A Systematic Trading Approach from Data Mining to Live Deployment

Documentation

Submitted by:
Juri Stoffers

Supervised by:

Dr. Geoffrey Ostrin

August 21, 2025

Abstract

This document documents the mathematical and statistical background of the research and testing I conducted for my strategy development. Explination of the formulars I used and the different versions of my algorithms.

Contents

1 Outlier Detection Using Mean and Standard Deviation (Z-Score Based Outlier Detection)

1.1 Orderbook Delta defenition

The orderbook is a real-time electronic list of all pending buy (bid) and sell (ask) orders for a specific asset, organized by price level. It represents the current market depth and shows:

- Bids: The prices at which buyers are willing to purchase an asset
- Asks: The prices at which sellers are willing to sell their asset
- The difference between the highest bid and lowest ask creates what we call the "spread"

Each price level in the orderbook shows:

- The price of the order
- The total volume (quantity) of orders at that price
- The number of individual orders at that price (on some exchanges)

The **orderbook delta** is calculate by the difference between the sum of the bid and ask orders at a certain depth. Formular:

$$\Delta_{x\%} = \sum_{i=1}^{x\%} \Delta_i$$

- $\Delta_{x\%}$ is the sum of the orderbook delta for the last $x\%$ of the orderbook.
- Δ_i is the orderbook delta for the i-th level of the orderbook.

1.2 Normal Range

What I want to test is how price reacts to anomalous orderbook delta movements, particularly in scenarios where unrealistic or clearly outlying values are detected. In cryptocurrency markets, such inefficiencies can be caused by various events, one example is liquidation [1] events that interact with passive demand order stacked zones [2] During these events, the orderbook delta exhibits significant increases, providing a clear signal of market stress. This research will focus on understanding the relationship between rapid delta movements and how price reacts after these events.

[1]A liquidation event in crypot is when traders who borrowed money from an exchange to open a so called leveraged position are forced to sell or buy at a loss.

[2]A passive demand order stacked zone is a zone where a lot of orders are stacked at a certain price level.

1.3 Outlier Condition

A outlier is defined when a the Orderbook Δ_t is outside of the normal range. I defnine the normal rage as the mean $\mu(\Delta)$ plus or minus 2 times the standard deviation $\sigma(\Delta)$. For mean and standard deviation I use a rolling window of 1440 Δ values. On a timeseries dataset with a interval of 1 minute that is equal to one day.

Equation for outlier defenition:

$$\Delta < \mu(\Delta) - 2\sigma(\Delta) \quad \text{or} \quad \Delta > \mu(\Delta) + 2\sigma(\Delta) \tag{1}$$

1.4 My Hypothesis

- I expext realized volatility to increase after an outlier is detected.
- I expect that I can get a directional bias based on outliers if we combine the outlier signal with a underlying bias.

1.5 Bullish and Bearish Outliers version 1

What does bullish and bearish even mean? A bullish signal is a signal where we anticipate price will go up. A bearish signal is a signal where think price will go down.

Now I defnined a a bullish outlier as bullish if he has a certain z-score value and a bearish outlier as a bearish if he has a certain z-score value.

The mean is calculated is calculate by making a rolling window of the last 1440 Δ values. (Basically a moving average of the last 1440 Δ values) Same period is applied for the standard deviation.

Python code:

```
df['mean'] = df['delta'].rolling(window=1440).mean()
df['std'] = df['delta'].rolling(window=1440).std()
```

The z-score is calculated by the following formular:

$$z = \frac{\Delta - \mu(\Delta)}{\sigma(\Delta)} \tag{2}$$

and shows how many standard deviations Δ_t is away from current mean

1.5.1

1.5.2

1.6 Idea behind

- This method assumes data is roughly normally distributed.
- Using 2σ captures approximately 95% of data points under a normal distribution.
- You can adjust the multiplier (e.g., 3σ) for stricter or looser thresholds.

1.7 Future Plans

- Test on more data
- use rolling windows (e.g. 1 day or 1 week) for local context.
- Compare sensitivity with +- 1.5σ or +-2.5σ $\rightarrow$ optimize for best results

2 Measuring Volatility After Outlier Detection

The first Idea I had was to measure the volatility of the price after an outlier is detected. Volatility is a measure of how much price changes in eighter direction over a certain period of time. Using the following formular:

$$r_t = \frac{P_t - P_{t-1}}{P_{t-1}} \tag{3}$$

2.1 Dictionary of Terms

- P_t Asset price at time t.
- $r_t = \dfrac{P_t - P_{t-1}}{P_{t-1}}$ – 1-minute price return at time t.
- $\sigma_t^{(15)}$ – Realized volatility: the standard deviation of the next 15 one-minute returns,

$$\sigma_t^{(15)} = \sqrt{\frac{1}{14}\sum_{i=1}^{15}\bigl(r_{t+i} - \bar{r}_t\bigr)^2}, \quad \bar{r}_t = \frac{1}{15}\sum_{i=1}^{15} r_{t+i}. \tag{4}$$

  aligned so that at time t it measures volatility over $t+1$ to $t+15$.

2.2 In Python code

```
import pandas as pd
df = pd.read_csv(file_path)
df.set_index('timestamp', inplace=True)
#Compute 1-min return of delta_5

df['r_t'] = df['price'].pct_change().fillna(0)

#compute rolling std of the future 15 min window
window = 15

#rolling on r_t, then shift forward so index t hold vol of t+1...t+15
df['future_vol_15] = (
    df['r_t']
    .rolling(window=window)
    .std()
    .shift(-window)
)
```

2.3 Statistical evidence and results

Once an outlier is detected (1) inside of the Orderbook Δ, we calculate the15-minute ahead realized volatility using Equation: (4)

if a Δ_t values is flagged as an outlier (1) we record

$$\sigma_t^{(15)} = \sqrt{\frac{1}{14}\sum_{i=1}^{15}\bigl(r_{t+i} - \bar{r}_t\bigr)^2},$$

We then form two samples over our full dataset which during this test includes 104 957 one minutes intervals of P and Orderbook Δ:

$$\mathcal{S}_{\text{out}} = \{\sigma_t^{(15)} : t \text{ is an outlier}\}, \quad \mathcal{S}_{\text{non}} = \{\sigma_t^{(15)} : t \text{ is not an outlier}\}.$$

Sample mean results:

$$\overline{\sigma}_{\text{out}}^{(15)} = 0.0006244, \qquad \overline{\sigma}_{\text{non}}^{(15)} = 0.0005138,$$

This concludes an increase of r_t of roughly 21.5%

To check Statistical evidence

- a two-sample *t*-test (unequal variances), which yields

$$T = 24.72, \quad p = 4.79 \times 10^{-132},$$

- a Mann–Whitney *U*-test, which returns

$$p = 4.02 \times 10^{-157}.$$

3 Optimising for best Z-Score thresold for outliers

As stated inside of (1) we use a Z-Score thresold of 2 to detect outliers. I now want to see if by any chance there is a value where ther realized volatility is higher than if we use 2 as a thresold.

To compare the outliers Volatility with the non outliers volatility I will use the following formular:

$$U(z) = \frac{\overline{\sigma}^{(15)}_{\text{out}}}{\overline{\sigma}^{(15)}_{\text{non}}} \tag{5}$$

First I run an optimization for the thresholds of the Z-Score to find the best thresold value for the outliers on a 45 days dataset. After that I compare the result with a 107880 minutes dataset. Where I also run an optimization for the thresholds of the Z-Score to find the best thresold value for the outliers.

Top 3 z-values with largest volatility uplift 69811-minutes sample

z	N_{out}	$U(z)$ (%)	Mann–Whitney p
3.8	221	+58.36	1.913×10^{-51}
3.9	166	+61.99	4.227×10^{-41}
4.0	117	+58.36	8.228×10^{-28}

Same z-values on extended dataset (Walk forward $v1$)

z	N_{out}	$U(z)$ (%)	Mann–Whitney p
3.8	644	+51.73	2.549×10^{-77}
3.9	561	+53.28	6.150×10^{-88}
4.0	479	+50.90	5.517×10^{-59}

Top 3 z-values with largest volatility uplift (Walk forward $v2$)

z	N_{out}	$U(z)$ (%)	Mann–Whitney p
4.8	214	+65.10	6.475×10^{-33}
4.9	197	+66.81	3.693×10^{-29}
5.0	181	+70.53	6.599×10^{-28}

4 Measuring avearge return after price outlier detection

4.1 Formulars

Once a Δ_t Outlier is detected we calculate the 15-min forward return of BTC/USD price

$$\mathrm{Ret}_t^{(15)} = \frac{P_{t+15} - P_t}{P_t} \tag{4}$$

We then differentiate between a bullish and a bearish outlier. Which is already defined (1)

$$\overline{\mathrm{Ret}}_{\mathrm{bull}}^{(15)} = \frac{1}{|\mathcal{T}_{\mathrm{bull}}|} \sum_{t \in \mathcal{T}_{\mathrm{bull}}} \mathrm{Ret}_t^{(15)} \tag{7}$$

$$\overline{\mathrm{Ret}}_{\mathrm{bear}}^{(15)} = \frac{1}{|\mathcal{T}_{\mathrm{bear}}|} \sum_{t \in \mathcal{T}_{\mathrm{bear}}} \mathrm{Ret}_t^{(15)} \tag{8}$$

4.2 Dictionary of Terms

- Price at a certain time: P_t
- 15-min forward return: $\mathrm{Ret}_t^{(15)}$

```
#compute 15-min forward return of BTC/USD price
```

5 Underlying strategy Bias

Every single parameter has to fight to be implemented into my strategy. To get some kind of filter since we are working with an asset which has clear trends so it isn't stationary we need to do some trend identification. Different trends are also called diffferent regimes. I'll call it the underlying bias.

I will differentiate between three different types of regimes:

- Uptrend
- Downtrend
- Ranging

Uptrend is defined as a period where the price is making higher highs and higher lows. Downtrend is defined as a period where the price is making lower highs and lower lows. Ranging is defined as a period where price is not making higher highs and higher lows or lower highs and lower lows. But for my approach I will just use this if we are gettign mixed signals.

5.0.1 Trend identification version 1

To see in what kind of regime we currently are we first have to search for swing points. A swing point is a local extrema of a certain period. So a swing low is a local minimum and a swing high is a local maximum. Local is just that it is inside of our lookbackperiod.

Swing point identification code:

```
if current_price > np.max(lookback_window):
  swing_high = current_price
if current_price < np.min(lookback_window):
  swing_low = current_price
```

This code snipped definies the swing low/high variables, it loops through the lookback window and checks if the current price is higher than the highest price in the lookback window or lower than the lowest price in the lookback window. If it is the case we update the swing low/high variable.

Regime determination

```
window = 5
price_window = prices[i-window:i]
prev_window = prices[i-window-1:i-1]

#Then we calculate
price > local_high and local_high > prev_high
```

Meaning:

Current price > local high (recent 5)

Recent 5-period high > previous 5-period high

This would mean we are in an **uptrend**.

For a **downtrend** analogous logic applies. If none of both applies we define the regime as **ranging**.

5.0.2 Trend identification version $v2$

Now after conducting research with chatgpt I found the Aroon indicator. This indicator is a technical indicator that measures the strength and direction of a trend. It is calculated by taking the difference between the current price and the highest price over the last n periods.

In my trend identification the Aroon indicator is used to determine how recently price has made a new high or low within our defenied lookback period.

The closer the Aroon up the more recent the high and the closer th Aroon Down the more recent the low.

```
aroon_indicator = aroonIndicator(period=window)
arron_indicator.generate(df_resampled)
```

6 Finding an edge

6.1 Fees

So the first problem you run into if building a strategy is the fees. You have to pay fees for every trade you make. A fee is a fixed percentage amount of the trade size you take. On the exchange I choose, Hyperliquid the fee is about 0.003% of the trade size. Which is a normal amount inside the crypto space. Just to showcase how important fees are I will show you a simple example.

Inside of my Market simulation where I can test different kind of strategies I will set up a simple strategy. When ever a bullish outlier in the delta appears and the trend function sais that we are inside of an uptrend I will buy Bitcoin and wait for 240 Minutes.

```
#Define a mask for the bullish outliers inside of an uptrend
outlier_mask = (df_temp['outlier_context'] == 's') & (df_temp['trend'] == 'Uptrend'

#Define the entry and exit signals with the Vector BT libary
pf = vbt.Portfolio.from_signals(
  close=price,
  entries=entry_signals,
  exits=exit_signals,
  init_cash=100, #Begining amount if cash
  freq='1T'
)
```

6.2 No Fees results

Table 1: No Fees

Metric	Value
Holding Period	240 minutes
Total Return	2.599%
Mean sharp ratio	3.067088
Total Signals	187

6.3 With Fees results

Table 2: With Fees

Metric	Value
Holding Period	240 minutes
Total Return	1.537%
Mean sharp	1.841555
Mean Z-Score	2.48

6.4 No clear path

Finding an edge is a very hard task. There is no clear path to success. You kinda have to try by trail and error every error could be a step further but also a potential path into a dead end.

There are three main steps to find an edge.

- Copying other strategies
- Finding an edge by yourself
- Build a systematich version of your disgresionary trading.

While writting this I do not know yet if I was able to find an edge. Till now I searched for new things and combined different libaries from other people to develope a trend identification system. In the hyperlink below I linked the source of the trend identifiaction sysetm parts I copied.

Github account

7 Combining Indicators

Here I visulised the swing points, the EMA spread and the 100 outliers with the highest Z-Score in the same plot.

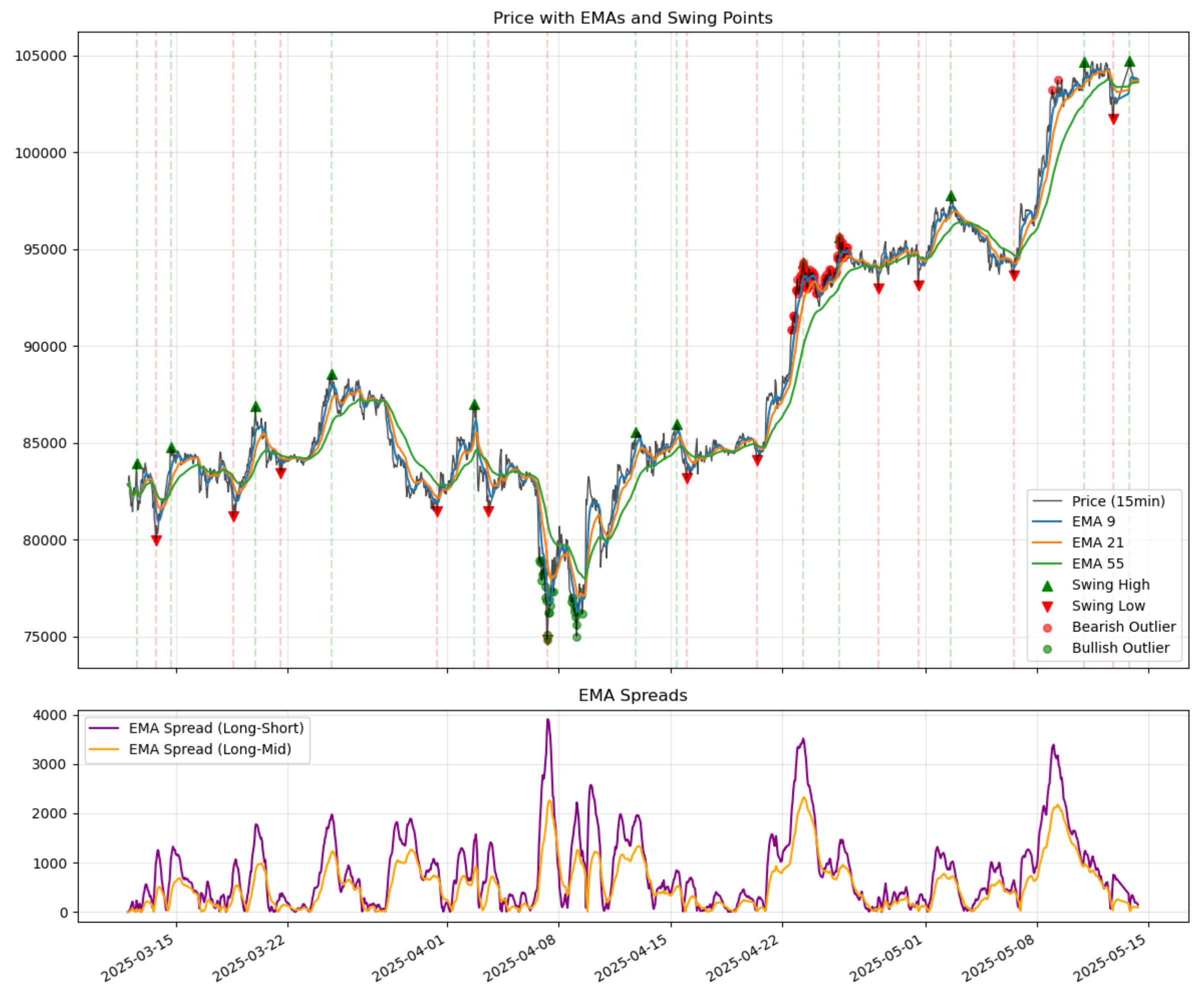

Figure 1: combined indicators png

1

[1]Chart made with Matplotlib and Seaborn

8 From outliers to strategy

8.1 Progress description

I had a pretty hard time going from developing a logic for the outliers of the Delta. Finding that volatility increases after outliers was a good first find which didn't take long and the process was pretty straight forward. But finding some defenition condtion is meet directional bias was very difficult. By directional bias I mean a price direction which is not random after some kind of condition. It was pretty clear from the begging on that the outliers by there selfe won't offer any kind of directional bias

An outlier is defined as:

- (1). $\Delta < \mu(\Delta) - 2\sigma(\Delta) \quad \text{or} \quad \Delta > \mu(\Delta) + 2\sigma(\Delta)$

Then since Bitcoin clearly is not a stationary asset I needed to find some kind of trend identification. I found that the swing points are a good indicator for this. We determined a trend by using looking at swingpoints. A swing point is a local extrema of a certain period.

8.2 Where did I start my research

```
#compute swing points
swing_points = swing_points(df['price'], period=n)
```

Parameters of the swing_points function:

- n is the lookback period
- $df['price']$ is the price column of the dataframe (Which is a timeseries)
- P_t is a value of $df['price']$ at time t

We basically got through the time series dataset and look back n P_t values. The highest and lowest points inside of that specific lookback period are the swing points. After that we detetermine if price is making higher highs or lower lows. We check this by storeing the last swing points and wait till we are eighter making a higher high or a lower low.

8.3 Finding an edge

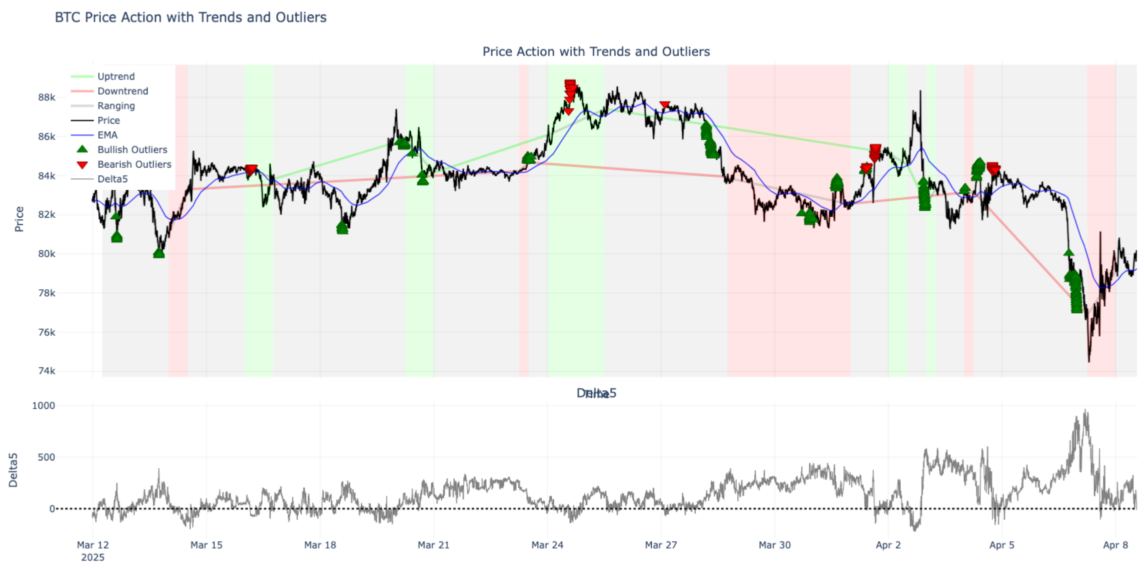

Figure 2: Visualization of Δ_t outliers and regime identification

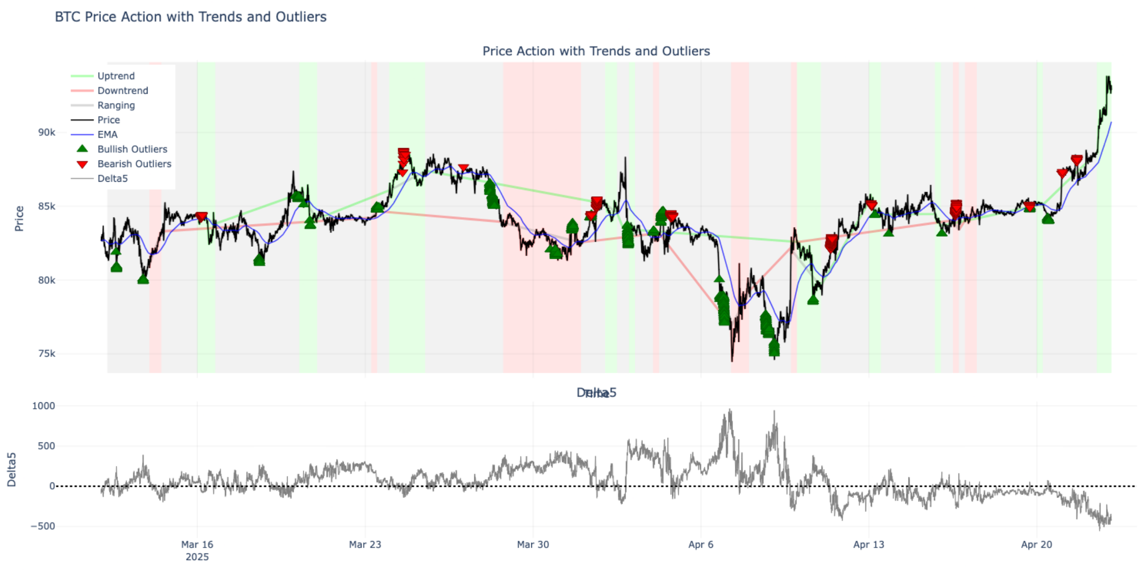

Figure 3: Visualization of Δ_t outliers and regime identification

Here you can see two visulaisations of outliers and regime identification. On two different timeseries datasets. This was my trend identification system $v1$ which is only based on swing point mapping.

After plotting different kind of parts of Timeseries Data sets I created I was pretty sure that somewhere I would be finding an edge since the outliers often where at good entry points for a strategy. Only problem was I wasn't able to get any statistical proof ot this. After finding a Github Repo which used Vectorbt to test 1000 strategies at the same time I tried a similar approach and tested different conditions to see if there was some kind of edge.

8.4 Monte Carlo Testing

First of I have to explain how I search for a strategy and how it nearly killed my computer. We start by defining a different kind of sample sizes of all the Δ_t outliers time periods after the outlier detection.

We want to know the return after our condition meets and then check in after z minutes after that

```
sampling_percentages = [0.1, 0.2, 0.6, 0.8, 1.0]
holding_periods = [60, 120, 240, 360]
```

The sampling percentage definies how many of the existing outliers we use and test the returns on. So let's say we have x amount of Δ_t Outliers. We then randomly select z % amount of outliers and test the returns on them. We do this 1000 times and then take the average return of all the 1000 tests. The second step is to test each sampling percentage on the different holding periods. So let's say we have a sampling percentage of z % and a holding period of y minutes. We then take the z % amount of outliers and test the returns on them after y minutes. This test is basically copied from the Vectorbt Github Respository. I just changed the code to fit my needs and added some extra information returns. My code gives back the average return, the Sharpe, expectancy and mean Z-Score.

- **Sharpe Ratio**: The Sharpe ratio is a measure of the risk-adjusted return of a strategy. It is calculated by dividing the average return of the strategy by the standard deviation of the returns.
- Calculated by $\frac{\overline{r}}{\sigma}$ Where $\overline{r}$ is the average return and σ is the standard deviation of the returns.
- **Expectancy**: The expectancy is a measure of the profitability of a strategy. It is calculated by dividing the average return of the strategy by the average loss of the strategy.
- Calculated by $\frac{\overline{r}}{\overline{l}}$ Where $\overline{r}$ is the average return and $\overline{l}$ is the average loss.
- **Mean Z-Score**: The mean Z-Score is a measure of the average Z-Score of the strategy. It is calculated by taking the average of the Z-Scores of the strategy.

- Calculated by $\frac{1}{n}\sum_{i=1}^{n} z_i$ Where z_i is the Z-Score of the strategy at time i and n is the number of Z-Scores.
- **Average Return**: The average return is a measure of the average return of the strategy. It is calculated by taking the average of the returns of the strategy.
- Calculated by $\frac{1}{n}\sum_{i=1}^{n} r_i$ Where r_i is the return of the strategy at time i and n is the number of returns.

Now it is time to test some strategies. The first strategy I want to test is pretty intuitive. Measure the average return of bullish outliers inside of an uptrend.

spacing

label A outlier is bullish if the the orderbook Δ_t is two standard deviations above the mean of the last 1440 Δ_t values and bearish if it is two standard deviations below the mean of the last 1440 Δ_t values.

Now when we backtest a strategy we have to have a few things in mind. First of all we are backtesting on on historical data and if we just use different kind of entry conditions we might just change the entry conditions till we have a good end results. This would be overfitting and not work on future data. In order to prevent overfitting we have to use a holdout sample. I have one dataset on which we test our entry conditions to see if they even have potential.

If these conditions are met for a condition I test on some out of sample data. (Out of sample data just means that we test on new data) Most of the time things end up not even passing the first out of sample test and lose their potential instantly. If not I have another out of sample test and if that one is passed we are ready to do some more monte carlo testing and look how potential equity curves look like. And to make things even worse we can assume that the sharpen ratio will decrease by atleast 20% compared to the backtest simulation.

8.5 Backtesting framework

My backtesting framework is built in Python using the VectorBT library and consists of several key components:

Data Pipeline

Connection to Postgres database hosted on railway.app with global access. Live data mining progress

- Historical price data from Coinbase (BTC/USD)
- Orderbook delta data at different depths ($\Delta_{1\%}$, $\Delta_{2.5\%}$, $\Delta_{5\%}$)
- 1-minute timeframe for base calculations
- Direct ema calculation which a lenght of 50 on the 1h timeframe

Testing Methodology

The framework implements a three-stage testing process:

1. **Initial Sample Testing**
   - Test strategy on initial dataset
   - Multiple sampling percentages: [0.1, 0.2, 0.6, 0.8, 1.0]
   - Various holding periods: [60, 120, 240, 360] minutes
2. **Out-of-Sample Validation**
   - Test promising strategies on separate dataset (Two different datasets)
   - Require consistent performance across datasets
3. **Monte Carlo Simulation**
   - Random sampling of trade opportunities
   - 1000 iterations per test configuration
   - Analysis of distribution of outcomes

Performance Metrics

Each strategy is evaluated using:

- **Sharpe Ratio**: $\frac{\bar{r}}{\sigma}$
- **Expectancy**: $\frac{\bar{r}}{\bar{l}}$
- **Average Return**: $\frac{1}{n}\sum_{i=1}^{n} r_i$

Risk Management

The framework incorporates:

- Trading fees (0.003% on Hyperliquid)
- Slippage of (0.0001% on Hyperliquid)
- Maximum drawdown limits
- Stop-loss and take-profit levels

Implementation

```
# Example of strategy implementation
def backtest_strategy(data, params):
    # Define entry/exit signals
    outlier_mask = (data['outlier_context'] == 's') &
                   (data['trend'] == 'Uptrend')

    # Create portfolio simulation
    pf = vbt.Portfolio.from_signals(
        close=data['price'],
        entries=outlier_mask,
        exits=exits_after_n_bars(outlier_mask, n=params['holding_period']),
        init_cash=100,
        freq='1T'
    )

    return calculate_metrics(pf)
```

This framework allows for rapid testing of multiple strategy variations while maintaining strict validation criteria to prevent overfitting.

9 Developing a strategy

9.1 Initial thoughts

I will only able to show a few strategies backtests since I tested on about 150 different logics and only 1 of them passed all the out of sample tests. I will show the best ideas I had and the results of the backtests.

9.2 Reverse strategy

So logical thinking we could assume that long strategy can be reversed into a short strategy and vice versa. The problem is that we have fees and slippage. So we need a certain amount of profit to cover the fees and slippage. And since my strategy has a rather high frequency we need to have a lot of trades to cover the fees and slippage.

9.3 First strategy test

So the first strategy I test was before I developed the backtesting framework. I was testing on a simple logic. If the price on the 1h timeframe closed above the 50 period EMA and the Δ_5 of the orderbook was positive we would enter a buy position if price was in a range of 0.05% of the $EMA_5 0$.

In addition to that I was using a trailing stop. The trailing stop was set to 1% of the price. A trailing stop is a stop loss that is adjusted to the price of the asset. So if the price goes up the stop loss goes up with it. The idea behing this is that we can catch a bigger upwards in contrast to a fixed stop profit.

I only tested this on a short sample of data because I just had started my data mining process and didn't have any more data. But I can asure that more data didn't make it more profitable just worse.

9.3.1 Code

```
# Define entry/exit signals
range_pct = 0.0005
trailing_stop_pct = 0.01

entries = (
  (df['bias'] > df['ema']) &
  (df['price'] >= df['ema'] * (1 - range_pct)) &
  (df['price'] <= df['ema'] * (1 + range_pct))
)
```

9.3.2 Strategy 1 results

Table 3: Detailed backtest results for the first strategy test.

Metric	Value
General	
Start	2025-03-12 00:00:01
End	2025-03-17 07:24:01
Period (minutes)	7,636
Performance	
Start Value	100.00
End Value	97.13
Total Return [%]	-2.87
Benchmark Return [%]	0.38
Total Fees Paid	2.21
Open Trade PnL	0.23
Trades	
Total Trades	23
Closed Trades	22
Win Rate [%]	31.82
Profit Factor	0.75
Expectancy	-0.14
Best Trade [%]	3.15
Worst Trade [%]	-1.24
Avg Winning Trade [%]	1.35
Avg Losing Trade [%]	-0.83

9.4 Learning from the first strategy

So from march till july I didn't backtest any strategies. I was focusing on developing signals and indicators. This backtest clearly showed me that I have to do research first and have some kind of background idea of what I am doing.

9.5 Explaining figure

So on the Y axis we have the average return of the strategy. And on the X axis we have the sampling percentage. A sampling percentage of 10% means that we are using 10 random precentage of the Δ_t outliers my system identified over the backtesting timeseries dataset and then testing the returns on them. We conduct this 1000 times and then take the average return of all the 1000 tests. In addition to that we are testing the returns on different holding periods. So for example if we are testing the returns on a holding period of 60 minutes we are testing the returns on the 10 random precentage of the outliers after 60 minutes. We conduct this 1000 times and then take the average return of all the 1000 tests.

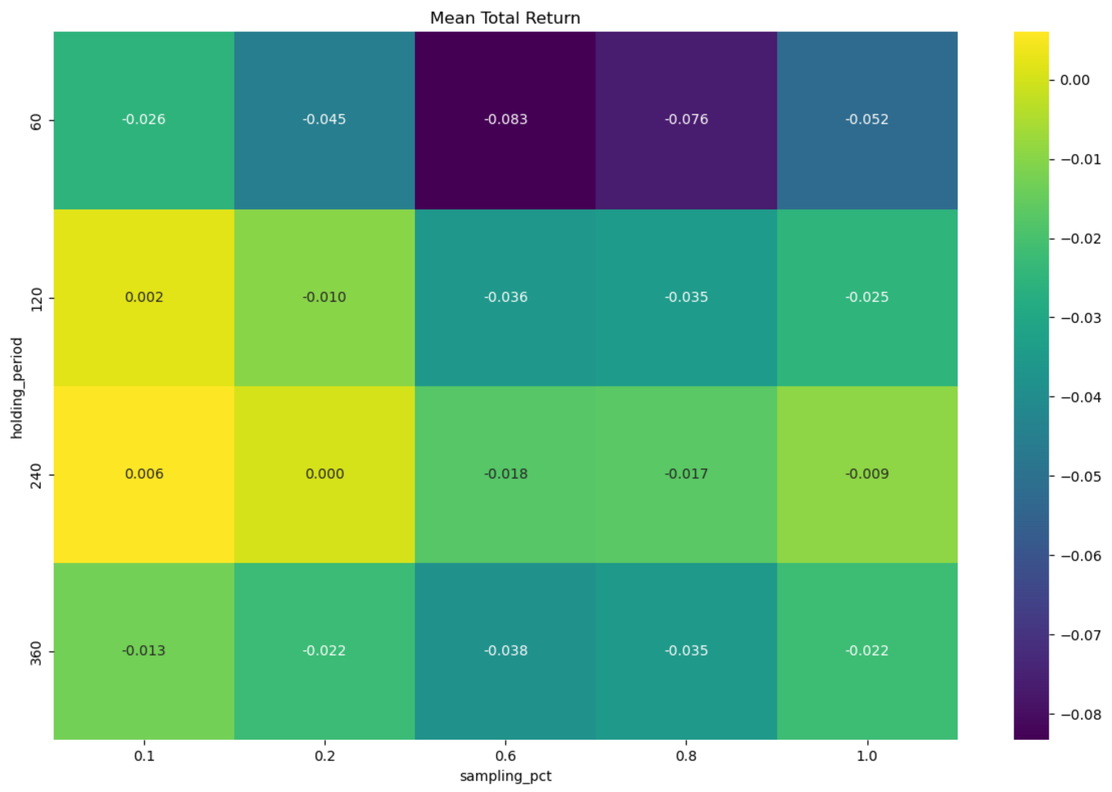

Figure 4: Strategy 2 results

9.6 Strategy 2

So this was the second strategy I tested and the first time I used my new backtesting framework. Results from this backtest: 4

- A $\Delta_{5,t}$ value is identified as a bullish outlier (see Section 1).
- Trend is identified as uptrend

10 Algorithms Implementation

10.1 Outlier detection version 2

After not finding any edge with the normal z-score outlier detection I decided to implement a new outlier [2] detection logic.

The IQR method divides data into for equal groups.

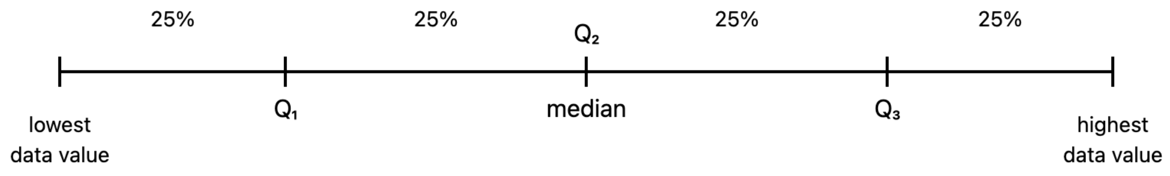

Figure 5: IQR method

- Q_2 (median): The middle value that divides the entire dataset into two equal halves
- Q_1 (first quartile): The median of the lower half of the data (values below Q_2)
- Q_3 (third quartile): The median of the upper half of the data (values above Q_2)

Each quartile represents 25% of the data, making it easy to identify the middle 50% of values (between Q_1 and Q_3).

An outlier is now defined as a value which falls outside of the interquartile range mutlitpled by a factor z in my case $z = 2$. IQR is defined as $IQR = Q_3 - Q_1$.

Outlier defenition:

$$\Delta_t < Q1 - z \cdot (IQR) \quad \text{or} \quad \Delta_T > Q3 + z \cdot (IQR)$$

[2] an extrem high or low value compared to other values inside of the dataset

IQR method python implementation

```
delta5_15 = df15['delta5']
roll = delta5_15.rolling(window, min_periods=window // 4)
#window is a parameter inside of function
#calculating rolling quartiles
q1 = roll.quantile(0.25).shift(1) #first quartile
q3 = roll.quantile(0.75).shift(1) #third quartile
iqr = q3 - q1 # interquartile range

#calculating bounds
lower_bound = q1 - z_value * iqr
upper_bound = q3 +  z_value * iqr



#identifying outliers
outliers = (delta5_15 < lower_bound) | (delta5_15 > upper_bound)
```

Problems with z-score threshold

If the data is skewed or has fat tails the z-score method will be biased. The IQR method is more robust to outliers and is not sensitive to the distribution of the data. In simple words if the market is volatile the z-score method will be biased. The standard deviation be higher than it should and the mean will be affected too by strong volatility.

11 Algorithmic infrastructure implementation

Now since my strategy allows mutliple positions open at the same time the infrastructure turns out to be a bit more complex. If we open two positions on the BTC/USD pair with a size of 1000$ we end up having one positions with a size of 2000$.

Vectorbt backtest implementation

```
entries = (df_temp['outlier_context'] == 's') & (df_temp['trend'] == 'Uptrend')
exits = exits_after_n_bars(entries, n=120) #120 minutes after entry

pf = vbt.Portfolio.from_signals(
  close=price,
  entries=entry_signals,
  exits=exit_signals,
  accumulate=True, #allows multiple positions open at the same time
  init_cash=100, #Begining amount if cash
  freq='1T'
)
```

As you can see in the figure 6 there are times where multiple positions are open at the same time.

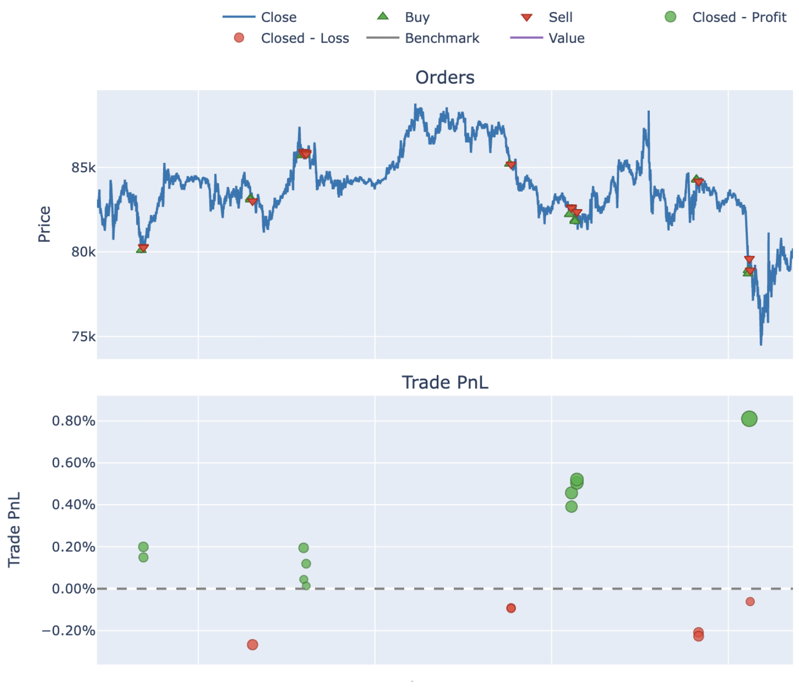

Figure 6: Trades

11.1 Handeling trades asynchronously

I decided to handle the issues of multiple positions with different exit times by using an asynchronous approach. Every positions is handled in a seperate python thread. The thread just sells the exact amount of one single position which is 10% of the portfolio. The maximum of open positions is set to 20.

The algorithm uses a class [3] called AsyncPositionManager. The position manager initializes is initialized as part of the singalloop which is responsable to check every minute if an entry signal is detected. If an entry signal is detected the position manager will open a position which is handled inside of an asynchronous position task that runs independently.

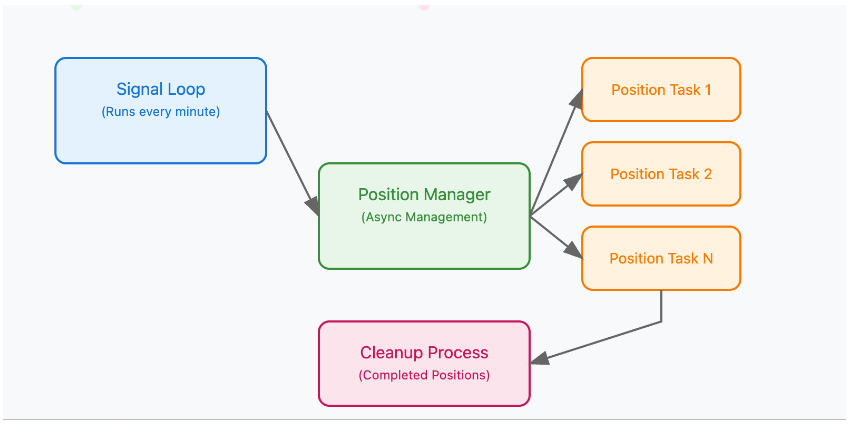

Figure 7: AsyncPositionManager

[3] A class in Python is a blueprint for creating objects. It defines a set of attributes and methods that an object of that class will have.

11.2 Strategy 3

So this was the third strategy I tested. I was testing on a simple logic. If the price on the 1h timeframe closed above the 50 period EMA and the Δ_5 of the orderbook was positive we would enter a buy position if price was in a range of 0.05% of the EMA_50.

12 Code implementation for data creation

I chose to create my own dataset because it offers a key advantage: it allows me to experiment with ideas and patterns that are less likely to have been explored before. This increases the chance of discovering something new that might be missed when using more commonly available datasets.

There are platforms that offer order book delta data for free, but only for discretionary trading (i.e. they only display the data in a chart and do not provide it as a downloadable time series). My idea was to create a Python script which hits the API in one-minute intervals and stores the data inside a Postgres database. I have been hosting the data mining script on my PaaS since March 11, 2025 Live database connection.

The creation of the time series was done through a public exchange API[1], with data fetched in one-minute intervals and stored inside a database on a PaaS[2].

In my dataset, I calculate the 1%, 2.5%, and 5% delta of the order book. Additionally, I include the current Bitcoin price and the corresponding timestamp. The program I wrote collects data by sending API requests to the Coinbase Exchange API at one minute intervals. It retrieves both the current Bitcoin price and a full snapshot of the order book at that moment. A complete snapshot is necessary to calculate order book deltas at different percentage depths accurately.

The order book delta is calculated by summing the total value of bid orders $n\%$ below the current price (P_t) and subtracting the total value of ask orders that are $n\%$ above the current price (P_t).

$$\Delta_{OB} = \sum_{i=1}^{n} V_{bid_i} - \sum_{i=1}^{n} V_{ask_i}$$

The main problem I faced here is that price time series in trading are usually constructed so that each data point represents the closing price of a fixed time interval (e.g. every minute at `xx:00`). If data points are missing or misaligned (e.g. recorded at `xx:00:15` instead of `xx:00:00`), then the time series becomes irregular and any backtest conducted on it would be flawed. A few key things flow together here which make it difficult to get a perfectly aligned timeseries. As python runs in a single threaded [3]

[1]API stands for Application Programming Interface. It is a tool that allows programs to request and receive specific data from external services—in this case, used to fetch real-time Bitcoin price and order book data from the Coinbase crypto exchange.

[2]PaaS stands for Platform as a Service. It is a cloud computing model which gives developers a platform to host their applications so that they are globally accessible on the internet and run 24/7

[3]A python program runs by executing line after line. Using the thread

12.1 Handling Time Series Alignment in High-Frequency Data Collection

The implementation addresses a critical challenge in high-frequency trading data collection: ensuring precise temporal alignment of price data points. The primary technical constraint is the inherent latency of the Coinbase Exchange API, which introduces approximately 100ms of delay per request. This delay, while minimal for general purposes, becomes significant when constructing accurately time-aligned price series.

12.1.1 Core Challenges

- API request latency (100ms) causing timestamp misalignment
- Need for regular intervals (xx:00:00) versus actual recording times (xx:00:15)
- Risk of missing data points affecting backtest validity

12.1.2 Solution Implementation

The solution implements a two-layer approach to handle these challenges:

1. **Continuous Data Collection Layer**
   - Implements parallel API requests using ThreadPoolExecutor
   - Maintains a bounded queue (maxsize=1000) to buffer data points
   - Collects data at 2-second intervals to ensure sufficient sampling density
2. **Time Series Alignment Layer**
   - Aligns timestamps to 10-second boundaries:

     ```
     current_timestamp = current_timestamp.replace(
         second=(current_timestamp.second // 10) * 10,
         microsecond=0
     )
     ```

   - Tracks collection latency for each data point:

     ```
     collection_latency = (current_timestamp -
                           last_data.timestamp).total_seconds()
     ```

   - Flags interpolated data points when latency exceeds threshold:

     ```
     is_interpolated = collection_latency > 10
     ```

12.1.3 Data Quality Assurance

The system maintains data quality through several mechanisms:

- Records collection latency in the database for each data point
- Marks data points as interpolated when timing constraints are violated
- Provides transparency for backtest validation by tracking data quality metrics

This implementation ensures that while API latency cannot be eliminated, its effects on the time series are documented and trackable, allowing for more accurate backtesting by either filtering out or appropriately handling interpolated data points.

13 sources

13.1 How good or random is your trading

Twitter article about random walk theory and how to test if a strategy is random or not.

13.2 Tweets

A lot of my ideas come from thiy guys Tweets, to research into this. He is a disgresionary trade what means he trades based on his own decisions and has a mental framework to trade. I learned what an orderbook delta is from him.

13.3 TRDR This platform allow you to use different kind of metrics on different Timeseries datasets (BTC/USD Price, orderbookdelta and Open Interest

his platform allow you to use different kind of metrics on different Timeseries datasets like: BTC/USD Price orderbookdelta and Open Interest

13.4 Trend line automation

Used this for the $v3$ version of my trend identification system

13.5 Vectorbt github documentation respository.

I implemented some ideas from this repository for my backtesting framework and used vectorbt for the fees and slippage implementations

13.6 Deep research by ChatGPT

The OpenAI subscription enables you use deepsearch function from chatGPT I would say this tool is probably useful for research but I didn't find any value from this besides finding out about new indicators. The only thing finding I made from these two pdfs was the Aroon indicator which I implemented in the second verions of my trend indentification system $v2$

13.7 Markets in profile – James Dalton

Altho this book is for a disgresionary trading approach it taught me a lot about the Efficient Market Hypothesis and trading psychology.

13.8 Do retail traders stande a chance?

Good read about if retail traders even have a chance to beat the market.