This is the outline of the Python programme written for the analysis of the log files and price data generated by the IMC Prosperity 2 challenge.

The development of the trading algorithms is done in collaboration with Vincent Betschart (`@ltV100` on GitHub).

1 Background info

The details of the challenge are outlined in the Prosperity Wiki. The relevant page is Writing an Algorithm in Python, where the structure of the trading algorithm and the mechanism of the market exchange are outlined. The file `datamodel.py` is taken from the Wiki.

Upon the submission of a working algorithm and hence the completion of a round, the Prosperity server outputs a log file. Each log file includes:

1. **Sandbox log**, the output from AWS Lambda.
2. **Activities log**, the information of the market activities on that day at each timestamp. For example, this includes the top 3 bid and ask prices and volumes, and the pnl accrued by the algorithm at that timestamp.
3. **Trade history**, which includes information of all trades carried out by all market participants on that day, including those involving the algorithm.

2 Repo structure

The `data` folder includes the log files, as well as the market activities from the **Activities log** which have been processed into csv files (named "`trades_day_n.log`" or "`prices_day_n.csv`" for the nth day).

2.1 `utils.py`

This file includes the utility functions used to analyse the performance of the trading algorithm. These include:

Utility functions to analyse the Prosperity log files:

1. `get_tradehistory`
   Extracts the trade history from the log file in the form of a pandas Dataframe.
2. `get_mytrades`
   Takes the trade history Dataframe and product name as inputs. Selects

the trades involving the algorithm for the specified product from the trade history.

Functions to analyse the performance of the trading algorithm:

1. `get_pnl`
   Takes the Dataframe returned by `get_mytrades`, timestamps and a Dataframe containing the market price at each timestamp as inputs.

   The calculation of the pnl at every timestamp is as follows:

   (a) Check if a previously-open position was closed. For the closed position, calculate the realised pnl as the product of price difference (sell−buy) and quantity.

   (b) Calculate unrealised pnl as the product of open position and price difference with the market price [1] mp (if short, price=sell−mp; if long, price=mp−buy).

   (c) Add own trades done at that timestamp to the list of open positions.

   The total pnl is then the sum of realised and unrealised pnl.

   Below an example of the pnl calculated using the function, compared against the pnl returned by the Prosperity server.

2. `aggregate_trades`
   Given a list of `Trade` objects, aggregate all trades with the same nature (buy/sell) and the same price into one entry.

2.2 `exchange.py`

The figure below shows the mechanism of the interaction between the algorithm (class `Trader`) and the market (class `Exchange`) at every timestep.

The process of order matching by the exchange (function `Exchange.match()`) is implemented as described here (with an example) and shown in the Fig. 3 [2].

[1] The market price is defined as the midpoint between the best ask and the best bid at that timestamp.

[2] At the end of an iteration, all outstanding market orders, together with new market orders, are fed into the algorithm in the form of a `TradingState` object. However, upon inspection of the price data from Prosperity and the Wiki example, it seems like the top 3 bid/ask prices at each timestamp (from the csv) do not necessarily agree with the outstanding market quotes from the previous timestamp. In this case, we decided to prioritise the values given in the price data, and consider the outstanding market quotes from the previous timestamp only if they do not fall in the top 3. This ensures that the top 3 offers from the csv get executed with higher priority.

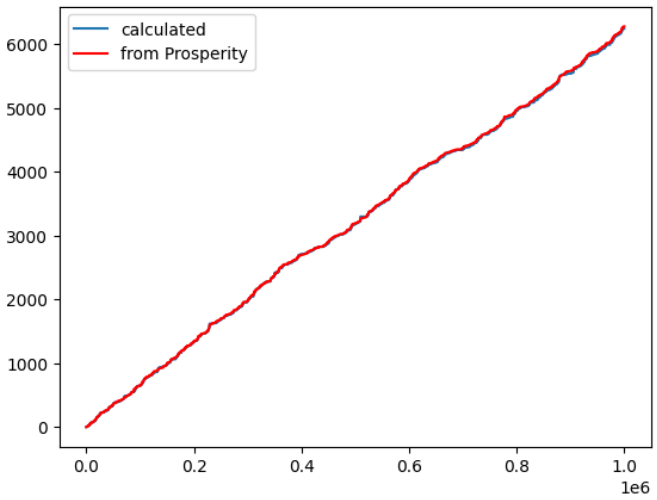

Figure 1: The pnl calculated using the `get_pnl` function, compared against the pnl returned by the Prosperity server. For this calculation, data from day 5 was used (`prices_day_5.csv` and `trades_day_5.log`).

The biggest limitation in our implementation is that *our market is static*: the implementation doesn't take into account other market participants and bots. Our key assumption is that the outstanding market quote Q_i at every timestamp is represented by the log files from the Prosperity server, regardless of the action of the algorithm and other market participants. What's the effect of this on the PnL? Does it over- or underestimate it?

For the potential extra orders from the bots to match outstanding algorithm order, we propose 3 mechanisms [3]:

1. The bots never make extra orders.
   In this case, it would not matter if the algorithm placed orders exceeding the quantities of the market quotes, since they will not be matched and will be cancelled at the end of every iteration. This will underestimate the PnL.

2. The bots always make extra orders fully matching the outstanding algo orders.
   In this case, the algo orders will always be excecuted in full, resulting in an overestimation of the PnL.

3. The bots make extra orders matching the outstanding algo orders, up to some threshold price.
   Concretely, we set the condition that the algo buy orders $>$ mid price and

[3] See the Appendix for a concrete example of the impact of the mechanisms on the pnl.

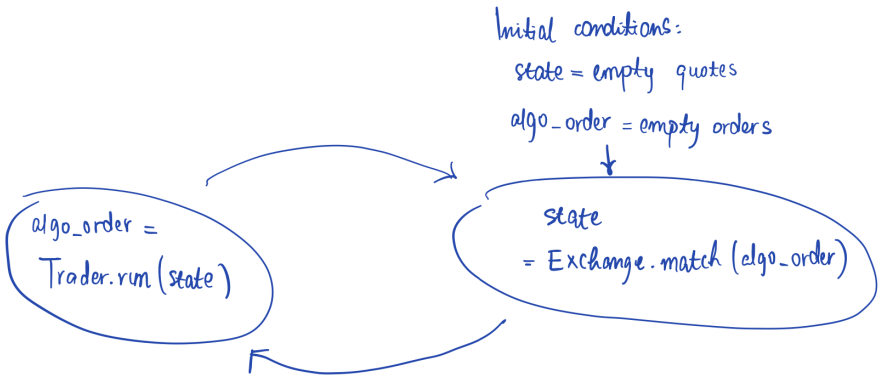

Figure 2: At each timestep, a `TradingState` is fed into `Trader`, which places some orders based on the state. The `Exchange` then matches the orders with the market quotes in the state that was fed into `Trader`. After the matching, a new `TradingState` is generated.

the algo sell orders $<$ mid price for the bots to make extra orders. The extra order quantity is q times the full quantity of the outstanding order, where $0 < q \leq 1$.

Another limitation is the lack of other market participants/counterparties. The `TradingState` object should include trades between other market participants, but this is obviously missing in the current implementation. Currently, this is left blank and so the algorithm can only take into account its own trades when making trading decisions. One possible solution is to provide this information in the static format as well, using the log files from Prosperity.

This is summarised in the `Exchange.match()` function, which takes as inputs current timestamp, current market listing, current algo orders, current positions and extra bot orders. It generates a `TradingState` object at the given timestamp as follows:

1. Iterate over each product available on the market.
2. Check algo orders if the algo has placed an order for this product. If yes, process the orders of the algo (sort in increasing/decreasing order, merge duplicates etc).
3. Observe the position limit: if the algo orders are placed such that the limit is exceeded if all orders are executed, then cancel all algo orders immediately.
4. Match algo orders and market orders [4]. The market orders are taken from

[4] A note on priority: we take the algo orders as the incoming order, so the orders are matched

the previous state, such that the state that `Trader` saw to place the orders is the same state is used in the matching.

5. Update the positions.
6. Put outstanding market orders from this time step and the market orders for the next time step in the `TradingState` for the next iteration by `Trader`.
7. Depending on the choice of the parameter `extra_bot_orders`, do either of the following: ignore outstanding algo orders, execute all outstanding algo orders, or match all outstanding algo bid (ask) orders above (below) the current midprice with probability p.

This is summarised in Fig. 3.

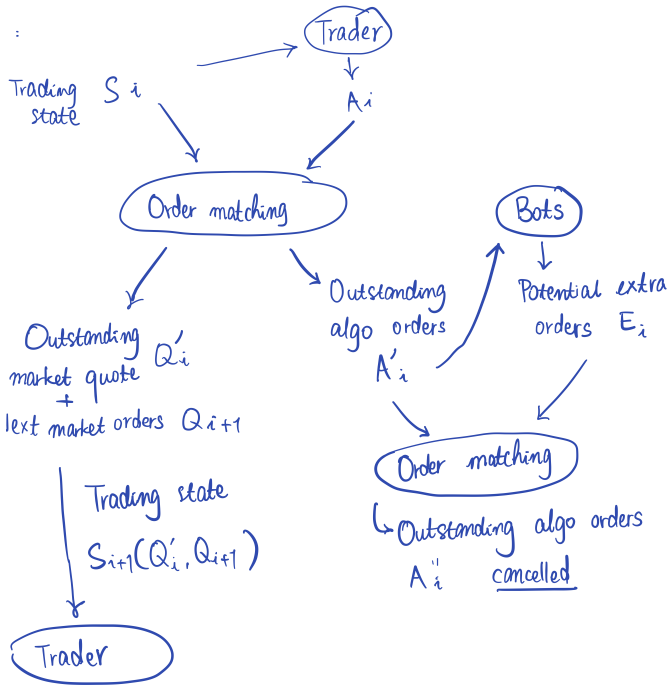

Figure 3: The outline of the `Exchange.match()` function. According to the Prosperity Wiki, the orders by the algorithm are matched up to twice: once with the market quotes, and once more if the bots decide to place extra orders in response to the algorithm orders. At the end of the iteration, all outstanding orders from the algorithm are cancelled. At the next time step, all outstanding market orders, together with the new market orders, are fed into the algorithm in the form of a `TradingState` object (see Footnote 3 for the caveat).

in order of attractiveness for the algorithm. This is in accordance with the example provided in the Wiki.

2.3 `trader.py`

Includes the `Trader` class. This includes the `run` function which executes the trading algorithm based on the `TradingState` received.

2.3.1 Round 1

In Round 1, there are 2 products available: AMETHYSTS and STARFRUIT.

Since AMETHYSTS has a low volatility, all we have to do is to make market around the mean price, 10000.

For STARFRUIT, we use Bollinger bands as the first step. We calculate the bands as the 20-timestep moving average $\pm$ 2.5 standard deviation. We simplify the algorithm such that it only buys and sells sequentially: it buys when the midprice is within the threshold of the lower band, only if it is not already long. It then tries to exit the position at a later time by selling when the midprice hits the upper band n times, where n is a free parameter. Since the algorithm does not always manage to exit the position with one order, we keep track of the open position and set a maximum value for the possible loss. It then tries repeatedly to exit the long position when the sell price is within the loss tolerance. Here the trade-off is between trying to sell at a higher price (lower loss tolerance) and being able to exit a position quickly (higher loss tolerance) to be able to buy more frequently.

The PnL generated for the two products is shown in Fig. 4.

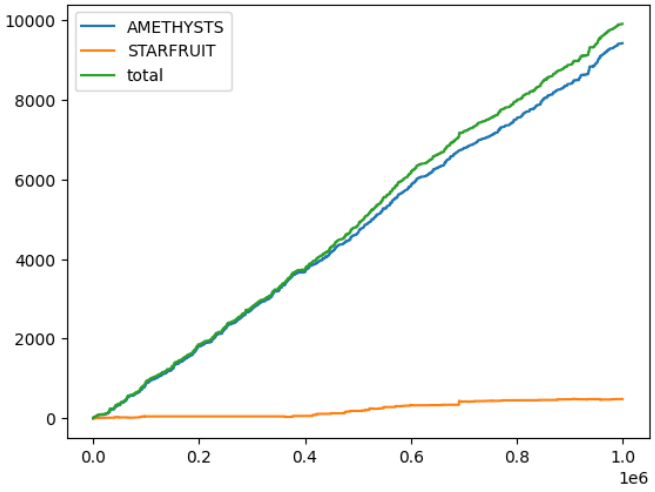

Figure 4: PnL for round 1 products. Here the extra orders are executed probabilistically.

The possible next step is to use an ARIMA model.

2.4 The execution of algorithmic trading

Initial condition: generate `TradingState` containing the market quotes at $t = 0$, by using an empty algo order.

Then at each timestep:

1. Generate algo orders by using `Trader.run(TradingState)`
2. Match the algo orders and generate the next `TradingState` including the next market listing
3. Calculate the PnL using `TradingState.own_trades`

Repeat this for 10000 timesteps using the price data of the Nth day

3 Appendix: examples

Here we illustrate the mechanism of price matching implemented in the class `Exchange`, and the calculation of PnL as a sum of realised and unrealised PnL.

3.1 Example with 2 iterations

We focus on the product AMETHYSTS. The data in this example is taken from the day 4 data, `prices_day_4.csv`. At timestamps $t_1 = 900$ and $t_2 = 1000$, the order book is as follows, with the incoming order from the algorithm shown in blue [5].

	t_1	
buy	price level	sell
	10005	29
	10004	1
	10003	
	10002	
3	10001	
	10000	
	9999	
	9998	
	9997	
1	9996	
29	9995	3

	t_2	
buy	price level	sell
	10005	20
	10004	2
2	10003	
	10002	7
	10001	
	10000	
	9999	
	9998	
	9997	
2	9996	
20	9995	5

At the end of the first iteration, the following trades have been executed for the product AMETHYSTS: sell 1 at 9996, sell 2 at 9995. Since the parameter `extra_bot_orders` is set to always execute 0.7 of the remaining algorithm orders (given that they're better than the midprice, which is the case here), 2 of the 3 buy orders are also executed at price 10001. The resulting position is -1 for this product. The order depth (outstanding market quotes) is thus sell 29 at 10005, sell 1 at 10004, buy 27 at 9995. This should be then fed into the class `Trader` to output the algo orders for the next timestamp. In the example notebook, we omit this step and code the algo orders by hand for illustrative purposes.

At the next iteration, the best bid prices are 9995 and 9996, while the best ask prices are 10002, 10004 and 10005 according to the csv files from Prosperity. Since none of the order depths from the previous iteration are worse [6] than these prices,

[5] This is not a profitable trade at all but it's for illustration purposes!

[6] From the algorithm's perspective, worse means bid<minimum bid, ask>maximum ask.

we omit them. From the order book, we see that the executed trades at the end of this iteration are: buy 2 at 10002, sell 2 at 9996 and sell 3 at 9995. Since there is no outstanding order from the algorithm, there is no need for extra orders from the bots. The resulting position is then -4. The order depth is now: sell 20 at 10005, sell 2 at 100004, sell 5 at 10002 and buy 17 at 9995.

PnL calculation. At the start of $t = 1000$, we calculate the PnL from the trades executed at $t = 900$, which are: sell 1 at 9996, sell 2 at 9995 and buy 2 at 10001. The realised pnl is $2(9995 - 10001) = -12$. For the unrealised pnl, we take the midprice at $t = 1000$ which is 9999. There is an open position, which is sell 1 at 9996, with an unrealised pnl $1(9996 - 9999) = -3$. In total, the pnl is -15.

At $t = 1100$, we calculate the pnl from the trades buy 2 at 10002, sell 2 at 9996 and sell 3 at 9995 using the midprice 9998. We first close the oldest open position, which gives a realised pnl of $9996 - 10002 = -6$. Then we close another buy order with either of the sell orders, e.g. $9996 - 10002 = -6$. Since the realised pnl is cumulative, this is $-12 - 6 - 6 = -24$. The unrealised pnl is now $-2 - 3 \times 3 = -11$. This gives a pnl of -35.

3.2 Impact of extra bot orders

In the `Exchange.match` function, there is a parameter controlling how the bots behave in response to outstanding algo orders.

We find that the PnL from the Prosperity server is closer to a probabilistic model with $p = 0.5$ and $q = 1$ (conditionally executing the remaining algo orders with probability 0.5; provided that the algo order is more attractive than the market price). This is set with the parameter "probabilistic". As expected, if we let the bots execute all outstanding orders unconditionally (parameter "always"), this results in the over-estimation of the PnL. In contrast, if we never execute any outstanding order (paremeter "never"), then we underestimate the PnL. This is shown in the figure below. Note that the figure is generated using the old version of the trading algorithm (in class `Trader` that was submitted to Prosperity), so as to reproduce the results obtained from the Prosperity server.

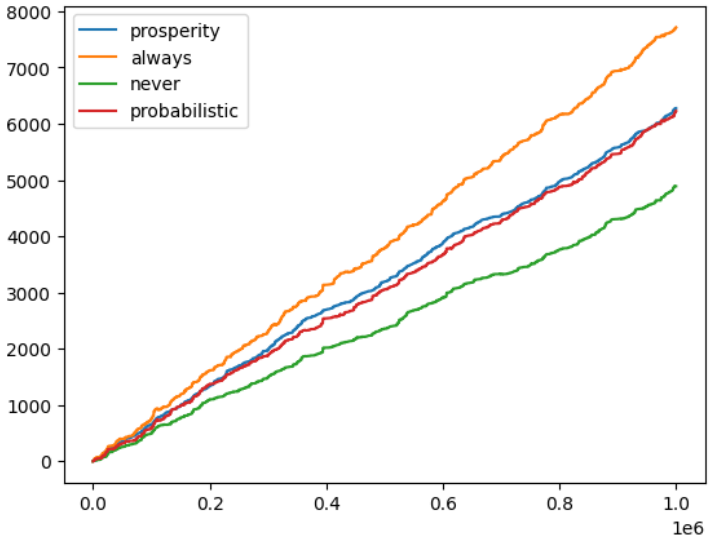

Figure 5: The PnL computed by always executing all outstanding algo orders (orange), never executing outstanding algo orders (green), or conditionally executing it with probability 0.5 (red). This is compared to the matching by the Prosperity server (blue) where the outstanding orders are conditionally (and possibly only partially) executed. The probabilistic model with $p = 0.5$ and $q = 1.0$ reproduces the result from the Prosperity server closely. The results are generated using the old version of the trading algorithm, so as to reproduce the results from the Prosperity server.