Quant Trading Project Draft

Class : Quantitative Trading Strategies

Authors : Giovanni Longo (12228450), Felix Poirier (12410053)

Instructor : Brian Boonstra

Date : 2024-02-22

Abstract

In this analysis, we explore a novel way to incorporate book data into future price predicition (theo). Among other metrics we use an approach mentionned in Jean-Philippe Bouchaud et al 2002 Quantitative Finance 2 251 to synthesize the shape of the book using the first two eigenvalues involving the distance between incoming orders and the midprice. We then use a variety of Machine Learning models with this new information incorporated to forecast the midprice one second in the future. We select our favorite model by performing accuracy testing on our validation dataset.

Using this timeserie, we then run a market making strategy, and explore its performance on validation data. We involve utility functions in both models to manage inventory risk and embed both the theo and the confidence metrics, which we where inspired to do through M. Avellaneda et al 2006. We analyze the strategies using multiple parameters and establish a set of preferred hyperparameters.

Table of Contents

3. Training and Model Performance Analysis

In this section, we train the models and assess their performance on the dataset, through the validation squared loss.

4. Strategies

In this section, we run the strategies and explore the effect of changing the parameters.

5. Take-Aways

In this section, we share our major takaways, about the performance of this HFT strategies and what we have learned in the process.

In [1]:
```
from utils.plotting import plot_order_book, plot_theo_lagg, plot_butterfly_s
from utils.data_formatting import get_crypto_data, sample_data, generate_y
from utils.feature_spaces import FirstFeatureSpace
from utils.analysis import theo_empirical_accuracy_ts, rolling_mean
import utils.market_making as mm
import pandas as pd
import warnings
import importlib
import itertools
import matplotlib.pyplot as plt
import pickle
warnings.filterwarnings("ignore")
```

Exploratory Data Analysis

In this section we should mainly provide much of the plots used in the draft and show how some of the building blocks behave.

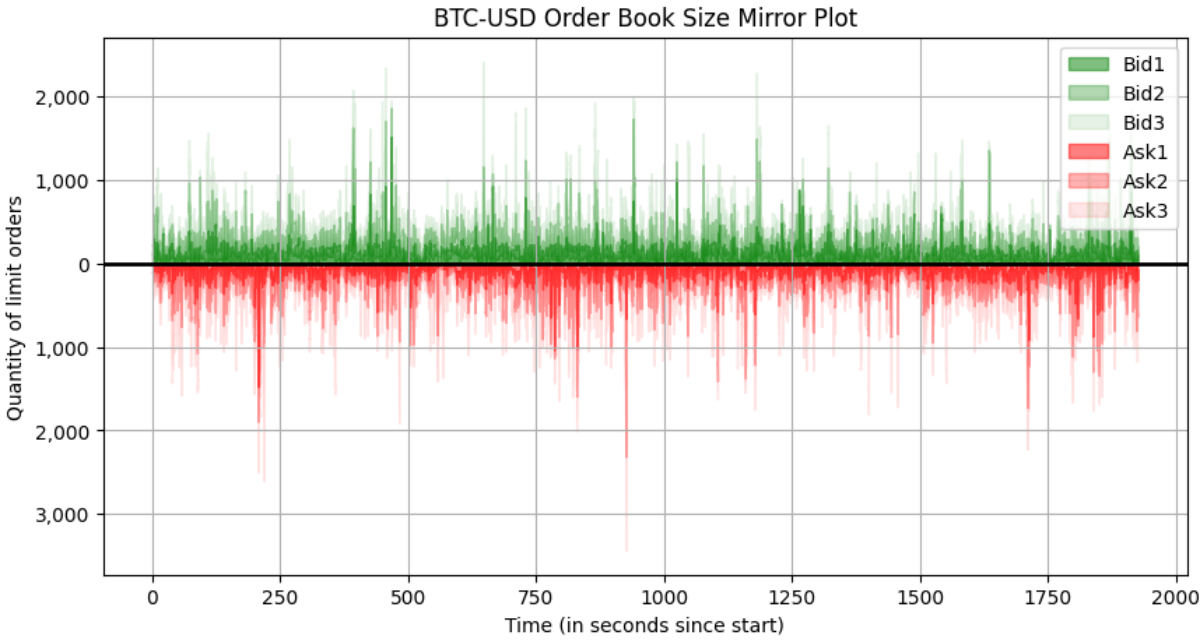
In [2]:
```
sources = [
    ("2023-01__BTC-USD_orders.h5", "2023-01__BTC-USD_trades.h5"),
]
data = get_crypto_data(sources)
```

Splitting of our Dataset

In [3]:
```
PCT_TRAINING = 0.5
PCT_VALIDATION = 0.3
PCT_TESTING = 0.2
TOTAL_OBS = len(data["BTC-USD"]["books"])
training_data = sample_data(data, (0, int(TOTAL_OBS*PCT_TRAINING)))
validation_data = sample_data(data, (int(TOTAL_OBS * PCT_TRAINING), int(TOTA
```

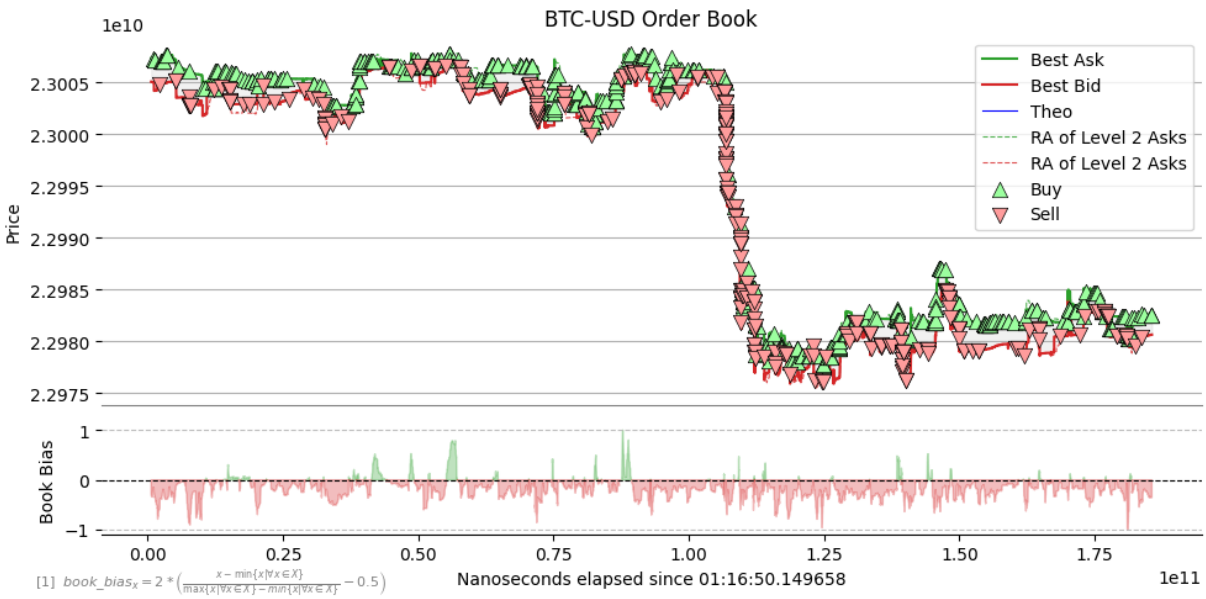
please note: not all of the data could be used due to limitations in computational ressources

In [4]:
```
c = 209000
temp = sample_data(training_data, (c, c+200000))
plot_butterfly_size(temp["BTC-USD"]["books"], "BTC-USD Order Book Size Mirrc
```



we see in the above plot that the volume of this specific subset of the book has a lot of variation, and even has a huge outlier. Hence using an exponential weighted value model, such as the one described in M. Avellaneda et al 2006 could highly affect the performance of our algorithm.

In [5]:
```
temp["BTC-USD"]["books"]["theo_value"] = None
temp["BTC-USD"]["trades"]["Price"]*=1_000_000
viz_data = sample_data(temp, (0, 20000))
plot_order_book(viz_data["BTC-USD"]["books"],viz_data["BTC-USD"]["trades"],
```



We can see from our data, that there are significant spreads at any moment in time. We calculate an average spread of about $2. This tick is not completely insignificant given that the price of Bitcoin in our sample is 22976. We see that this market is relatively illiquid, with inconsistent changes in the spread.

In [6]:
```
if (temp["BTC-USD"]["books"]["Ask1PriceMillionths"]>=temp["BTC-USD"]["books"
    print("No arbitrage opportunity")
```

```
No arbitrage opportunity
```

Given that this specific dataset has some issues regarding the books crossing, we've specifically selected above a subset which contains no arbitrage opportunity

Explaining the Setup of the Analysis

After initially composing an exponential market making model, similar to the model pioneered by M. Avellaneda and S. Stoikov, in their famous paper titled "High-frequency trading in a limit order book", we found in our draft that such a model was incredibly primitive, and served more as a backbone to a much greater problem than as a standalone approach to complex market behavior. We find that this approach seems to have little to no predictive power over the direction of the price. Hence, the "agent's value function" coined in the paper, though useful as a starting point can clearly not be used to robustly establish a "theo" value.

Since the release year of the paper, emergence in data processing power and the discovery of many learning algorithms, lend much relevance to using empricism to propose dynamic solutions to the market making problem. We introduce an approach, that will aid our final market making algorithm in making more educated decisions.

Furthermore we seperate our "feature spaces" which are preprocessed independent values that'll help us robustly predict our 3 distinct objective functions. We are deeply concerned with both the accuracy of these given models, which ceteris paribus should help our market making strategies take better decisions.

Methodology

Predictive Models

Our project will mainly aim to design and implement 3 distinct models using our feature space:

$$\begin{aligned} \mathrm{M}(\mathrm{B}_t) &= \mathrm{E}[\mathrm{P}_{t+\delta} - \mathrm{P}_t | \mathrm{B}_t] \\ \mathrm{C}(\mathrm{B}_t, t) &= \mathrm{P}\{\mathrm{M}(\mathrm{B}_t) = \mathrm{P}_{t+\delta} - \mathrm{P}_t | \mathrm{B}_t\} \end{aligned}$$

The symbols are defined as follows:

B_t: Information contained within the book at time t and before

P_t: The current midprice. Note that: $\mathrm{P}_t \in \mathrm{B}_t$

$\mathrm{M}(\mathrm{B}_t)$: The expected change in mid price, from time t up to some δ units of time into the future

$\mathrm{C}(\mathrm{B}_t, \mathrm{M})$: Our degree of certainty regarding our estimate in $\mathrm{M}(\mathrm{B}_t)$

please note that the above equations make no assumptions about the distribution of the specific variables involved

In [7]:
```
from sklearn.linear_model import LinearRegression, Lasso, Ridge
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import AdaBoostRegressor, GradientBoostingRegressor
from sklearn.svm import SVR
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline

models = [
    LinearRegression(),
    Lasso(),
    Ridge(),
    RandomForestRegressor(n_estimators=100, max_depth=5),
    AdaBoostRegressor(n_estimators=100),
    GradientBoostingRegressor(n_estimators=100),
]

DELTA = 1e+9 # 1 second
LATENCY = 5e+6 # 5 milliseconds
```

These are the specific machine learning models we will use to predict theos. Theos in this analysis are defined as midprices that occur 1 second in the future.

In [8]:
```
def make_model(feature_space, model):
    return Pipeline([
        ('features', feature_space),
        ('scaler', StandardScaler()),
        ('model', model)
    ])
```

Feature Exploration

Initial Feature Space

Concerned Features

- Mid Price
- Bid-Ask Spread
- VWAP
- Eigenvalues of covariance Matrix, inspired by Jean-Philippe Bouchaud et al 2002 Quantitative Finance 2 251

Justification:

In Bouchaud et al., the authors make a note of the given properties of the book;

"We have also studied the full covariance matrix of the fluctuations $C_V(\Delta, \Delta')$ *(note that* $C_V(\Delta, \Delta') \equiv \sigma 2V(\Delta)$*). For France-Telecom, its first eigenvalue corresponds to a dilation of the book, whereas the second one reflects an even/odd oscillation (orders are more numerous at prices in tenth of Euros than in twentieth of Euros). "*

From this we derive our own similar version of this metric, which tries to emulate Δ through our consolidated dataset, that contains aggregated levels.

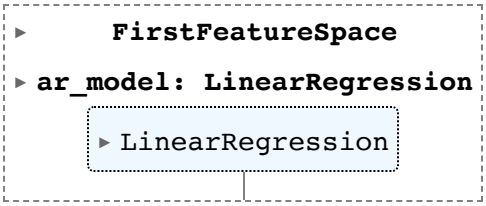
Let $i \in \{1, 2, \ldots, n\}$, and $1m_t \in R^6$ (we have 6 levels in total) which represents the midprice at time t, then we have

$$\Delta = \begin{bmatrix} \frac{1}{6}\sum_j |V_{j,t_1} - V_{j,t_0}| \times |P_{j,t_1} - 1m_{t_0}| \\ \frac{1}{6}\sum_j |V_{j,t_2} - V_{j,t_1}| \times |P_{j,t_2} - 1m_{t_1}| \\ \vdots \\ \frac{1}{6}\sum_j |V_{j,t_n} - V_{j,t_{n-1}}| \times |P_{j,t_2} - 1m_{t_{n-1}}| \end{bmatrix}$$

We can choose a subsample of changes in the book, we chose to look in the past, and determine the eigenvalues of these Δ_i from a rolling window. We then find the first two

eigenvalues of the following matrix: $\Delta\Delta^{\mathrm{T}}$

In [9]:
```
feature_space = FirstFeatureSpace()
display(feature_space, feature_space.info())
```



	method name	midprice	vwap	bid_ask_spread	delta_eigvals
output column name		midprice	vwap	bid_ask_spread	delta_eigval_<ith_eigval>

note that the mapping between methods and output columns is not 1-to-1, for `delta_eigval` *we will extract 2 eigenvalues and for* `rolling_ar_vol` *we will extract 5 distinct rolling windows.*

Example Training

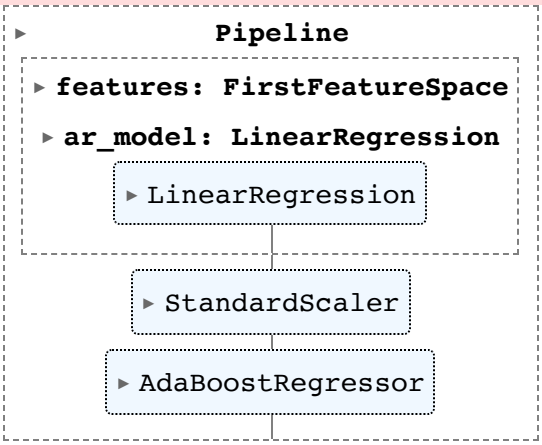
In this section we run an example model to showcase the methodology that will be used in the validation selection process

In [15]:
```
y = generate_y(temp["BTC-USD"], DELTA)[['1sec_midprice']]
```

In [16]:
```
m = make_model(feature_space, AdaBoostRegressor())
m.fit(temp["BTC-USD"], y)
```

```
Computing delta eigenvalues:   0%|          | 457/199995 [00:00<03:44, 887.08it/s]
Computing delta eigenvalues: 100%|██████████| 199995/199995 [17:41<00:00, 188.46it/s]
```

Out[16]:



As we can see in the time taken to run the cell above, calculating eigenvalues is pretty laborious, especially given the size of our reduced dataset. In practice this section of our analysis should be improved through a more efficient implementation in C++. Even

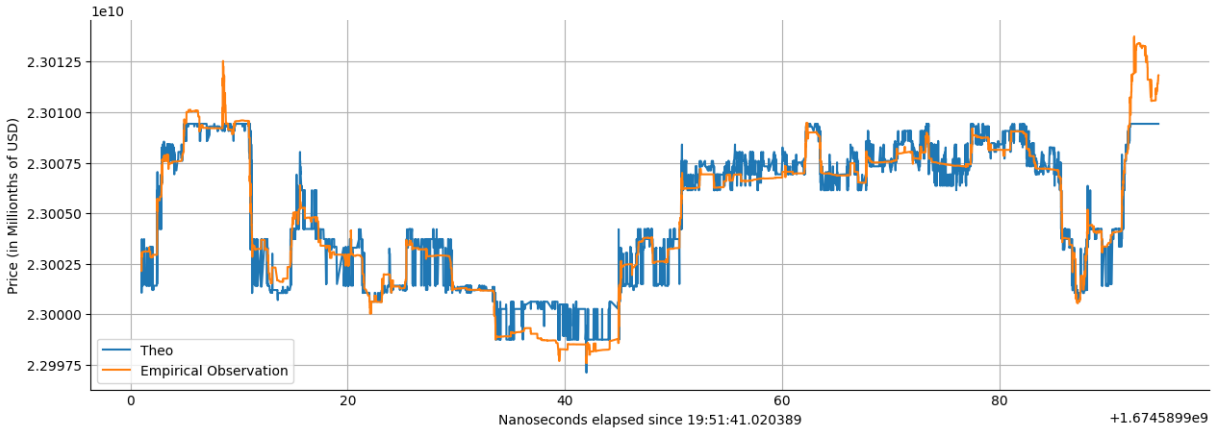
though this section of the program is vectorized, we eould still run into a pretty strong bottleneck if we were to actually implement this. The emergence of GPUs could be a potential solution to this, given that our CPUs can only handle so much.

In [81]:
```
temp_validation = sample_data(validation_data, (180000, 190000))
if (temp_validation["BTC-USD"]["books"]["Ask1PriceMillionths"]>=temp_validat
    print("No arbitrage opportunity")
theos = m.predict(temp_validation["BTC-USD"])
y_val = generate_y(temp_validation["BTC-USD"], DELTA)[['1sec_midprice']]
```

```
No arbitrage opportunity
Computing delta eigenvalues:   3%|          | 325/9995 [00:00<00:03, 3024.57
it/s]
Computing delta eigenvalues: 100%|██████████| 9995/9995 [00:03<00:00, 2828.9
4it/s]
```

The validation set is used to both assess the performance of our machine learning algorithms, and the performance of our trading algorithms. It's important to note that this approach has some inherent bias, namely because we run our trading strategy on the same data we used to select which model we prefer. Note that we aware of this, given a difficulty in the testing data we were unable to using a dataset to perform an unbiased estimation of our strategy's performance on data we haven't seen before. Not withstanding this, we made sure that our strategy doesn't have lookahead bias.

In [100…
```
plot_spread(temp_validation["BTC-USD"]["books"]["timestamp_utc_nanoseconds"]
```



We see here that AdaBoost doesn't seem to have an issue fitting and then predicting the prices of our asset 1 second into the future. However it is plagued by variance, and seems very noisy especially along flatter segments. This could send mixed signals to our trading algorithm and prone it to constantly modify order. This would decrease the probability of our orders being filled.

Validation

Here we run all models and assess their performance

In [30]:
```
temp_validation = sample_data(validation_data, (0, 100000))
```

increasing the size of our validation set

In [26]:
```
data_pipe = Pipeline([
    ('features', feature_space)
])

X_train = data_pipe.fit_transform(temp["BTC-USD"])
```

```
Computing delta eigenvalues:   0%|          | 0/199995 [00:00<?, ?it/s]
Computing delta eigenvalues: 100%|██████████| 199995/199995 [17:53<00:00, 18
6.33it/s]
Computing delta eigenvalues: 100%|██████████| 99995/99995 [04:47<00:00, 347.
23it/s]
```

In [31]:
```
X_val = data_pipe.transform(temp_validation["BTC-USD"])
```

```
Computing delta eigenvalues: 100%|██████████| 99995/99995 [05:06<00:00, 325.
98it/s]
```

Here we simply preprocess our features, in order to only have to calculate them once, as we will reuse the data multiple times for our various models.

In [32]:
```
y_train = generate_y(temp["BTC-USD"], DELTA)[['1sec_midprice']]
y_val = generate_y(temp_validation["BTC-USD"], DELTA)[['1sec_midprice']]
```

In [33]:
```
theo_val_preds = {}
for model in models:
    print(model)
    m = model
    m.fit(X_train, y_train)
    theos = m.predict(X_val)
    y_val['theo'] = theos
    sq_loss = (((y_val['1sec_midprice'] - y_val['theo'])/1_000_000)**2).mear
    print("\tSquared Error:", round(sq_loss, 4))
    theo_val_preds[model.__str__()] = (m, theos, sq_loss, theos)
```

```
LinearRegression()
        Squared Error: 0.2202
Lasso()
        Squared Error: 0.2198
Ridge()
        Squared Error: 0.2202
RandomForestRegressor(max_depth=5)
        Squared Error: 0.8451
AdaBoostRegressor(n_estimators=100)
        Squared Error: 1.1696
GradientBoostingRegressor()
        Squared Error: 0.3145
```

In [72]:
```
with open('data/output/theo_val_preds.pkl', 'wb') as f:
    pickle.dump(theo_val_preds, f)
```

Estimating the C Metric

please note: this metric ultimately was not used given that forecasting our own errors into the future is quite a laborious task, and as seen below it did not work!

In essence we define our prediction of C as;

$$\hat{\mathrm{C}}_t = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x + \ldots)}} \approx (\hat{\mathrm{M}}_t - \mathrm{M}_t)^2$$

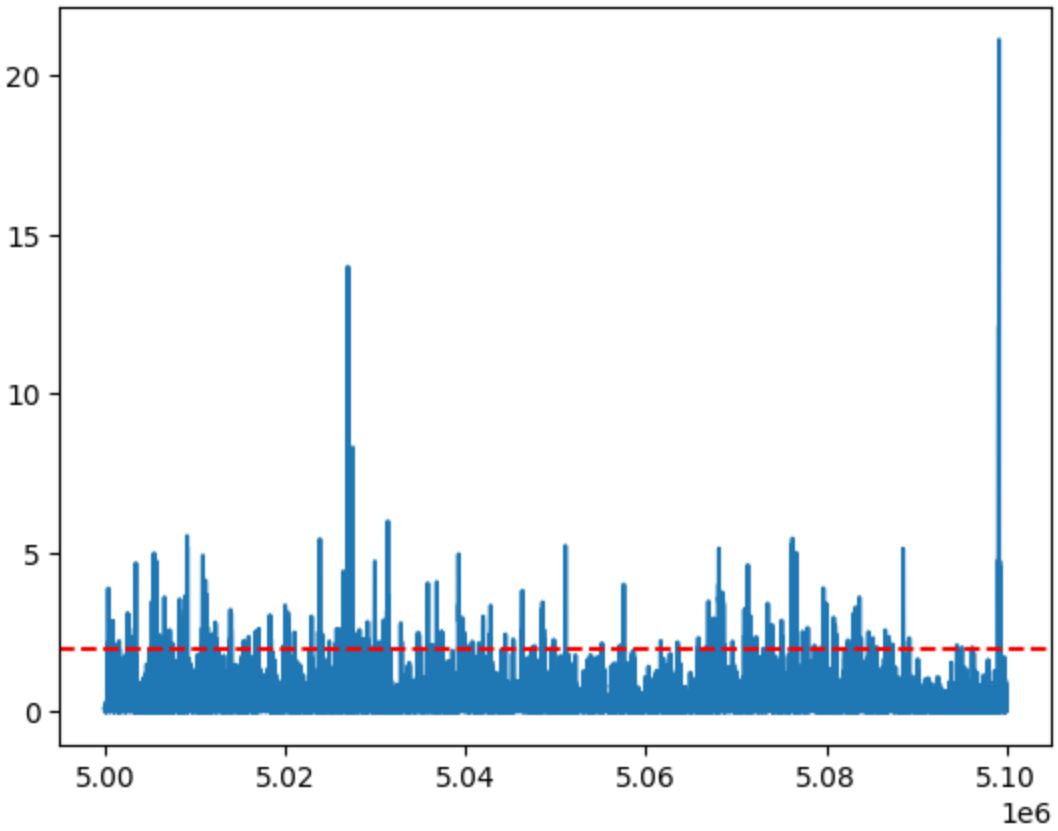
This is a logistic regression model over our residual errors. The idea behind this model was to find if there is a relationship between our squared error and the processed data.

In [39]:
```
temp_validation["BTC-USD"]["books"]["midprice"] = (temp_validation["BTC-USD"
temp_validation["BTC-USD"]["books"]["theo"] = theos
roll_pred_y = rolling_mean(temp_validation["BTC-USD"]["books"][["timestamp_u
```

In [ ]:
```
def plot_c_metric_tresh(tresh, squared_errors, title = "BTC-USD"):
    plt.figure(figsize=(15, 5))
    plt.plot(squared_errors)
    plt.axhline(y=tresh, color='r', linestyle='--')
    #below the line is good add text to the right
    plt.text(len(squared_errors), tresh, f"Treshold: {tresh}", ha='right', v
    plt.grid()
    plt.title(title)
    plt.xlabel("Treshold")
    plt.ylabel("Squared Error")
    plt.show()
```

In [40]:
```
plt.plot(((temp_validation["BTC-USD"]["books"]["midprice"] - temp_validatior
#show line at 0.25e+13
plt.axhline(y=2, color='r', linestyle='--')
```

Out[40]:
```
<matplotlib.lines.Line2D at 0x2c222a82660>
```



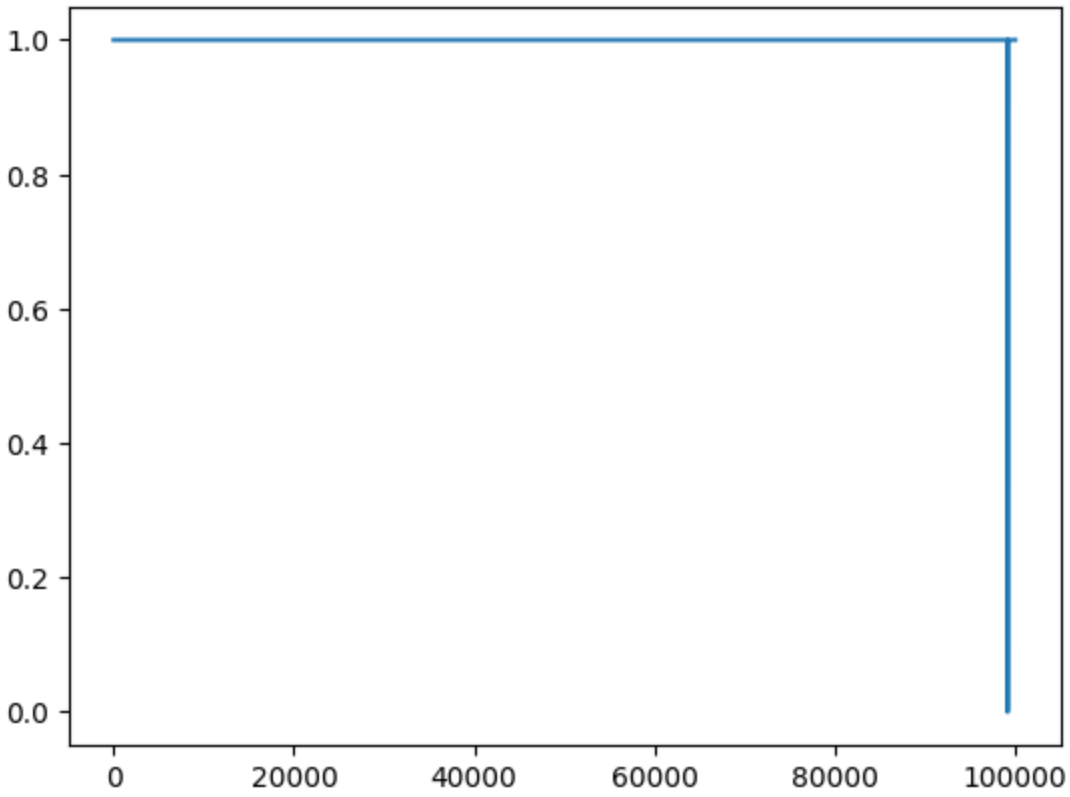
Any prediction below that line,is to be interpreted as a period where we deem a prediction "close enough" to the actual price of the BTC-USD pair 1 second into the future

In [41]:
```
correct_pred = (((roll_pred_y["midprice"] - roll_pred_y["theo"])/1_000_000)*
```

In [42]:
```
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression()
clf_confidence = make_model(feature_space, classifier)
confs = clf_confidence.fit(temp_validation["BTC-USD"], correct_pred)
confs = confs.predict(temp_validation["BTC-USD"])
```

In [60]:
```
confs.mean()
```

Out[60]:
```
[<matplotlib.lines.Line2D at 0x2c226928e90>]
```



We see above that our model seems to have no choice but to be confident in it's estimate, this signals that the model has to predict that it's theo estimate is good given that the information it is being given simply cannot explain the relationship between accuracy and features

Strategies

In this section we run our various models through our `MarketMakingStrategy` object. This object incorporates a lot of important parameters, among these we have:

- latency: the time we expect it to take an order to be processed. This includes the time that it takes for a new order to be added to the market, as well as the time it takes for a new order modification to be updated on the market.
- theo_offset_for_price_selection: a value that we subtract or add to our theoretical utility that gives us an actual price to place our limit order. Higher values mean that we will place our order further from our predicted utility theo, and smaller values mean that we will be more likely to place an order close to our utility theo.
- utility_difference_to_change_theo: we run our previous theo through our utility function to calculate the previous utility based on our current position. This metric is the absolute difference between our optimal utility theo for our current position and previous utility.
- price_difference_to_change_order: the absolute price difference we must observe between our a hypothetical new position and our current position in order to modify

an order.
- seconds_to_half_prob_of_trading: given that an appropriate trade is made, this metric is the amount of seconds required before we have half probability of our algorithm processing a trade as a successful order fill

The rationale behind using utility functions

Utility functions, are a common occurence in undergraduate economics microeconomic courses. Given our backgrounds, this felt like a natural approach to trading. In fact, most of the earliest models used for elaborating complex decisions by agents in the high frequency marketplace were based on microeconomic rationale. Though Stoikov wrote his own, and to his credit probably wrote something better than we did, we still wanted to write our own, as we tought this was a great way to learn about trading on a deeper level.

The utility function:

$$u(x) = i/k(t - x) - c(x - t)^2$$

x: Some price input, this is used namely i: The inventory (quantities held) c: Our level of confidence (this turns out to always be one given the lack of predictive power of the C model) t: The theo value generated by our model $M\ k$: Some constant to regulate the effect of inventory on the recommendation of the utility function

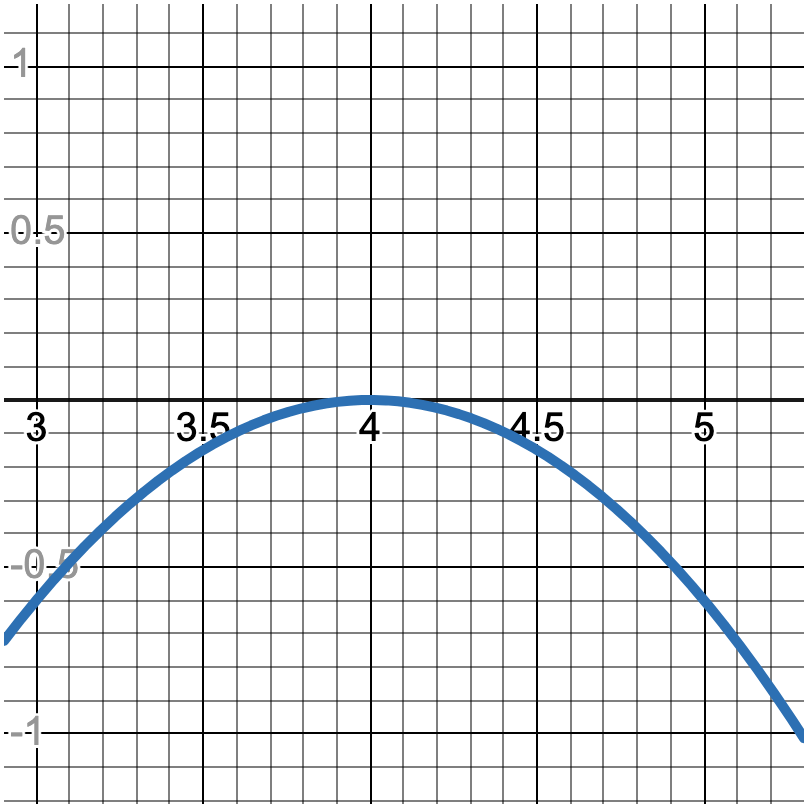
When we find:

$$\frac{\partial}{\partial x}u(x) = 0$$

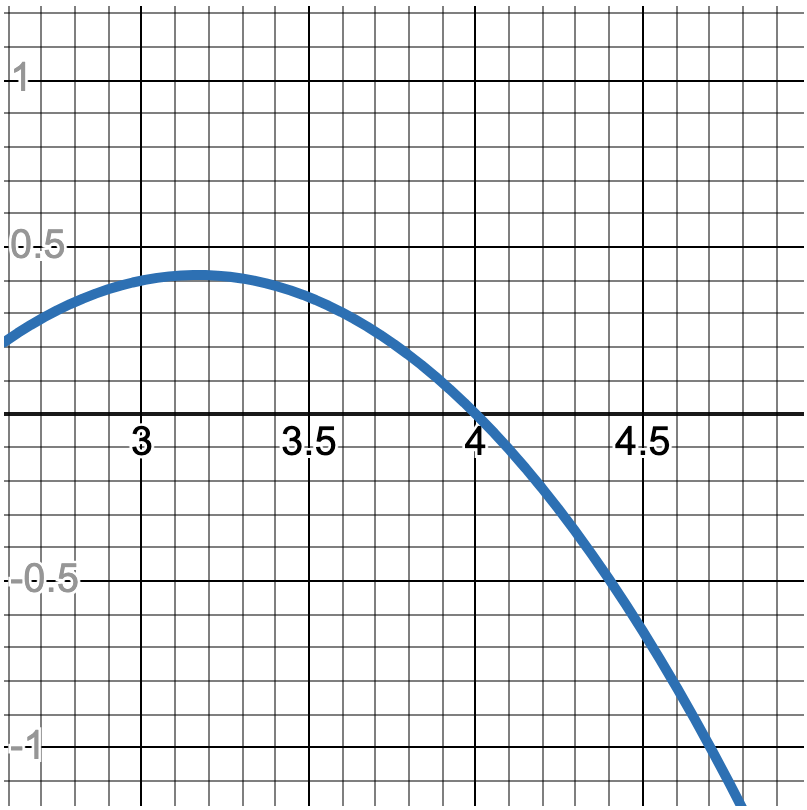
we get a "new" theo prices, which instead of simply looking at the markets, also takes into consideration our current situation. Below, we show an example of this:

Example:

Let the initial state be a market where we hold no quantity of the asset, we predict a theo of $4 and we are absolutely certain about it :(i = 0,c = 1,t = 4$)



Here the price x that maximizes our utility is $4, this makes sense sice we hold no position and therefore should trust our theo value and trade in accordance with that. Let's assume that we place an a bid order, and it gets filled. We therefore have an inventory of $\mathrm{i} = 1$, with $\mathrm{k} = 1$ in this scenario our new theo measure is:



In this case we observe that the theo recommended by our utility function has been adjusted to approximately 3.2. This makes sense, as holding a position implies some level of inventory risk, as the inventory increases the utility function keeps reducing the theo value. Eventually buying more units of the asset stops becoming interesting as the theo as changed the price at a level large enough to not be interesting anymore, given market dynamics.

In [ ]:
```
book = temp_validation["BTC-USD"]["books"].copy()
book['mid_price'] = (book['Ask1PriceMillionths'] + book['Bid1PriceMillionths

trades = temp_validation["BTC-USD"]["trades"].copy()
```

Ranking our Theos

(this is based on the squared error found during validation)

In [57]:
```
# 1st
validation_theos = theo_val_preds[GradientBoostingRegressor().__str__()][1]
# 2nd
validation_theos1 = theo_val_preds[LinearRegression().__str__()][1]
# 3rd
validation_theos2 = theo_val_preds[Lasso().__str__()][1]
# 4th
validation_theos3 = theo_val_preds[Ridge().__str__()][1]
# 5th
validation_theos4 = theo_val_preds[RandomForestRegressor(max_depth=5).__str_
# 6th
validation_theos5 = theo_val_preds[AdaBoostRegressor(n_estimators=100).__str
```

Our initial hyperparameters

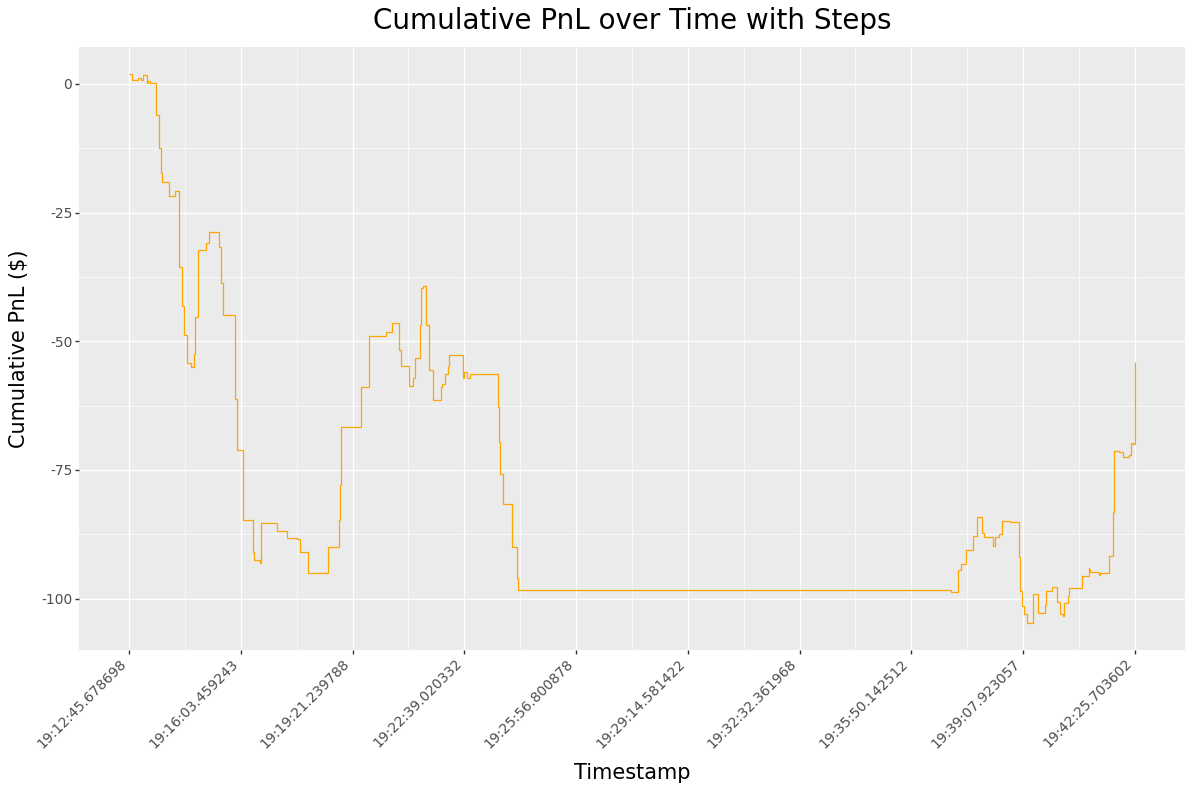
In [220…
```
# hyperparamters
latency = 1_000_000
theo_offset_for_price_selection = 200_000
utility_dfference_to_change_theo = 1
price_difference_to_change_order = 200_000
seconds_to_half_prob_of_trading = 0.5
```

We begin with some pre-chosen hyperparameters that were used during the initial testing of the strategy. These were simply to help us test that the strategy functioned as intended, and we will run simulations later to find optimal values.

In [221…
```
importlib.reload(mm)

mms = mm.MarketMakingStrategy(pd.DataFrame(validation_theos), pd.DataFrame(c

mms.run_strategy()
mms.pnl_plot.draw()
```

```
Running strategy: 100%|██████████| 100000/100000 [00:23<00:00, 4261.71it/s]
```

Out [221…



Clearly, our initial choices were not a great for actual performance, but it does help give us some insight that our strategy can be improved. The first observation we can make is that there is a long period where our PnL is entirely stagnant. Let us investigate to see if this is a product of our strategy or market conditions.

In [222…

```
df = trades.copy()

df.timestamp_utc_nanoseconds = pd.to_datetime(df.timestamp_utc_nanoseconds)

# No trading activity for 12 minutes
df[df['timestamp_utc_nanoseconds'] >= '2023-01-24 19:24:22']
```

Out [222…

	Price	PriceMillionths	Side	Size	SizeBillionths	received_utc	rece
323263	23009.00	23009000000	-1	0.001600	1600000	1.674588e+09	
323264	23007.18	23007180000	1	0.005103	5102530	1.674589e+09	
323265	23006.29	23006290000	-1	0.000047	47280	1.674589e+09	
323266	23007.17	23007170000	1	0.001000	1000000	1.674589e+09	
323267	23007.18	23007180000	1	0.062579	62579470	1.674589e+09	
…	…	…	…	…	…	…	
325003	23005.00	23005000000	-1	0.050000	50000000	1.674589e+09	
325004	23005.00	23005000000	-1	0.219605	219605050	1.674589e+09	
325005	23005.00	23005000000	-1	0.050000	50000000	1.674589e+09	
325006	23005.00	23005000000	-1	0.050000	50000000	1.674589e+09	
325007	23005.00	23005000000	-1	0.050000	50000000	1.674589e+09	

1745 rows × 9 columns

As we can see, there is a massive jump from in timestamp within the sample. Looking at the book data, we also observed that this pause was present. This means that either our data is incomplete and lacks the information from this period, or there was a market freeze. We suspect this is a market freeze given that the price jump is not astronomical after trading begins again, which would warrant further investigation into the market during this specific time frame.

Plotting Inventory Comapred to Book Data

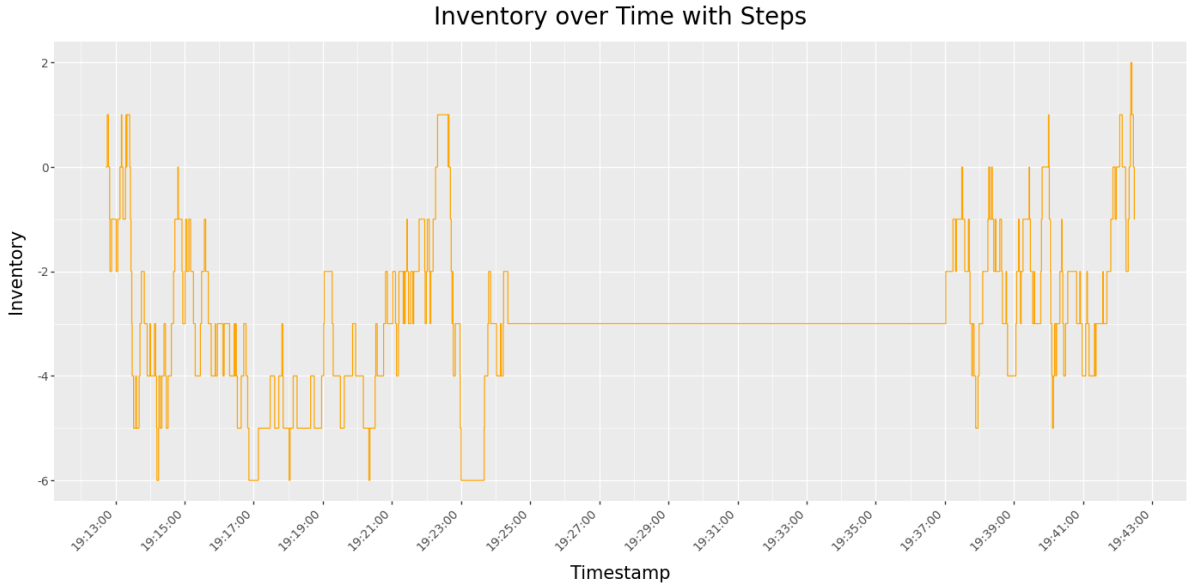
In [224…

```
df = mms.action_log.copy()

inventory_plot = create_plot(df=df,timestamp_col='timestamp',y_col='inventor

df1 = book.copy()
df1['mid_price'] = df1['mid_price'] / 1_000_000

mid_price_plot = create_plot(df=df1,timestamp_col='timestamp_utc_nanoseconds
```

In [225…

```
inventory_plot.draw()
```

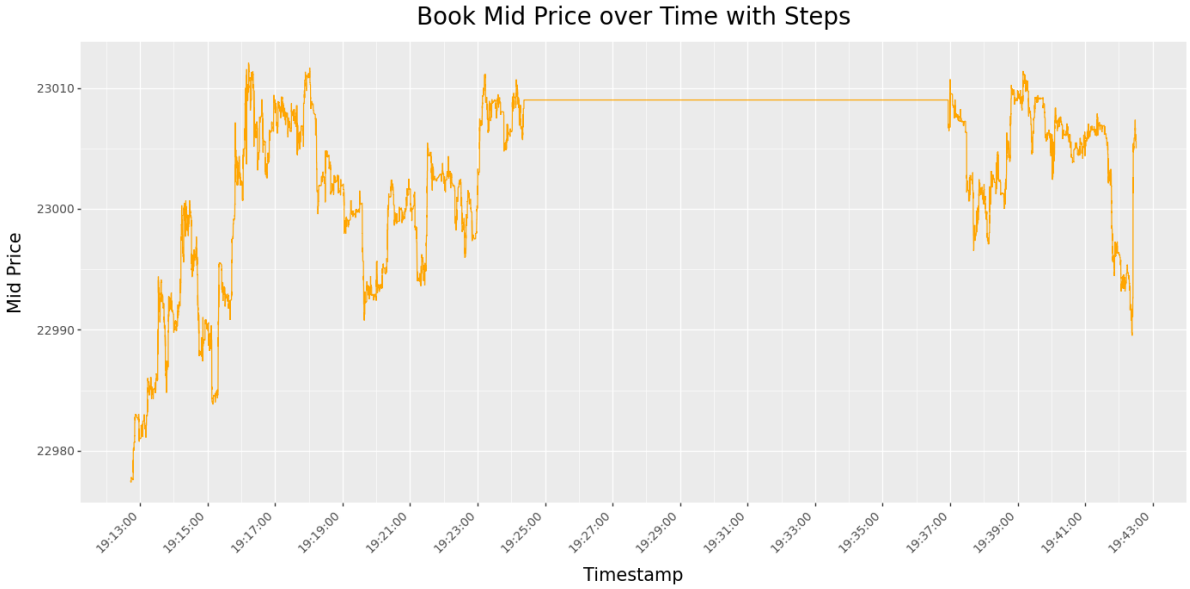
Out [225…



In [226…

```
mid_price_plot.draw()
```

Out [226…



Note that there are clearly some periods of similarlity between the book mid-price and strategy inventory fluctuations. As we should expect, as price increases in the observed mid-price, we get filled on more of our ask orders (and vice versa). Since the price saw a lot of increases during our sample, we hold a negative position for most of the period.

Optimal Hyperparameter Simulation

In [227…

```
# Testing different hyperparameters
latency_list = [1_000_000]
theo_offset_for_price_selection_list = [0, 25_000, 50_000, 100_000]
utility_dfference_to_change_theo_list = [0, 0.25, 0.5, 1]
price_difference_to_change_order_list = [0, 100_000, 200_000, 500_000]
seconds_to_half_prob_of_trading_list = [0.5]
```

Note that we are going to keep latency and half_prob constant, testing the other 3 hyperparameters for optimal values. This is because we should expect higher latencies and higher half_prob values to decrease PnL. If we take longer to execute on information, or if we find that we do not get filled as often because our chance of getting filled is lower (remember that higher half_prob values indicate lower probabilities of being filled with less time), then we should expect (with good information) to make less profit.

In [228…
```
def collect_strategies(latency_list, theo_offset_for_price_selection_list, u
    mms_strategy_list = []
    param_combinations = itertools.product(latency_list, theo_offset_for_pri

    for combination in param_combinations:
        latency, theo_offset, utility_difference, price_difference, seconds_

        mms = mm.MarketMakingStrategy(pd.DataFrame(theos), pd.DataFrame(conf

        mms.run_strategy()

        mms_strategy_list.append((combination, mms))

        print(f'strategy {combination} complete')

    return mms_strategy_list
```

In [87]:
```
importlib.reload(mm)

mms_strategy_list = collect_strategies(
    latency_list,
    theo_offset_for_price_selection_list,
    utility_dfference_to_change_theo_list,
    price_difference_to_change_order_list,
    seconds_to_half_prob_of_trading_list,
    validation_theos, confs, book, trades
)
```

```
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4662.48it/s]
strategy (1000000, 0, 0, 0, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4739.23it/s]
strategy (1000000, 0, 0, 100000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4728.64it/s]
strategy (1000000, 0, 0, 200000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4711.57it/s]
strategy (1000000, 0, 0, 500000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4746.08it/s]
strategy (1000000, 0, 0.25, 0, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4794.69it/s]
strategy (1000000, 0, 0.25, 100000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4808.94it/s]
strategy (1000000, 0, 0.25, 200000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4714.27it/s]
```

```
strategy (1000000, 0, 0.25, 500000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4751.21it/s]
strategy (1000000, 0, 0.5, 0, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4633.03it/s]
strategy (1000000, 0, 0.5, 100000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:22<00:00, 4449.47it/s]
strategy (1000000, 0, 0.5, 200000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4551.00it/s]
strategy (1000000, 0, 0.5, 500000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4753.02it/s]
strategy (1000000, 0, 1, 0, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4794.23it/s]
strategy (1000000, 0, 1, 100000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4779.99it/s]
strategy (1000000, 0, 1, 200000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4813.01it/s]
strategy (1000000, 0, 1, 500000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4730.03it/s]
strategy (1000000, 25000, 0, 0, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4651.70it/s]
strategy (1000000, 25000, 0, 100000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4663.09it/s]
strategy (1000000, 25000, 0, 200000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4744.34it/s]
strategy (1000000, 25000, 0, 500000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4783.67it/s]
strategy (1000000, 25000, 0.25, 0, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4771.34it/s]
strategy (1000000, 25000, 0.25, 100000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4766.37it/s]
strategy (1000000, 25000, 0.25, 200000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4814.20it/s]
strategy (1000000, 25000, 0.25, 500000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4815.07it/s]
strategy (1000000, 25000, 0.5, 0, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4836.76it/s]
strategy (1000000, 25000, 0.5, 100000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4771.31it/s]
strategy (1000000, 25000, 0.5, 200000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4777.32it/s]
strategy (1000000, 25000, 0.5, 500000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4846.85it/s]
strategy (1000000, 25000, 1, 0, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4745.51it/s]
strategy (1000000, 25000, 1, 100000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4807.29it/s]
strategy (1000000, 25000, 1, 200000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4838.30it/s]
strategy (1000000, 25000, 1, 500000, 0.5) complete
```

```
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4737.48it/s]
strategy (1000000, 50000, 0, 0, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4763.61it/s]
strategy (1000000, 50000, 0, 100000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4792.23it/s]
strategy (1000000, 50000, 0, 200000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4771.09it/s]
strategy (1000000, 50000, 0, 500000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4820.37it/s]
strategy (1000000, 50000, 0.25, 0, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4776.74it/s]
strategy (1000000, 50000, 0.25, 100000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4797.72it/s]
strategy (1000000, 50000, 0.25, 200000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4824.30it/s]
strategy (1000000, 50000, 0.25, 500000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4835.05it/s]
strategy (1000000, 50000, 0.5, 0, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4807.48it/s]
strategy (1000000, 50000, 0.5, 100000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4796.32it/s]
strategy (1000000, 50000, 0.5, 200000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4840.98it/s]
strategy (1000000, 50000, 0.5, 500000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4771.06it/s]
strategy (1000000, 50000, 1, 0, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4830.57it/s]
strategy (1000000, 50000, 1, 100000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4850.41it/s]
strategy (1000000, 50000, 1, 200000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4772.63it/s]
strategy (1000000, 50000, 1, 500000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4728.71it/s]
strategy (1000000, 100000, 0, 0, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4759.00it/s]
strategy (1000000, 100000, 0, 100000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4819.22it/s]
strategy (1000000, 100000, 0, 200000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4806.26it/s]
strategy (1000000, 100000, 0, 500000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4730.30it/s]
strategy (1000000, 100000, 0.25, 0, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4678.60it/s]
strategy (1000000, 100000, 0.25, 100000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4662.33it/s]
strategy (1000000, 100000, 0.25, 200000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4747.39it/s]
strategy (1000000, 100000, 0.25, 500000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4812.52it/s]
```

```
strategy (1000000, 100000, 0.5, 0, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4780.01it/s]
strategy (1000000, 100000, 0.5, 100000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4716.45it/s]
strategy (1000000, 100000, 0.5, 200000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4771.92it/s]
strategy (1000000, 100000, 0.5, 500000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4749.45it/s]
strategy (1000000, 100000, 1, 0, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:20<00:00, 4790.84it/s]
strategy (1000000, 100000, 1, 100000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4751.80it/s]
strategy (1000000, 100000, 1, 200000, 0.5) complete
Running strategy: 100%|██████████| 100000/100000 [00:21<00:00, 4758.33it/s]
strategy (1000000, 100000, 1, 500000, 0.5) complete
```

In [95]:
```python
def create_info_df(strategies_list):
    data = []

    for tup in strategies_list:
        combination, strategy = tup
        pnl = strategy.pnl_log['cumulative_pnl'].iloc[-1]
        avg_square_inv = (strategy.action_log["inventory"]**2).mean()
        num_trades = len(strategy.trade_log)

        latency, theo_offset, utility_dfference, price_difference, half_prob

        data.append({
            "strategy": strategy,
            "pnl": pnl,
            "avg_squared_inv": avg_square_inv,
            "num_trades": num_trades,
            "theo_offset": theo_offset,
            "utility_difference": utility_dfference,
            "price_difference": price_difference,
        })

    df = pd.DataFrame(data)

    return df
```

In [97]:
```python
strategy_info_df = create_info_df(mms_strategy_list)
strategy_info_df_sorted = strategy_info_df.sort_values(by='pnl', ascending=F
```

It is just as important to find which values of hyperparameters perform poorly as it is to look for the values that perform well. If we naively took the top PnL value, we could be selecting an outlier event since we are working on a limit sample. Let us then observe the best and worst 10 strategies based on their PnL.

In [99]:
```python
display(strategy_info_df_sorted.head(10))
display(strategy_info_df_sorted.tail(10))
```

	strategy	pnl	avg_squared_inv	num_trades	theo_offset	utility_difference
18	Strategy with combo (1000000, 25000, 0, 200000...	143.478745	37.475935	177	25000	0.00
19	Strategy with combo (1000000, 25000, 0, 500000...	105.012162	29.205836	254	25000	0.00
33	Strategy with combo (1000000, 50000, 0, 100000...	104.290537	23.988110	153	50000	0.00
21	Strategy with combo (1000000, 25000, 0.25, 100...	98.052709	42.390878	150	25000	0.25
23	Strategy with combo (1000000, 25000, 0.25, 500...	96.066318	36.822237	244	25000	0.25
17	Strategy with combo (1000000, 25000, 0, 100000...	83.940238	36.414729	152	25000	0.00
34	Strategy with combo (1000000, 50000, 0, 200000...	69.967404	25.290898	173	50000	0.00
37	Strategy with combo (1000000, 50000, 0.25, 100...	55.922470	29.969849	145	50000	0.25

	strategy	pnl	avg_squared_inv	num_trades	theo_offset	utility_difference
42	Strategy with combo (1000000, 50000, 0.5, 2000...	51.885477	27.796105	202	50000	0.5(
58	Strategy with combo (1000000, 100000, 0.5, 200...	49.529332	17.555385	186	100000	0.5(

	strategy	pnl	avg_squared_inv	num_trades	theo_offset	utility_differenc
32	Strategy with combo (1000000, 50000, 0, 0, 0.5)	-137.241663	53.114302	41	50000	0.0
13	Strategy with combo (1000000, 0, 1, 100000, 0.5)	-160.270617	2856.430286	366	0	1.0
24	Strategy with combo (1000000, 25000, 0.5, 0, 0.5)	-161.604828	78.661287	101	25000	0.5
36	Strategy with combo (1000000, 50000, 0.25, 0, ...	-162.133510	56.906347	57	50000	0.2
16	Strategy with combo (1000000, 25000, 0, 0, 0.5)	-177.826717	74.424720	43	25000	0.0
20	Strategy with combo (1000000, 25000, 0.25, 0, ...	-186.662984	84.105715	58	25000	0.2
0	Strategy with combo (1000000, 0, 0, 0, 0.5)	-272.729762	4596.599923	160	0	0.0
12	Strategy with combo (1000000, 0, 1, 0, 0.5)	-330.977767	4856.426383	378	0	1.0

	strategy	pnl	avg_squared_inv	num_trades	theo_offset	utility_differenc
4	Strategy with combo (1000000, 0, 0.25, 0, 0.5)	-389.540433	7620.839367	213	0	0.2
8	Strategy with combo (1000000, 0, 0.5, 0, 0.5)	-466.770298	9930.315373	272	0	0.5

Note that theo_offset set to 0 leads to a huge inventory because of the way the market making strategy implements this parameter. It multiplies theo_offset by an exponential function of inventory, so when it is 0 then this multiplication is irrelevant. This is likely why 0 theo_offset performs poorly.

Also, a price difference of 0 performs incredibly poorly as well, which is likely due to the fact that we are constantly changing our orders. Sometimes we don't have a drastic enough predicted theo change to warrant a an actual order modification.

In [104…
```
# Correlation across all data points
strategy_info_df_sorted.iloc[:, 1:].corr()['pnl']
```

Out[104…
```
pnl                  1.000000
avg_squared_inv     -0.724001
num_trades           0.005617
theo_offset          0.303707
utility_difference  -0.148931
price_difference     0.426338
Name: pnl, dtype: float64
```

In [105…
```
# Correlation across data points that have non-zero theo_offset
strategy_info_df_sorted[strategy_info_df_sorted['theo_offset'] > 0].iloc[:,
```

Out[105…
```
pnl                  1.000000
avg_squared_inv     -0.672034
num_trades           0.473142
theo_offset         -0.063769
utility_difference  -0.173654
price_difference     0.497960
Name: pnl, dtype: float64
```

Notice that removing our theo_offset values of 0 changes the PnL/theo_offset correlation from a medium positive to weak negative. This begins to indicate that (as long as theo_offset is not zero), it is more advantageous to have a smaller theo_offset value. Let us examine the worst 10 strategies given that our theo_offset is not zero.

In [108…

```
# We need to filter out the price_difference of 0
strategy_info_df_sorted[strategy_info_df_sorted['theo_offset'] > 0].tail(10)
```

Out[108...

	strategy	pnl	avg_squared_inv	num_trades	theo_offset	utility_differe
28	Strategy with combo (1000000, 25000, 1, 0, 0.5)	-87.363980	64.121857	259	25000	1
56	Strategy with combo (1000000, 100000, 0.5, 0, ...	-102.904430	36.404190	99	100000	0
40	Strategy with combo (1000000, 50000, 0.5, 0, 0.5)	-127.407208	58.091984	100	50000	0
48	Strategy with combo (1000000, 100000, 0, 0, 0.5)	-135.861783	35.864506	40	100000	0
52	Strategy with combo (1000000, 100000, 0.25, 0,...	-136.400949	41.700130	55	100000	0
32	Strategy with combo (1000000, 50000, 0, 0, 0.5)	-137.241663	53.114302	41	50000	0
24	Strategy with combo (1000000, 25000, 0.5, 0, 0.5)	-161.604828	78.661287	101	25000	0
36	Strategy with combo (1000000, 50000, 0.25, 0, ...	-162.133510	56.906347	57	50000	0

	strategy	pnl	avg_squared_inv	num_trades	theo_offset	utility_differe
16	Strategy with combo (1000000, 25000, 0, 0, 0.5)	-177.826717	74.424720	43	25000	0
20	Strategy with combo (1000000, 25000, 0.25, 0, ...	-186.662984	84.105715	58	25000	0

Clearly, a price difference of 0 is not advantageous. Even when we remove the theo_offset value of 0, we see that a price difference of 0 performs incredibly poorly. Therefore, let us also filter out this hyperparameter value.

In [112…
```
# Worst performing strategies that have sensible theo_offset and price_diffe
strategy_info_df_sorted[(strategy_info_df_sorted['theo_offset'] > 0) & (stra
```

Out[112...

	strategy	pnl	avg_squared_inv	num_trades	theo_offset	utility_differen
46	Strategy with combo (1000000, 50000, 1, 200000...	-1.805178	32.813144	283	50000	
41	Strategy with combo (1000000, 50000, 0.5, 1000...	-2.096415	31.715548	186	50000	
57	Strategy with combo (1000000, 100000, 0.5, 100...	-6.273968	20.631346	171	100000	
25	Strategy with combo (1000000, 25000, 0.5, 1000...	-6.773794	45.733826	187	25000	
27	Strategy with combo (1000000, 25000, 0.5, 5000...	-9.340851	36.216519	273	25000	
45	Strategy with combo (1000000, 50000, 1, 100000...	-13.037374	30.524429	264	50000	
61	Strategy with combo (1000000, 100000, 1, 10000...	-14.075839	22.369188	256	100000	
62	Strategy with combo (1000000, 100000, 1, 20000...	-40.986231	22.478904	269	100000	

	strategy	pnl	avg_squared_inv	num_trades	theo_offset	utility_differen
63	Strategy with combo (1000000, 100000, 1, 50000...	-45.158944	18.914453	291	100000	
29	Strategy with combo (1000000, 25000, 1, 100000...	-45.389070	49.467782	268	25000	

Utility difference has no values of 0 in the worst 10 stratgeies where price difference and theo_offset are non-zero. We also did some testing outside this document that confirmed that utility difference is not beneficial as a non-zero value. Therefore, we will just set this value as 0 instead.

In [114…

```
best_params_matrix = strategy_info_df_sorted[(strategy_info_df_sorted['theo_
                        (strategy_info_df_sorted['price_difference'] > 0) &
                        (strategy_info_df_sorted['utility_difference'] == 0)
best_params_matrix
```

Out[114...

	strategy	pnl	avg_squared_inv	num_trades	theo_offset	utility_differen
18	Strategy with combo (1000000, 25000, 0, 200000...	143.478745	37.475935	177	25000	0
19	Strategy with combo (1000000, 25000, 0, 500000...	105.012162	29.205836	254	25000	0
33	Strategy with combo (1000000, 50000, 0, 100000...	104.290537	23.988110	153	50000	0
17	Strategy with combo (1000000, 25000, 0, 100000...	83.940238	36.414729	152	25000	0
34	Strategy with combo (1000000, 50000, 0, 200000...	69.967404	25.290898	173	50000	0
50	Strategy with combo (1000000, 100000, 0, 20000...	37.998457	14.272147	145	100000	0
51	Strategy with combo (1000000, 100000, 0, 50000...	34.598672	16.735136	225	100000	0
35	Strategy with combo (1000000, 50000, 0, 500000...	29.782454	21.266500	237	50000	0

	strategy	pnl	avg_squared_inv	num_trades	theo_offset	utility_differen
49	Strategy with combo (1000000, 100000, 0, 10000...	25.127976	15.929089	130	100000	0

In [115…
```
best_params_matrix.iloc[:, 1:].corr()['pnl']
```

Out[115…
```
pnl                   1.000000
avg_squared_inv       0.837835
num_trades            0.034938
theo_offset          -0.789693
utility_difference         NaN
price_difference     -0.212869
Name: pnl, dtype: float64
```

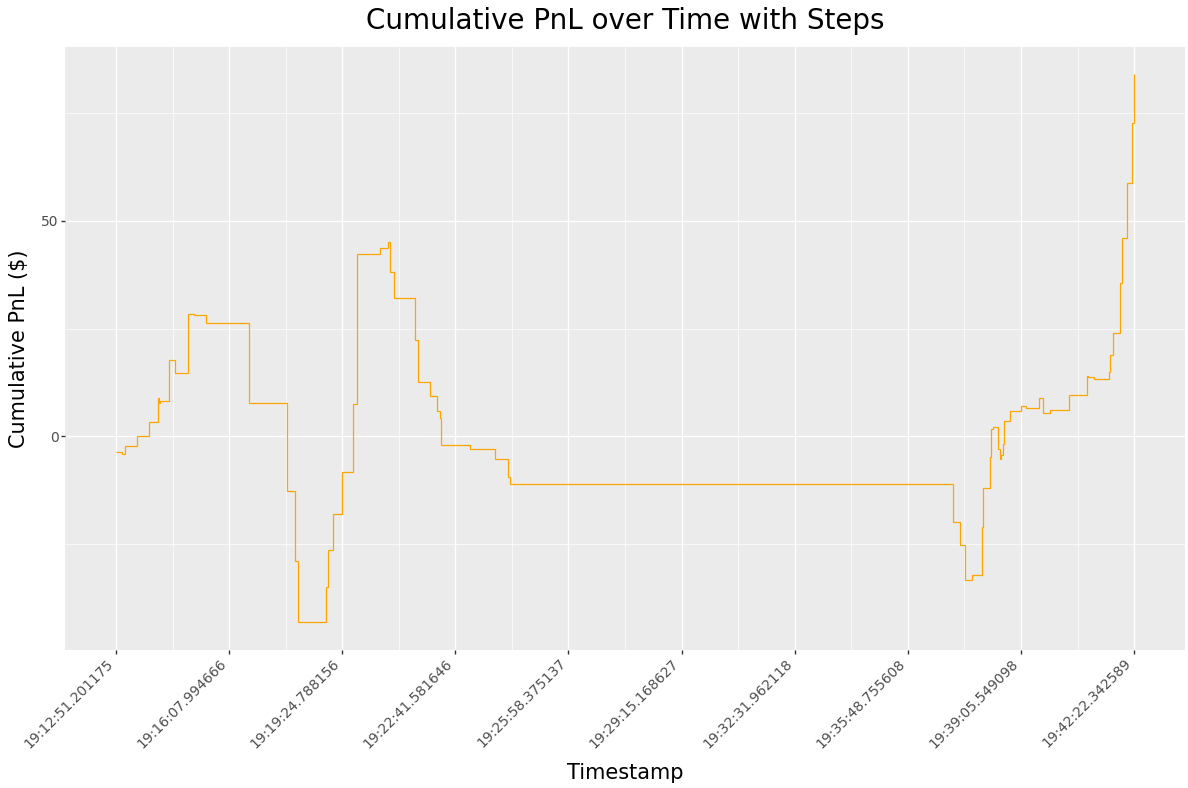
It seems like we should use a lower value of theo_offset and price difference given our parameter constraints. Therefore, we will choose our optimal parameters as theo_offset = 25_000, optimal utility_difference = 0, and optimal price_difference = 100_000.

In [182…
```
# optimal_params
latency = 1_000_000
opt_theo_offset = 25_000
opt_utility_difference = 0
opt_price_difference = 100_000
half_prob = 0.5
```

In [183…
```
importlib.reload(mm)

optimal_mms = mm.MarketMakingStrategy(pd.DataFrame(validation_theos), pd.Dat

optimal_mms.run_strategy()
optimal_mms.pnl_plot.draw()
```

```
Running strategy:   0%|          | 0/100000 [00:00<?, ?it/s]
Running strategy: 100%|██████████| 100000/100000 [00:22<00:00, 4350.74it/s]
```

Out [183…



This plot does not have the best PnL of all the strategies, but our optimal values are not naively chosen. Therefore, we can be confident that our optimal hyperparameters are chosen with quantitative reasoning, and therefore can be confident that this PnL is a valid observation.

Analyzing the effects of latency and half_prob

In [184…

```
latency1 = 0
latency2 = 100_000
latency3 = 500_000
latency4 = 2_000_000
```

In [191…

```
def run_strategy_for_latencies(latency_values):
    latency_list = []
    for latency in latency_values:
        mms = mm.MarketMakingStrategy(pd.DataFrame(validation_theos), pd.Dat

        mms.run_strategy()
        latency_list.append(mms)

    return latency_list
```

In [186…

```
latency_list = run_strategy_for_latencies([latency1, latency2, latency3, lat
```

```
Running strategy: 100%|██████████| 100000/100000 [00:23<00:00, 4172.01it/s]
Running strategy: 100%|██████████| 100000/100000 [00:23<00:00, 4194.68it/s]
Running strategy: 100%|██████████| 100000/100000 [00:23<00:00, 4314.44it/s]
Running strategy: 100%|██████████| 100000/100000 [00:23<00:00, 4328.15it/s]
```

In [208…
```
def pnl_info_df(lst):
    data = []

    for strat in lst:
        pnl = strat.pnl_log['cumulative_pnl'].iloc[-1]
        avg_square_inv = (strat.action_log["inventory"]**2).mean()
        num_trades = len(strat.trade_log)

        data.append({
            "pnl": pnl,
            "avg_squared_inv": avg_square_inv,
            "avg_inv": strat.action_log['inventory'].mean(),
            "num_trades": num_trades,
            "latency": strat.latency,
            "half_prob": strat.half_prob
        })

    df = pd.DataFrame(data)

    return df
```

In [209…
```
pnl_info_df(latency_list)
```

Out[209…

	pnl	avg_squared_inv	avg_inv	num_trades	latency	half_prob
0	100.203212	38.198134	-5.340789	144	0	0.5
1	104.605912	35.889068	-5.012667	150	100000	0.5
2	81.370238	37.343795	-5.182257	150	500000	0.5
3	83.795319	36.028016	-5.062826	152	2000000	0.5

Here we observe the effects of latency on PnL, we see that from instantaneous trading our PnL seems quite high. Changing our latency from 0 nanoseconds to 0.1 milliseconds doesn't seem to have a significant impact on our PnL, which is likely due to the fact that little to no market participants are able to place their trades at such a speed on the crypto exchange. We see a drastic reduction in our PnL once we reach 0.5 milliseconds. Our strategy seems to be quite latency sensitive, which is something to keep in mind when exploring the actual latency times on crypto markets.

In [194…
```
def run_strategy_for_half_probs(half_prob_values):
    half_prob_list = []
    for half_prob in half_prob_values:
        mms = mm.MarketMakingStrategy(pd.DataFrame(validation_theos), pd.Dat

        mms.run_strategy()
        half_prob_list.append(mms)

    return half_prob_list
```

In [202…
```
half_prob1 = 0
half_prob2 = 0.1
```

```
half_prob3 = 0.25
half_prob4 = 1
```

In [203…
```
half_prob_list = run_strategy_for_half_probs([half_prob1, half_prob2, half_p
```

```
Running strategy: 100%|██████████| 100000/100000 [00:24<00:00, 4117.79it/s]
Running strategy: 100%|██████████| 100000/100000 [00:23<00:00, 4211.59it/s]
Running strategy: 100%|██████████| 100000/100000 [00:24<00:00, 4150.94it/s]
Running strategy: 100%|██████████| 100000/100000 [00:23<00:00, 4328.10it/s]
```

In [210…
```
pnl_info_df(half_prob_list)
```

Out[210…

	pnl	avg_squared_inv	avg_inv	num_trades	latency	half_prob
0	145.392945	46.197177	-6.176364	487	1000000	0.00
1	85.757361	45.148749	-6.348934	335	1000000	0.10
2	105.442284	46.060249	-6.148827	244	1000000	0.25
3	172.663258	34.663612	-3.679746	58	1000000	1.00

Interestingly, we see that half_prob has its best PnL performance at the highest value tested. This means that when we are forced to wait longer to have our order filled, we acutally see a better PnL as a result. Looking at the mid-price book data from before, we notice that this is likely due to the fact that there are rapid spikes to price during the sample period. This indicates that implementing a half_prob value of 1 could pose a psuedo-look ahead bias, as we cannot predict whether future crypto markets will have this same price fluctuation. Since we notice that our half_prob of 0 to 0.25 is decreasing, we can likely assume that it is better to have your orders filled more quickly.

Risk metrics

In [218…
```
calculate_risk_metrics(optimal_mms.pnl_log['cumulative_pnl'].to_frame())
```

Out[218…

	Metric	Value
0	Maximum Drawdown	-78.451419
1	VaR	-29.990576
2	CVaR	-35.910101
3	Sharpe Ratio	0.329524
4	Avg PnL	7.806889

This indicates that the strategy performance has a high standard deviation, which could pose issues with reliability of PnL. However, we do see that our maximum drawdown is not too bad given the overall PnL produced by the strategy, and VaR/CVaR metrics for

95% are reasonable. Let us observe our performance 80% of the time to make some final remarks on the strategy.

Risk metrics of 80%

In [219…
```
calculate_risk_metrics(optimal_mms.pnl_log['cumulative_pnl'].to_frame(), 0.8
```

Out[219…

	Metric	Value
0	Maximum Drawdown	-78.451419
1	VaR	-6.424423
2	CVaR	-22.419527
3	Sharpe Ratio	0.329524
4	Avg PnL	7.806889

Our CVaR value is unfortunately not positive for our 80-percentile value, but it is also not incredibly negative. This indicates that our strategy produces a positive PnL for a good portion of the time it is active, which is always beneficial.

Take-Aways

In this analysis we have learned quite a few things about building a backtesting system for high frequency trading. Regarding predicting Theo values, we learned the following:

- Prices can be hard to predict, and even good theo values can be difficult to trade against.
- Some hyperparameters that you test don't end up being beneficial. It is just as important to remove poor performing metrics as it is to introduce ones that help the strategy.
- Creating a market making strategy requires a combination of careful order placemenet because you want to remain as market neutral as possible. We found that always modifying our orders could lead to huge inventory holdings, which would be antithesis to the market making strategy.

Future Development

In the future, we intend to continue working on some of the shortcomings of the strategy. Particularly, we want to find accurate latency and fill probability values based on real market data, as these two metrics are going to be incredibly impactful on PnL. Additionally, we can continue to develop our confidence predictions using high level

statistical knowledge and technqiues, as our current model implies that our utility function always predicts our current theo prediction.

References

[1] Ait-Sahalia, Yacine and Mykland, Per A. and Zhang, Lan, Ultra High Frequency Volatility Estimation with Dependent Microstructure Noise (May 2005). NBER Working Paper No. w11380, Available at SSRN: https://ssrn.com/abstract=731035

[2] Bouchaud, J.P., Mézard, M., Potters, M., 2008. Statistical properties of stock order books: empirical results and models. arXiv preprint cond-mat/0203511

[3] M. Avellaneda and S. Stoikov. High-frequency trading in a limit order book. Quantitative Finance, 8(3):217–224, 2008.