zrml-neo-swaps Documentation

0.4.1 (September 14, 2023)

1 Introduction

This document provides the mathematical and technical details for zrml-neo-swaps. The automatic market maker (AMM) implemented by zrml-neo-swaps is a variant of the Logarithmic Market Scoring Rule (LMSR; [2]) which was first developed by Gnosis (see `https://docs.gnosis.io/conditionaltokens/docs/introduction3/`). We often refer to it as AMM 2.0.

Unlike the typical implementation using a cost function (see [1]), this implementation of LMSR is a *constant-function market maker* (CFMM), similar to the classical constant product market maker, which allows us to implement *dynamic liquidity*. In other words, liquidity providers (LPs) can come and go as they please, allowing the market to self-regulate how much price resistance the AMM should provide.

As of v0.4.1, the AMM is only available for markets with two outcomes. This will be mitigated in a future publication.

2 The Trading Function

We consider a prediction market with n outcomes, denoted by $1, \ldots, n$ for simplicity. Every complete set of outcome tokens is backed a unit of collateral, denotes by \$. The AMM operates on a *liquidity pool* (or just *pool*), which consists of a *reserve* $(r_1, \ldots, r_n)$ of outcome tokens and a *liquidity parameter* b. The trading function is defined as

$$\varphi(b, r) = \sum_i e^{-r_i/b}.$$

In fact, $\varphi(b, r)$ must always equal 1. This means that a trader may change the reserve from r to r' and receive the delta provided that $\varphi(b, r') = 1$. We denote such a trade by $r \to r'$. We call these outcome-to-outcome (O2O) swaps.

However, we do not allow users to execute these types of trades. Instead, we only allow *buys* (exchange collateral for outcome tokens) and *sells* (exchange outcome tokens for collateral).

3 Buying and Selling

3.1 Buying

Buying and selling is implemented by combining complete set operations (exchange x dollars for x units of every outcome) and O2O swaps.

Alice wants to swap x dollars for units of outcome i. This is done by exchanging x dollars for x complete sets and then selling all outcomes $k \neq i$ for more i using an O2O swap $r \to r'$, which yields $y(x)$ additional units of i. *Ignoring swap fees*, this modifies the reserve to r', where $r'_k = r_k + x$ for $k \neq i$ and $r'_i = r_i - y(x)$. As trades don't change the invariant, we have $1 = \sum_k e^{-r'_k/b}$. Thus, using $1 = \varphi(b, r) = \sum_k e^{-r_k/b}$,

$$\begin{aligned}
1 &= \sum_k e^{-r'_k/b} \\
&= \sum_{k \neq i} e^{-(r_k+x)/b} + e^{-(r_i - y(x))/b} \\
&= e^{-x/b} \sum_{k \neq i} e^{-r_k/b} + e^{y(x)/b} e^{-r_i/b} \\
&= e^{-x/b}(1 - e^{-r_i/b}) + e^{y(x)/b} e^{-r_i/b}.
\end{aligned}$$

Rearranging these terms gives

$$e^{y(x)/b} = e^{r_i/b}(1 - e^{-x/b}(1 - e^{-r_i/b})),$$

and, thus,

$$\begin{aligned}
y(x) &= b \ln(e^{r_i/b}(1 - e^{-x/b}(1 - e^{-r_i/b}))) \\
&= b \ln(1 - e^{-x/b}(1 - e^{-r_i/b})) + r_i \\
&= b \ln(e^{x/b} - 1 + e^{-r_i/b}) + r_i - x.
\end{aligned}$$

Note that the total amount of outcome i that Alice receives is $y(x)$ from the O2O trade and x from the complete set operation. We denote this by $z(x) = y(x) + x$.

This allows us to calculate the *spot price* of outcome i

$$p_i(b, r) = \lim_{x \to 0} \frac{x}{z(x)} = \frac{1}{z'(0)} = \frac{1}{y'(0) + 1}.$$

Calculating the derivative of y yields

$$y'(x) = \frac{e^{x/b}}{e^{x/b} - 1 + e^{-r_i/b}} - 1$$

and thus $y'(0) = e^{r_i/b} - 1$, which yields $p_i(b, r) = e^{-r_i/b}$.

Note that this means

$$1 = \varphi(b, r) = \sum_i p_i(b, r).$$

In particular, $(p_1, \ldots, p_n)$ always maps to a probability distribution.

Trading fees are specified as fractional (a fee of $f = .01$ means that 1% are charged) and deducted from the amount of collateral before the complete set operations are executed. In other words, the liquidity providers receive fx dollars (fees are distributed pro rata amongst the liquidity providers) and Alice goes through the entire process described above with $\tilde{x} = (1-f)x$ in place of x. The spot price taking the fees into account is (as expected)

$$p_i(b, r, f) = (1-f)^{-1} e^{-r_i/b}.$$

3.2 Selling

Alice wants to swap x units of i for dollars. This is done by selling $x' < x$ units of i for $v(x) = x - x'$ units of all outcomes $k \neq i$ and then selling $v(x)$ units of complete sets, which yields $v(x)$ dollars. *Ignoring swap fees*, this modifies the reserve from r to r', where $r'_k = r_k - v(x)$ and $r_i = r_i + x'$. Using $1 = \varphi(b, r')$ and $x' = x - v(x)$, we get

$$\begin{aligned}
1 &= \sum_k e^{-r'_k/b} \\
&= \sum_{k \neq i} e^{-(r_k - v(x))/b} + e^{-(r_i + x')/b} \\
&= e^{v(x)/b} \sum_{k \neq i} e^{-r_i/b} + e^{-x'/b} e^{-r_i/b} \\
&= e^{v(x)/b}(1 - e^{-r_i/b}) + e^{-x/b} e^{v(x)/b} e^{-r_i/b} \\
&= e^{v(x)/b}(1 - e^{-r_i/b} + e^{-(r_i+x)/b}).
\end{aligned}$$

Thus, we get

$$e^{-v(x)/b} = 1 - e^{-r_i/b} + e^{-(r_i+x)/b},$$

which in turn yields

$$\begin{aligned}
v(x) &= -b \ln(1 - e^{-r_i/b} + e^{-(x+r_i)/b} \\
&= -b \ln(e^{r_i/b} - 1 + e^{-x/b}) + r_i \\
&= -b \ln(e^{(x+r_i)/b} - e^{x/b} + 1) + r_i + x.
\end{aligned}$$

Trading fees are deducted from the amount of collateral received from the complete set operation. In other words, the liquidity providers receive $fv(x)$ dollars and Alice receives $(1-f)v(x)$. The selling price (the amount of collateral received for each unit of x), is then (as expected)

$$\lim_{x \to 0} \frac{(1-f)v(x)}{x} = (1-f)v'(0) = (1-f)e^{-r_i/b} = (1-f)p_i(b, r).$$

This leads to a typical bid-ask spread between buy and sell price.

4 Dynamic Liquidity

Liquidity may be added or removed dynamically to regulate the market's price resistance. Each LP's share of the pool is tracked using pool share tokens, which record their *pro rata* share of the pool.

We consider a pool with liquidity parameter b, reserve r and a total issuance of q pool shares.

4.1 Adding Liquidity

Alice wants to add liquidity to the pool. She's willing to pay x dollars. To implement this, she first spends x dollars to buy x complete sets.

Now let i be so that $r_i = \max_k r_k$. Let $\lambda = x/r_i$ and $\mu = 1 + \lambda$. For each k, Alice moves λr_k units of k into the pool and receives λq pool shares. The liquidity parameter changes from b to $b' = \mu b$. Alice's transfers change the reserve from r to $r' = \mu r$.

The new total issuance of pool shares is μq and Alice's share of the pool now is λ/μ. Note that Alice retains the balance $(x)^n - \lambda r$ of "left-over tokens".

4.2 Withdrawing Liquidity

Alice wants to withdraw liquidity from the pool. She's willing to burn p pool shares.

Let $\lambda = p/q$ and $\mu = 1 - \lambda$. For each k, Alice receives λr_k units of k from the pool. The liquidity parameter changes from b to $b' = \mu b$. Alice's transfers change the reserve from r to $r' = \mu r$.

Alice could now convert $x = \min_i r_i$ complete sets of her newly received funds into x dollars. The remainder of the funds will remain in her wallet until the market resolves or she opts to sell them.

4.3 Fee Distribution

Fees are distributed pro rata amongst the liquidity providers. These funds are completely separate from the reserve used for trading. Transferring the fees into the pool (like the constant product market maker does) wouldn't make any sense here as collateral is not directly traded on the pool.

5 Creating Pools

Creating a pool is straightforward. The initial odds are defined by adding different amounts of each outcome to the pool. If Alice wants to deposit liquidity worth x units of collateral with initial probability p, then she starts off by buying x complete sets. The following algorithm is used to calculate how many units of each outcome go into the pool. Alice retains the other tokens as "left-overs".

Let $b = 1$, and let $r_i = -b \ln p_i$ for all i. Now let $y = x/\max_i r_i$. Then $yr_i \leq x$ for all i and there exists i_0 so that $yr_{i_0} = x$. Set $\tilde{r}_i = yr_i$ and $\tilde{b} = yb$. Then

$$p_i(\tilde{r}) = e^{-\tilde{r}_i/\tilde{b}} = e^{-r_i/b} = p_i$$

and $\max_i \tilde{r}_i = x$ (so Alice uses up at least one of her outcome balances).

In pseudocode:

```
Procedure CalculateBalances(p[1...n], x)
    b ← 1  // Initialize b, larger values may be picked for
       numerical stability
    For i from 1 to n do
```

```
        r[i] ← -b * log(p[i])
    End For
    y ← x / max(r[1...n])
    For i from 1 to n do
        r[i] ← y * r[i]
    End For
    b ← y * b
    Return r, b
End Procedure
```

Listing 1: Procedure to Calculate Balances

6 Additional Formulas

6.1 Estimated Price After Execution

After executing a buy for x units of collateral for outcome i, the new reserve of i is

$$r_i' = r_i - y((1-f)x) = -b\ln(1 - e^{-(1-f)x/b}(1 - e^{-r_i/b})).$$

Thus, the new price is

$$p_i(f, b, r') = \frac{1}{1-f}(1 - e^{-(1-f)x/b}(1 - e^{-r_i/b})).$$

After executing a sell of x units of outcome i for collateral, the new reserve of i is

$$r_i' = r_i + x' = r_i + x - v(x) = b\ln(e^{(x+r_i)/b} - e^{x/b} + 1).$$

The new price is therefore

$$p_i(f, b, r') = \frac{1}{1-f}(e^{(x+r_i)/b} - e^{x/b} + 1).$$

7 Numerical Issues

Special care must be taken to avoid over- and underflows when calculating expressions like

$$y(x) = b\ln(e^{x/b} - 1 + e^{-r_i/b}) + r_i - x,$$
$$v(x) = -b\ln(e^{r_i/b} - 1 + e^{-x/b}) + r_i.$$

The magnitude of $y(x)$ is the same as x, but the exponentials $e^{x/b}$ and $e^{-r_i/b}$ over- or underflow easily.

Let $A = 20$. Python calculates $e^A = 485165195.4097903$ and $e^{-A} = 2.061153622438558 \cdot 10^{-9}$. The fixed crate (see `https://crates.io/crates/fixed`) can represent these using `FixedU128<U80>` without considerable loss of precision or risk of over- or underflow. Let $M = e^A$.

Note that for any number a, the following are equivalent: 1) $M^{-1} \leq e^a \leq M$, 2) $M^{-1} \leq e^{-a} \leq M$. Thus, the following restrictions prevent over- and underflows when calculating the exponential expressions:

- The amount x must satisfy $x \leq Ab$.
- The price of i must satisfy $p_i(b, r) = e^{-r_i/b} \geq e^{-A}$.

How "bad" are these restrictions? The first restriction is completely irrelevant: Suppose Alice executes a trade of $y(x)$ units of outcome i for $x = Ab$ dollars, the maximum allowed value. Let $q = 1 - e^{-r_i/b} \in (0, 1)$. Then

$$\begin{aligned}
\ln(e^A) - \ln(e^A - q) &= \ln\left(\frac{e^A}{e^A - q}\right) \\
&\leq \ln\left(\frac{e^A}{e^A - 1}\right) \\
&\approx 2.0611536900435727 \cdot 10^{-9} \\
&\leq 10^{-8}.
\end{aligned}$$

Let $\varepsilon = 10^{-8}$. Then we have

$$\begin{aligned}
y(x) &= b \ln(e^A - 1 + e^{-r_i/b}) + r_i - x \\
&\geq b(\ln(e^A) - \varepsilon) + r_i - x \\
&= bA - b\varepsilon + r_i - x \\
&= r_i - b\varepsilon.
\end{aligned}$$

Thus, Alice receives all funds from the pool except $b\varepsilon$, which is very small unless the pool contains an inordinate amount of liquidity.

The second restriction means that no trades of outcome i can be executed if the price of i drops below the threshold $\varepsilon' = e^{-A}$. On markets with two outcomes (binary or scalar), this is equivalent to the price of the other outcome rising above $1 - \varepsilon'$. Due to risk considerations, these are generally scenarios that won't occur.

For markets with two outcomes (binary or scalar), we therefore make the following restriction: Any trade that moves the price of an outcome below $\varepsilon'' = .005$ (or equivalently, moves the price of an outcome above $1 - \varepsilon''$) is not allowed. This will ensure that the pool is always in a valid state where it can execute trades. Note that in the presence of a swap fee of 1%, this isn't even a restriction.

Markets with more than two outcomes are currently not allowed to use AMM 2.0 pools. The issue in a market with three or more outcomes $A, B, C, \ldots$ is that if C is a clear underdog and most of the trading happens between the favorites A and B, then the price of C might drop below the allowed threshold and *brick* the market of C (all trades involving C must be rejected due to numerical issues). While this is most likely to happen if the market is C-weakly trivialized (it is common knowledge that C will almost certainly not materialize), which should never happen on a live market, this is unfortunate. A solution for this issue is provided in the near future.

References

[1] Yiling Chen and Jennifer Wortman Vaughan, *A new understanding of prediction markets via no-regret learning*, EC '10: Proceedings of the 11th ACM conference on Electronic commerce, June 2010, Pages 189–198. https://doi.org/10.1145/1807342.1807372

[2] Robin Hanson, *Logarithmic Market Scoring Rules for Modular Combinatorial Information Aggregation*, The Journal of Prediction Markets, 1(1), May 2003. https://doi.org/10.5750/jpm.v1i1.417