, or α in z2 and prev_posn=-1
- 0, if α in z3, or α in z4 and prev_posn=0 or -1, or α in z2 and prev_posn=0 or 1

```
self.df.at[i, 'position'] = 1 if (z5 or (z4 and c1)) else
                            0 if (z3 or (z4 and (c0 or c2))
                                  or (z2 and (c0 or c1))) else
                            -1 if (z1 or (z2 and c2)) else np.nan
```

Note: nan value was used as it helped in finding wrong assignments
The motivation behind using boolean logic lies in realizing that the **system can be modeled as a Finite State Machine**, and the problem is essentially to **build a state-transition table** between **the three possible states**, based on the above-mentioned conditions.

2.2 backtest *(self)*

Note: The strategy class returns many other metrics other than PnL, such as max drawdown, win%, etc. Thus, the implementation of the backtest method is probably different from the expected algorithm

In order to calculate all metrics as well as an equity curve, **it is necessary to record the portfolio value** at every instant and also **record the per-trade return**. Thus, the backtest method appends a column labeled 'pfolio_value' which records the portfolio value at every instant and returns a list containing the per-trade return.
Updating the portfolio value, and recording the entry and exit prices for trades depends on the transition between positions. If δ is assumed to be the difference between the current price and price at the previous instant, the following table can be obtained:

Table 1: Value Update Condition Table

Change in position	Portfolio update	Trade status update
0 to 0	-	-
0 to 1	-	enter
0 to -1	-	enter
1 to 0	$+=\delta$	exit
1 to 1	$+=\delta$	-
1 to -1	$+=\delta$	exit, then enter
-1 to 0	$-=\delta$	exit
-1 to 1	$-=\delta$	exit, then enter
-1 to -1	$-=\delta$	-

It can be easily seen that the following relation pops out:

```
pf_val[i] = pf_val[i-1] + (last_posn * delta_arr[i])
```

Assume γ is the absolute value of numerical difference between current and previous position. Through further inspection, it can also be realized that:

- Enter a trade if gamma * abs(current_position) = 1
- Exit a trade if gamma * abs(last_position) = 1
- Exit, then Enter again if gamma = 2

Carefully arranging the exit and enter blocks, the following block of code takes care of updating the per-trade return:

```
if (abs_posn_delta * abs(last_posn)):
    per_trade_ret.append(curr_pfval - entry_val)
    entry_val = np.nan

if (abs_posn_delta * abs(curr_posn)):
    entry_val = curr_pfval
```

Note: The above block takes advantage of the fact that '2' is treated as a truth value in Python, i.e., treated as boolean value 1
The per-trade list returned is later used to calculate various metrics, and the portfolio value column is used to plot the equity curve, as well as calculate max drawdown.

2.3 max_drawdown*(self, arr)*

Maximum drawdown is the maximum drop between any two portfolio values (thus, the minimum has to occur after the maximum). This is a standard problem available on many websites offering practice questions for C++ arrays. The below code **calculates the max drawdown in O(n) time and O(1) space using a two-pointer approach**:

```
max_dd = 0
curr_max_element = 0
curr_dd = 0
for i in range(len(arr)):
    curr_val = arr[i]
    if (curr_val > curr_max_element):
        curr_max_element = curr_val
    if ((curr_val - curr_max_element) < curr_dd):
        curr_dd = curr_val - curr_max_element
    if (curr_dd < max_dd):
        max_dd = curr_dd
return (-1 * max_dd)
```

2.4 evaluate_metrics*(self, per_trade_ret)*

This method takes the per-trade return array (returned by the backtest method) and calculates various metrics. All metrics are returned in a list, the order of which is as follows:

Table 2: Index Mapping for Metrics array

Sr. No.	Index	Metric
1.	0	PnL
2.	1	Max Drawdown
3.	2	Total trades executed
4.	3	Win %
5.	4	Avg return per trade
6.	5	Avg return per profitable trade
7.	6	Avg return per loss-making trade

These metrics are a characteristic of any trade report. Thus, for the sake of completeness, I chose to include them along with PnL as well.

2.5 equity_curve*(self)*

To keep the equity curve interactive, **plotly was used as the graphing library** instead of matplotlib. This results in the graph being opened automatically in the default browser. Note that, since this method requires the backtest method to be called first, it **raises an error if the portfolio value column is not detected**.

3 Finding optimum thresholds

The **naive approach** to finding the optimum threshold values would be to brute force through **all** possible values. When tried, it resulted in an estimated run-time of around 3 hours and 20 minutes. Algorithmic trading is all about minimizing latency, and needless to say, this is not an efficient approach.
The optimal approach would be to run multiple rounds of optimization and narrow down the range of optimal thresholds in every subsequent round.

3.1 Round 1

I'm not too proud to admit that I went along the naive path first, before realizing how unfeasible it was. It would result in 16440 iterations and it was after calculating the estimated run time that I explored other options.

3.2 Round 2

It can be realized that if the gap between liquidate and build thresholds is kept too low, trades would be exited fairly quickly and the overall trade count, as well as the return per trade, would drop. Thus, the gap between both thresholds was at least 0.25 for the first round of results.
Also, since most of the alpha values are below 1, it would suffice to keep an upper limit on the liquidate threshold at least.

Table 3: Round 2 Summary

Range of liq	0.1 to 1, increment 0.05
Range of build	liq+0.25 to 2, increment 0.05
No. of iterations	441
Max PnL achieved	36093.9
Optimal liq	0.15
Optimal build	0.55

3.3 Round 3

Table 4: Round 3 Summary

Range of liq	0.13 to 0.17, increment 0.005
Range of build	0.43 to 0.57, increment 0.005
No. of iterations	252
Max PnL achieved	36480.96
Optimal liq	0.15
Optimal build	0.515

3.4 Round 4

Table 5: Round 4 Summary

Range of liq	0.149 to 0.156, increment 0.001
Range of build	0.514 to 0.516, increment 0.0001
No. of iterations	168
Max PnL achieved	36595.56
Optimal liq	[0.149, 0.151]
Optimal build	[0.5151, 0.5153]

3.5 Round 5

Table 6: Round 5 Summary

Range of liq	0.1489 to 0.1513, increment 0.0001
Range of build	0.5150 to 0.5156, increment 0.0001
No. of iterations	96
Max PnL achieved	36595.56
Optimal liq	[0.149, 0.151]
Optimal build	[0.5151, 0.5153]

3.6 Optimal Values

It was found that the **PnL remained constant** at a value of 36595.56 for any value of **liquidate threshold between 0.149 and 0.151**, and any value of **build threshold between 0.5151 and 0.5153**.

4 Results

The optimal threshold for liquidate: Any value between 0.149 and 0.151
The optimal threshold for build: Any value between 0.5151 and 0.5153
Below are the metrics obtained for liq = 0.15 and build = 0.5151 (representative of the entire range given above):

```
Liquidate Threshold = 0.15; Build Threshold = 0.5151
Optimised PnL = 36595.56
Total number of trades executed = 1656; Win percentage = 53.32%
Maximum Drawdown (absolute) = 4348.86
Average Return per Trade = 22.10
Average Return per profitable trade = 167.57
Average Return per loss-making trade = -144.07
```

Figure 3: Final Metrics

Figure 4: Equity Curve

5 Conclusion

Identifying the max PnL and corresponding threshold values was done using Excel (after generating the corresponding trade report csv file). This is unfeasible for datasets containing large number of values, and will have to be automated. However, having gone through the process manually first, it will be fairly easy to code the algorithm and move forward.