University of Oxford


# Supervised Learning Approaches to Forecasting Liquidity Dynamics in Uniswap v3 Liquidity Pools

by

Radu Dragomirescu

St. Edmund Hall

A dissertation submitted in partial fulfilment of the degree of Master of Science in Statistical Science.

*Department of Statistics, 24–29 St Giles,*
*Oxford, OX1 3LB*

October 2023

This is my own work (except where otherwise indicated).

Candidate: Radu-Andrei Dragomirescu

Signed: Radu-Andrei Dragomirescu

Date: 16/10/2023

**Abstract**

Launched in May 2021, the decentralized exchange Uniswap v3 has rapidly emerged as a central venue for cryptocurrency trading, offering deeper liquidity than leading centralized exchanges across widely traded asset pairs. Its concentrated liquidity innovation enables highly active liquidity provision strategies, creating a uniquely complex and data-rich environment for liquidity dynamics. This paper produces the first forecasts of liquidity dynamics in Uniswap v3 liquidity pools through a comprehensive supervised learning framework using blockchain panel data about liquidity provision. Using Ethereum blockchain data for the period April 14, 2023 - June 28, 2023, we select as our setting pools USDC-WETH-0.0005, the top-ranked liquidity pool by total value locked and trading volume, and USDC-WETH-0.003, its inextricably linked sister pool. We develop a systematic procedure to preprocess blockchain panel data for supervised learning problems and create two datasets: one where we forecast liquidity dynamics from liquidity operations in pool USDC-WETH-0.0005, and one where we forecast liquidity dynamics from operations in USDC-WETH-0.003. For each dataset, we represent incoming liquidity dynamics through 2 classification targets recording the type (mint or burn) of the next liquidity operation in each pool and 4 regression targets recording the arrival time of the next operation of each type in each pool. We build classification and regression supervised learning pipelines to implement a comprehensive range of appropriate methods and find the estimator with the best estimated generalization performance for each target. We find classifiers that outperform baseline persistence models for 3 of 4 classification targets and regressors that materially outperform persistence models for all 8 regression targets. Finally, conducting model evaluation for the 12 best estimators, we highlight systematic shortcomings and identify dataset shift and feature space related limitations as likely causes. We propose sample size and feature space extensions and a future methodological adjustment to address limitations. We conclude that despite data constraints, our comprehensive supervised learning framework shows promising results for forecasting liquidity dynamics in Uniswap v3 pools.

**Acknowledgements**

I would like to thank my supervisor, Dr. Mihai Cucuringu, for his reassuring guidance throughout the project and his availability to give insightful feedback. I would also like to thank Emmanuel Djanga for regularly liaising with the project's data provider on my behalf about identified errors and for sharing his hyperparameter tuning wisdom. I express my thanks to Deborah Miori for the discussions which shaped the direction of this project and for sharing her expertise in implementing ideas. Mihai, Emmanuel, and Debbie have been instrumental in bringing this dissertation to fruition. More broadly, I'd like to thank my friends for their emotional support and putting some lightheartedness in even the most stressful days. Most importantly, I am grateful to my parents for their unconditional support and for the privileges they have afforded me. Whatever I achieve is made possible by your love and sacrifices.

# Contents

# List of Figures

# List of Tables

# List of Acronyms

**BTC** Bitcoin

**CEX** Centralized Exchanges

**CFMM** Constant Function Market Maker

**CL** Concentrated Liquidity

**CPMM** Constant Product Market Maker

**DeFi** Decentralized Finance

**DEX** Decentralized Exchanges

**ETH** Ethereum

**LP** Liquidity Providers

**LT** Liquidity Takers

**P2P** Peer-to-peer

**TVL** Total Value Locked

**USDC** USD Coin

**USP3** Uniswap v3

**WETH** Wrapped Ether

# Chapter 1

# Introduction

## 1.1 Decentralized Finance and Uniswap v3

Blockchains are computer programs that function as digital ledgers shared by a community of users to record transactions. Community-validated transactions are linked temporally as immutable blocks in an ever-growing chain which stores the full history of activity [1]. One of the most disruptive blockchain powered innovations is Decentralized Finance (DeFi), a financial system wherein transactions are facilitated by Peer-to-peer (P2P) networks, foregoing arbitration by financial institutions. DeFi shows promise in reducing transaction costs, improving execution times, broadening financial inclusion, enhancing financial transparency, removing geographical barriers, and promoting technological and financial innovation [2], [3]. The ecosystem has rapidly grown since its recent inception, with a Total Value Locked (TVL) – i.e., assets under management – of approximately $40 billion USD as of September 2023[1] and over 6 million regularly active participant wallets [4]. While Bitcoin (BTC) [5] was the first application of blockchain technology and remains the most highly capitalized blockchain, with a market capitalization in excess of $500 billion USD [6], it was the development of Ethereum (ETH) [7] in 2014 that set the foundation for DeFi. Unlike Bitcoin, Ethereum supports smart contracts, which are programming objects that live on the blockchain and can make up complex financial architectures, allowing participants to lend and borrow, exchange, and invest in digital assets and their derivatives [8].

Decentralized Exchanges (DEX) use smart contracts to coordinate the P2P, non-custodial exchange of digital assets in a transparent and systematic way. Compared to centralized exchanges, they provide potentially lower transaction fees, more diverse trading opportunities via access to riskier and less liquid assets, and lower counterparty risk (no trust in a centralized intermediary is required) [9]. As such, they are a main component of the DeFi ecosystem, with approximately $12 billion USD TVL and a daily trading volume hovering around $2 billion USD as of September 2023.[2] The largest and most active DEXs use Constant Function Market Maker (CFMM) protocols. Instead of matching buyers to sell-

[1] https://defillama.com/ tracks daily TVL in DeFi, which is naturally volatile. Since its proliferation in 2018, DeFi reports TVL as high as over $200 billion USD at the end of 2021.

[2] Based on data from https://defillama.com/dexs.

ers and determining market price by matched bid and asked orders like traditional order book exchanges, CFMMs operate a peer-to-pool system to satisfy trading demand and set prices algorithmically through a simple function that preserves a constant. Participants wishing to trade a cryptocurrency for another – so-called Liquidity Takers (LT) – send their outgoing tokens to a liquidity pool that holds reserves of both currencies and pay a percentage exchange fee. How much of the other token they get out of the pool (i.e., the price) is determined automatically such that the pool's programmed measure of its cumulative holdings stays constant. In turn, liquidity pools are created and funded by market participants – so-called Liquidity Providers (LP) – who deposit (**mint**) their tokens to earn a portion of the exchange fees commensurate to their share of the pool when trades occur. LPs can freely withdraw (**burn**) their deposit from a pool, but their share will reflect the composition of the pool at withdrawal time, not the token amounts they invested. Since prices are necessarily localized to liquidity pools, arbitrageurs have an extensive role in maintaining price consistency across DEXs and Centralized Exchanges (CEX) [10], [11]. CFMMs have been shown to be conducive to price behavior that closely tracks that on the most liquid CEXs [12]. Thus, CFMMs offer participants a price-stable trading alternative that often compares favorably to CEXs. Liquidity providers fulfill an essential role in this system, so their behavior is a key area of research.

The DEX Uniswap v3 (USP3) is among the richest settings available for studying liquidity provision. As a successor of Uniswap v2 [13] and Uniswap v1 [14], the first DEX to gain mass appeal, it's the largest DEX by TVL at over $2.1 billion USD as of September 2023.[3] Importantly, USP3 displays *market depth* – a common liquidity measure representing how much of one asset can be traded for another at a given price – that is significantly higher than CEXs like Coinbase and Binance across large-cap, mid-cap, and small-cap asset pairs [15]. Offering 2-3 times more liquidity than major CEXs across the most traded asset pairs, USP3 attracts LTs through a price impact advantage that grows with trade size [16]. Moreover, USP3 introduces Concentrated Liquidity (CL) [17], which allows LPs to earn as much as 4,000 times on their capital as they would've on predecessor Uniswap v2 and gives them an active management toolkit [18]. This comes alongside the choice of funding pools with different fee tiers for the same asset pair, between which trading and liquidity dynamics are linked [19]. Finally, while CEXs are dominated by institutional high frequency market makers [20], USP3's passive liquidity provision attracts more diverse participation, appealing to institutional and retail users alike [15]. It is clear that Uniswap v3 is a uniquely complex, data-rich, and relevant environment for an empirical study of liquidity provision on decentralized exchanges.

## 1.2 Main Contributions

This paper produces the first forecasts of liquidity dynamics in Uniswap v3 liquidity pools through a comprehensive supervised learning framework using blockchain panel data about liquidity provision. We access data from the Ethereum blockchain for the period April 14, 2023 - June 28, 2023 and select as our setting USDC-WETH-0.0005 – the USP3 pool with the largest TVL and trading volume which exchanges USD Coin (USDC) and Wrapped Ether (WETH) with a 0.05% swap fee – and USDC-WETH-0.003, its inextrica-

[3] Ibid.

bly linked sister pool which exchanges the same assets with a 0.3% swap fee. We develop a systematic procedure to preprocess blockchain panel data for supervised learning problems and apply it to create two datasets: one where we forecast liquidity dynamics from liquidity operations in pool USDC-WETH-0.0005, and one where we forecast liquidity dynamics from operations in pool USDC-WETH-0.003.

We represent incoming liquidity dynamics by choosing as targets the **type** (mint or burn) of the next liquidity operation in each pool and the **arrival time** of the next operation of each type in each pool, predicting these from each dataset. This yields 4 classification and 8 regression targets. We then build classification and regression supervised learning pipelines to tune and implement a comprehensive range of appropriate methods and find the estimator with the best estimated generalization performance for each target. Finally, we compare the 12 best estimators with respective baseline persistence models and conduct model evaluation to reveal their strengths and shortcomings. We find classifiers that outperform baseline predictions for all but one classification target and regressors that materially outperform baseline predictions for all 8 regression targets. Estimators with a wide range of inductive biases best predict various targets, from a simple ridge regressor to a gradient boosting classifier with 158 weak learners. This lends support to our approach of implementing a comprehensive range of methods. However, we notice systematic shortcomings in our estimators' performance on unseen data, which we find are likely caused by dataset shift and a limited sample size and feature space. To address the identified shortcomings and improve generalization performance, we propose sample size and feature space extensions and a future methodological adjustment. Overall, we conclude that despite data constraints, our comprehensive supervised learning approach shows promising results for forecasting liquidity dynamics in USP3 pools.

This paper is structured as follows. Chapter 2 describes the market making and liquidity provision mechanisms on USP3, contextualizing our feature space, and summarizes key literature on liquidity and trading on USP3, highlighting the literature gap this paper addresses. Chapter 3 explores the data available to us and discusses the empirical motivations and data constraints that focus our analysis on pools USDC-WETH-0.0005 and USDC-WETH-0.003. Chapter 4 describes the process through which we engineer supervised learning features and targets from raw blockchain data, following which we analyze feature and target distributions, justifying the transformations we apply to them in our machine learning pipelines. Chapter 5 provides a detailed explanation of all components of our classification and regression supervised learning pipelines, and for each individual method, discusses its merits and hyperparameter search. Chapter 6 discusses our pipeline's results for the 4 classification targets, conducting detailed model evaluation for the most successful classifier in each case and highlighting commonalities. Chapter 7 proceeds identically for the 8 regression targets. Finally, Chapter 8 synthesizes this paper's approach and results, discusses the identified limitations, and proposes extensions to address these and improve the scope and generalization performance of liquidity dynamics forecasts. The code for this project is extensive, so we provide it in this GitHub repository where files are organized by chapter.

# Chapter 2

# Technical Background & Literature

This chapter provides a technical description of the market making and liquidity provision mechanisms on Uniswap v3 to give the reader the background necessary to understand our feature space, elaborated upon in Chapter 4. This discussion concerns Uniswap v3, though other DEXs share some of the same structural components. See [21] and [22] for a broader mathematical review of CFMM protocols and their application in the most popular DEXs. The final section organizes the literature about liquidity and trading on USP3 and highlights the literature gap this paper addresses.

## 2.1 Uniswap v3

While some CFMM protocols support liquidity pools that consist of more than two assets ([23], [24]), a Uniswap pool can contain any two digital assets. We consider throughout this chapter a USP3 pool holding quantities $x > 0$ and $y > 0$ of some cryptocurrencies $X$ and $Y$ respectively. The price in a pool measures how many units of a token are worth a unit of the other token. Without loss of generality, we express price as how many units of Y one unit of X buys.

### 2.1.1 Constant Product Market Maker

USP3 uses a Constant Product Market Maker (CPMM) [25] protocol, a particular flavor of CFMM. At first leaving aside concentrated liquidity, the defining CPMM equation is the following **conservation function**:

$$x \cdot y = L^2, \tag{2.1}$$

where $L^2$ is the liquidity constant that must be preserved when trades occur. Namely, if a participant wants to swap $\Delta x$ in exchange for some quantity of $Y$, the pool will offer her $\Delta y$ such that:

$$[x + (1 - \gamma)\Delta x] \cdot (y - \Delta y) = x \cdot y = L^2, \tag{2.2}$$

where $0.0001 \leq \gamma \leq 0.01$ is the swap fee withheld by the protocol to compensate LPs. The motivation for this condition is revealed by its pricing implications. The **effective**

**price** is defined as:

$$P = \frac{\Delta y}{\Delta x}. \tag{2.3}$$

The **spot price** is:

$$p = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x}, \tag{2.4}$$

i.e., how much $Y$ the trader receives in exchange for an infinitesimal amount of $X$. By using the approximation $\gamma \approx 0$ and Equation (2.2), we notice:

$$\begin{aligned} y - \Delta y &= \frac{L^2}{x + \Delta x} = \frac{x \cdot y}{x + \Delta x} \\ \implies \frac{\Delta y}{\Delta x} &= \frac{y}{\Delta x} - \frac{x \cdot y}{\Delta x \cdot (x + \Delta x)} \\ &= \frac{y}{x + \Delta x} = P. \end{aligned}$$

Plugging this into Equation (2.4), we find that:

$$p = \lim_{\Delta x \to 0} \frac{y}{x + \Delta x} = \frac{y}{x}, \tag{2.5}$$

so the spot price is the ratio of tokens in the pool [26]. Thus, this mechanism prices in a way that reflects supply and demand. As traders swap $X$ for $Y$ in the pool, depleting $Y$ reserves, asset $Y$ becomes increasingly expensive in units of $X$. This is shown graphically for a toy example by Figure 2.1, where we use the equivalent definition of price $p = -\frac{dy}{dx}$ [27].

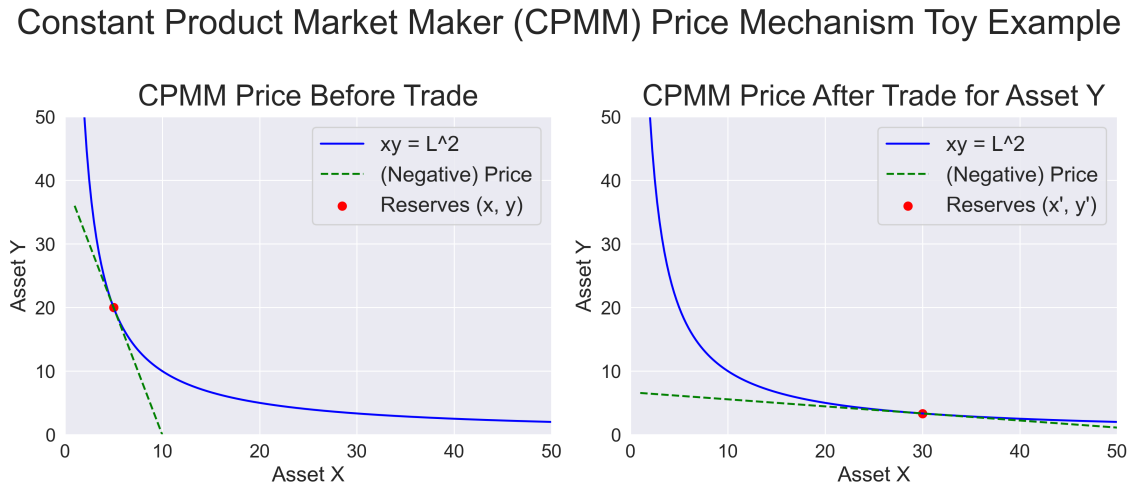



Figure 2.1: Toy example displaying the price impact of a trade for asset $Y$ as a function of the resulting reserves of X and Y for a CPMM protocol.

Importantly, Equation (2.5) shows that if a pool has a larger $L^2$ (i.e., more liquidity), the effective price is impacted less by trades. Nevertheless, **price slippage** – the change in the price of an asset during a trade – is always expected in CPMM mechanisms [22]. Using the above definitions, we express price slippage as:

$$S = p - P. \tag{2.6}$$

Price slippage is in fact often the biggest cost consideration for traders ([15], [28]), so it's particularly important to consider the liquidity dynamics of the pool they are trading in. This is a main motivation for this paper.

### 2.1.2 Concentrated Liquidity Provision

LPs can deposit (**mint**) tokens to the pool in a ratio that preserves the spot price. Namely, if an LP mints $x^*$ of asset $X$ and $y^*$ of asset $Y$ in the pool, it must be the case that $\frac{y+y^*}{x+x^*} = p$. This will increase the liquidity constant $L^2$ to $L^{*2} = (x+x^*)(y+y^*)$. $L^2$ increases if and only if new liquidity is minted. LPs who have contributed to a pool's reserves can always withdraw (**burn**) their liquidity, which decreases the liquidity constant $L^2$. The token ratio an LP receives when she burns her position in the pool is also decided such that the spot price is preserved, so it will differ from the ratio of tokens she minted unless the spot price is identical at the mint and burn times. Changes in an LP's liquidity composition always constitute a loss compared to holding the ratio of tokens she minted, as the convexity of the trading condition in Equation (2.2) causes her to be disproportionately exposed to the asset which becomes cheaper [29]. Because this loss is unrealized until burn time, it is called **impermanent loss**.

Concentrated liquidity is a key feature of USP3. When LPs mint liquidity, they must specify a price range $[p_l, p_u]$ over which they provide liquidity, rather than providing liquidity over the entire price space. LPs only earn swap fees if the trade occurs at a price included in their price range. In exchange, they receive a multiplier on the fees they earn which increases as their price range narrows [18]. They don't specify the lower ($p_l$) and upper ($p_u$) price limits directly, instead specifying a *lower tick* $i_l$ and an *upper tick* $i_u$. The relationship between a tick $i \in \mathbb{Z}$ and (spot) price is given by:

$$\sqrt{p}(i) = 1.0001^{\frac{i}{2}}, \tag{2.7}$$

meaning that each tick is 0.01% (1 basis point) away in price movement from its neighboring ticks [17]. The minimum distance required between $i_l$ and $i_u$ (**tick spacing**) for a valid mint increases with the fee $\gamma$ the pool charges. Table 2.1 gives the tick spacing for each fee tier on USP3 and the approximate percentage price change it represents. For more technical details on why the price space is discretized into ticks, see [17].

Under CL in USP3, mints and burns include positive quantities of both tokens if and only if the LP's specified tick range includes the current tick $i_c$ [30]. From Equation (2.7), $i_c$ is related to the current price $p$ by:

$$i_c = \lfloor log_{\sqrt{1.0001}} \sqrt{p} \rfloor. \tag{2.8}$$

Instead, if the mint (burn) operation has $i_l > i_c$, the LP only mints (burns) a positive quantity of asset $Y$ (which is currently cheaper than the LP's valuation of it). Conversely, if the mint (burn) operation has $i_u < i_c$, the LP only mints (burns) a positive quantity of asset $X$ (which is currently cheaper than the LP's valuation of it).

A notable implication of CL is that an LP's allocation over a price range is effectively a CPMM with higher liquidity, as elaborated upon in [17]. Alongside the fact that CL encourages LPs to provide liquidity around the current price to earn swap fees, this means that price slippage is lower on USP3 compared to a simple CPMM.

### 2.1.3 Fee Tiers

USP3 pools have four possible fee tiers for each token pair:

$$\gamma = \begin{cases} 0.01\% & \text{(1 basis point)} \\ 0.05\% & \text{(5 basis points} \\ 0.3\% & \text{(30 basis points)} \\ 1\% & \text{(100 basis points)} \end{cases} \tag{2.9}$$

Ceteris paribus, LTs would like to trade tokens with the lowest fee and LPs would like to earn swap fees at the highest rate. This creates an interesting dynamic between the two parties. Traders have to find the balance between a low fee and a pool with high liquidity in order to minimize slippage. LPs have to find the balance between a high fee and a pool that attracts high trading volume. It turns out that for the most traded and highly liquid asset pairs, trading is dominated by two pools with adjacent fee tiers [31], which we call **sister pools**. We find in Chapter 3 that this is also the case for liquidity operations on USP3. Table 2.1 shows the general trend in the consolidation of token pairs to particular fee tiers in Uniswap v3. While lower fee pools tend to include stablecoins[1] and blue chip tokens, the most volatile cryptocurrencies are traded in high fee pools to compensate LPs for exposure to these assets. Importantly, for the same asset pair, spillover effects in liquidity and trading dynamics have been shown to exist between sister pools [19]. Thus, when we forecast liquidity dynamics in a USP3 pool, we incorporate information about its sister pool into our feature space.

Table 2.1: Fee tiers on Uniswap v3 and their tick spacing and percentage change in price per tick, alongside common types of assets traded at each fee tier.

| **Fee Tier** | **Pair Archetype** | **Example Pair** | **Tick Spacing** | **$\Delta$ Price Per Tick** |
|---|---|---|---|---|
| 0.01% | Stablecoin pairs | USDC-USDT | 1 | 0.01% |
| 0.05% | Blue chip tokens paired with stablecoins | USDC-WETH | 10 | 0.1% |
| 0.3% | Pairs of blue chip tokens | WBTC-WETH | 60 | 0.6% |
| 1% | Pairs including exotic tokens | SHIB-WETH | 200 | 2.02% |

## 2.2 Literature on Liquidity & Trading in Uniswap v3

We introduced some of the key literature on the properties of CFMM protocols in Chapter 1. Nevertheless, these studies are outside the setting of CL, the most important change Uniswap v3 brings compared to its predecessor. Because Uniswap v3 launched in May

[1] Stablecoins are pegged to a reference asset, usually USD, in which case their value is always $\approx$ \$1 USD.

2021, the literature on liquidity and trading on the DEX is young and rapidly evolving. Studies can be broadly partitioned into three categories which we delve through in turn, though some papers navigate multiple of these. The first two categories are more theoretically oriented, while the latter is empirically focused.

### 2.2.1 Concentrated Liquidity Provision

Much of the early research on Uniswap v3 dynamics focuses on the theoretical implications of its CL innovation for optimal LP behavior. The authors in [32] derive an analytical expression for impermanent loss on USP3 and find that it increases at a faster rate in this CL setting than on Uniswap v2. Their modelling and data analysis suggest that liquidity provision on USP3 is more complex than on its predecessor, warranting active and sophisticated strategies. To that end, [27] introduces the concept of *dynamic liquidity provision strategies*, whereby LPs shift their allocation price interval when the current price moves outside of their original allocation interval, such that they keep earning swap fees. The authors show that stochastically optimized dynamic strategies which incorporate LP beliefs about price changes significantly outperform the uniform liquidity provision strategy of predecessor Uniswap v2. Finally, they find that more risk-averse LPs optimally provide liquidity over larger price ranges. In a similar vein, the authors in [29] derive a closed-form strategy for maximizing the log utility of terminal wealth as a LP operating in a CPMM with CL (like USP3). The strategy dynamically adapts liquidity provision price ranges to maximize earned fees *and* minimize **predictable loss**, an augmentation of impermanent loss introduced and formalized by the authors. Predictable loss measures the loss of value from depositing tokens in a liquidity pool compared to holding a self-financing portfolio outside of it (rather than just holding the assets outside the pool like in impermanent loss). An interesting finding is that when volatility increases, LPs are incentivized to widen their allocation's price range to reduce exposure to predictable loss.

In contrast, a non-stochastic approach is detailed in [26], where the authors develop and empirically validate an online learning model for USP3 liquidity provision which adaptively learns the optimal price range allocation. Given enough trading volume, this approach admits a positive payoff. Further, a static model for optimal liquidity provision is provided by [33], though for a finite time horizon over which LPs do have stochastic beliefs about price evolution. Similarly to [27] and [29], the authors find that the optimal liquidity range increases with risk aversion and price volatility. They also leverage their derived model to analyze the optimality of tick spacing on USP3 and show that a more detailed space of price partitions can benefit not only LPs, but also traders.

Finally, a study concerning aggregate LP behavior is [31], which shows fixed transaction costs lead to the fragmentation of LPs between sister pools, with institutional LPs actively managing their positions on the low fee pool that sees more trade volume, while retail LPs passively fund the high fee pool. Our contribution to this literature strand similarly concerns aggregate LP behavior, as we explore factors which have predictive power for liquidity dynamics through feature importance analysis for a wide range of supervised learning estimators.

### 2.2.2 Trading in Uniswap v3 Pools

Findings in the literature on liquidity taking in the broader class of CPMMs often apply directly to USP3, as CL doesn't change the actions available to LTs. As such, the literature on trading which applies to USP3 is more mature. An early relevant work is [34], where the author introduces mean-variance optimal trading strategies in continuous time for the setting of stochastic liquidity and volatility. While the asset class motivating the work is small market capitalization stocks, even the most highly capitalized cryptocurrency, Bitcoin, is known for its high volatility [35]. Given the dynamic liquidity provision strategies referred to in Section 2.2.1, it stands to reason that liquidity and volatility on USP3 pools can also be seen as stochastic phenomena. More recently, [10] derive two optimal trading models specifically for CPMMs. The first model assumes constant liquidity and that price discovery is led by a more liquid exchange, while the second assumes stochastic liquidity and on-pool price discovery, which is consistent with an efficient, highly active exchange. The authors test the performance of two strategies under the first model on USP3 data and find superior performance compared to benchmark strategies.

On a different note, the authors in [28] formalize *sandwich attacks*: a predatory combination of front-running and back-running that LTs on Uniswap can fall victim to. Using 2018-2019 Uniswap data (before v3), they find that a single adversarial actor can earn thousands of USD per day performing sandwich attacks, and these attacks may also remain profitable with multiple competing adversaries. The mechanism through which the victim trader suffers in sandwich attacks is price slippage, so this work also sheds light on how Uniswap should set price slippage limits. As stated earlier, even discounting sandwich attacks, price slippage remains the most significant trading cost Uniswap LTs generally face [15], [28]. Though it's a phenomenon which long precedes CPMM protocols [36], not much work has been dedicated to analyzing price slippage on USP3 to our knowledge. [37] derives a closed-form expression for price slippage that applies to CPMM curves like Uniswap v2 (see Figure 2.1), but doesn't consider the dampening effect of CL on price slippage. The authors in [38] propose an alternative liquidity pool mechanism wherein the mathematical relationship between the two assets on it is dynamically updated to always match some externally-defined market price. In such a dynamic CPMM, the slippage loss experienced by traders is fully captured as a gain by LPs. However, the proposed mechanism assumes that the liquidity pool is so small that the market price is not influenced by the pool price, which is not realistic for USP3 pools.

More broadly, [39] uses unsupervised representation learning to learn vector embeddings of LT *transaction graphs* (complete weighted graphs for each LT where nodes are her trades and edges are the time between trades). These embeddings are then clustered via k-means++ and yield 7 clusters of LTs. Summary statistics over the clusters show the groups have distinctive risk profiles and trading frequencies, suggesting this unsupervised method recovers different "species" of LTs. A departing point for this paper was categorizing LTs in our data according to this framework and investigating if different LT species affect pool price volatility and liquidity dynamics in different ways. However, we were unable to access DEX trading data, as detailed in Chapter 3. Our paper relates to the trading literature on USP3 through the fact that with further refinements, our liquidity dynamics forecasts may be usable as inputs for price slippage forecasts, addressing a gap in this literature strand and an important consideration for traders.

### 2.2.3 Empirical Studies

This final category includes selected works which study important empirical aspects of liquidity, volatility, and trading dynamics on Uniswap v3. In [40], the author compares LP returns before and after the introduction of CL by tracking the performance of passive and active liquidity strategies for a selection Uniswap v2 and v3 pools in the months around the launch of USP3. He finds that for the asset pair we focus our analysis on, ETH-USDC, concentrated liquidity provision is significantly more profitable on average and riskier. With respect to price volatility, [41] adapts for cryptocurrency-USDT pairs the insights of [42], where the authors forecast intraday realized volatility (RV) for top S&P 500 stocks by leveraging volatility commonalities in the stock market, and additionally forecast one-day-ahead RV using past intraday RVs as predictors. [41] similarly forecasts one-day-ahead RV for cryptocurrency pairs using intraday RVs and finds that incorporating information about commonality in cryptocurrency RV boosts forecasting performance. Though the data used in the study comes from Binance, it's reasonable to assume these findings about price volatility commonality can extend to USP3 as well, given its importance in the cryptocurrency trading ecosystem.

Finally, [19] builds a multivariate linear regression to predict the log of incoming trading volume on two active USP3 WBTC-WETH pools with fee tiers 0.05% and 0.3%. The authors design features that capture the latest liquidity and trading activity on both pools, while allowing for spillover effects from closely related USP3 pools and Binance BTC-ETH trading activity. Their final model explains a significant proportion of the variance in incoming trading volume in the lower fee pool (where LTs prefer to trade) following a mint in the higher fee pool (where LPs prefer to mint). The authors find evidence that trading and liquidity dynamics between the two pools are linked and nonlinearities in their features boost predictive power. Our paper uses the empirical success of [19] as inspiration for feature engineering. We contribute to this literature strand by developing a structured process which engineers blockchain panel data into such features and implementing a comprehensive set of supervised learning methods in the empirical risk minimization framework to forecast liquidity dynamics.

# Chapter 3

# Exploratory Data Analysis

In the previous chapter, we established the technical background of trading and liquidity provision on Uniswap v3 and summarized important strands of literature concerning the DEX, contextualizing the contributions this paper aims to make. This chapter explores the data available to us for this project, helping the reader understand our information constraints and problem setting choices.

## 3.1 Blockchain Data Summaries

Data for the project is provided by Kaiko[1]. We are given access to historical data collected directly from DEX blockchains for the period April 14, 2023 - June 28, 2023. Notably, we don't have access to trading data, but to data relating to DEX liquidity[2]. This data falls into three categories, which we explore in turn.

### 3.1.1 Pool Reference Data

The first data category from Kaiko is pool reference data[3], which stores defining and unchangeable information about each liquidity pool Kaiko collects data on. Features include the pool's protocol, fee tier, and (for USP3 pools) tick spacing. There are 13,685 pools in Kaiko's database. This data is not used in our prediction and classification pipelines, but briefly explored to illustrate how USP3 fits into the universe of DEXs tracked by Kaiko.

Figure 3.1 displays the top 7 protocols to which liquidity pools tracked by Kaiko belong. Uniswap v2 has the most recorded pools, with 4,858, followed by USP3 and PancakeSwap [43], the leading DEX on Binance's BNB Smart Chain blockchain [44] (as opposed to the Ethereum blockchain used by Uniswap). However, as we will see in Table 3.2, Uniswap

[1] https://www.kaiko.com.

[2] Initially, it was expected we would gain access to trade data and the plan for the project revolved around this data. However, due to issues with the data vendor outside of the author's control, we had to pivot to a project revolving around the DEX liquidity data, which is much lower frequency and somewhat more restrictive in the context of machine learning.

[3] https://docs.kaiko.com/#pools.

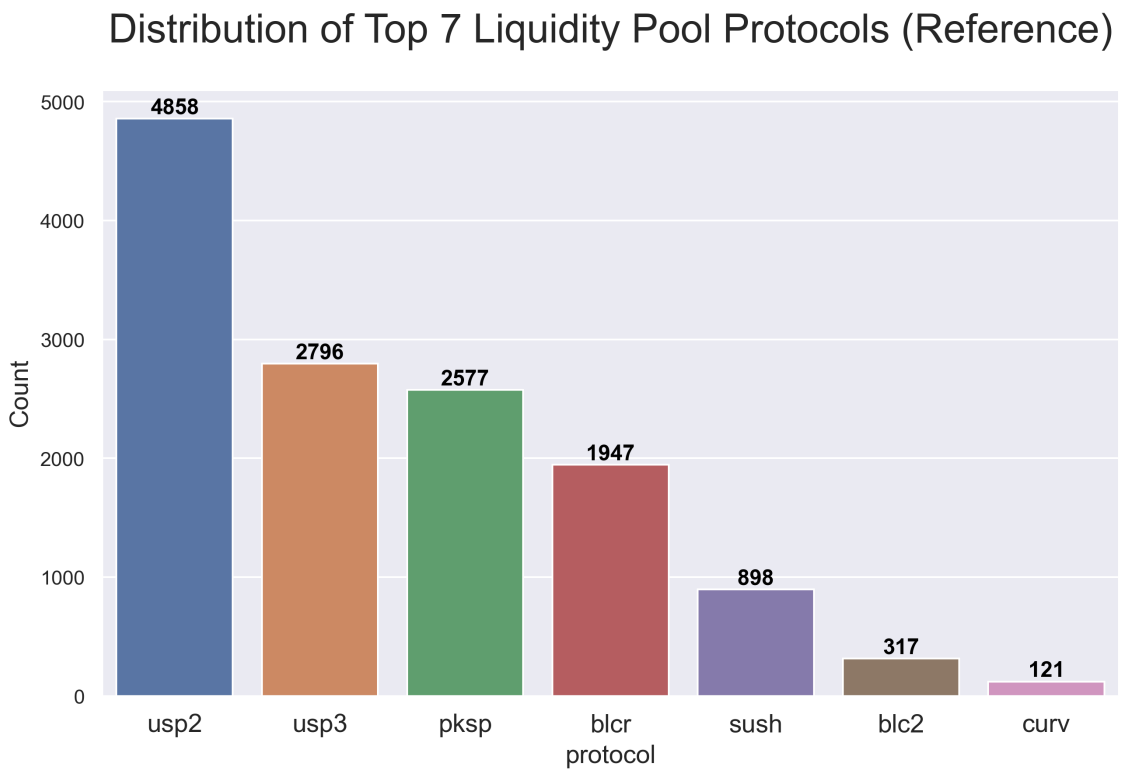



Figure 3.1: Distribution of the top 7 protocols for the liquidity pools tracked by Kaiko. 13,514 of the 13,685 pools in Kaiko's database belong to these protocols.

v2 significantly trails USP3 in the number of observed liquidity events. While Uniswap v2 may have more liquidity pools, LP activity heavily revolves around USP3.

We conduct a similar investigation for the distribution of the top 7 fee tiers in the reference data in Figure 3.2. The 0.3% fee unsurprisingly dominates, as it's the only fee tier on Uniswap v2 and a popular fee tier on USP3 and other exchanges. Fee tiers 1% and 0.05%, which are popular on USP3, complete the top 3, while the newer 0.01% fee which USP3 uses for stablecoin pairs is ranked seventh. This shows that USP3 fee tiers are popular choices across DEXs.

### 3.1.2 Liquidity Events Data

Our second data category is liquidity events data[4], where every mint and burn operation in the liquidity pools tracked by Kaiko is stored. We have data on 397,259 liquidity events in total. Table 3.1 presents an example of how mint and burn operations are stored in our raw dataset, with important features included as columns. Liquidity events are the most important data for our analysis. We construct our regression and classification targets and most of our features from this data (as elaborated on in Chapter 4).

Table 3.2 displays the distribution of all liquidity events by exchange, which are seemingly dominated by events on PancakeSwap. To explain this, we opt for more granularity

[4] https://docs.kaiko.com/#liquidity-events.

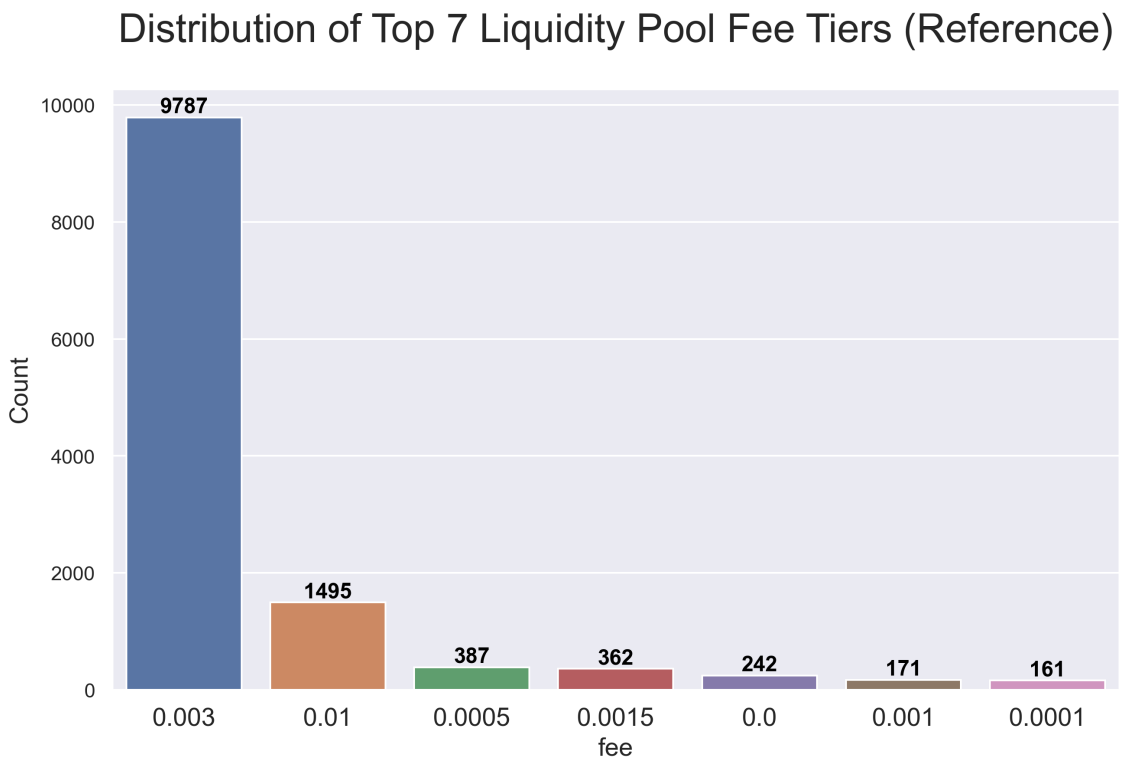



Figure 3.2: Distribution of the top 7 fee tiers for the liquidity pools tracked by Kaiko. 12,605 of the 13,685 pools in Kaiko's database belong to these protocols.

Table 3.1: Mint and burn liquidity event observation examples collected from the Ethereum blockchain. Only a relevant subset of columns is included for readability.

| **date** | **block** | **type** | **pool_name** | **price** | **amounts** | **metadata** |
|---|---|---|---|---|---|---|
| 2023-04-14 01:07:35 | 17041803 | mint | USDC-WETH-0.0005 | 0.0005 | {'symbol': 'USDC', 'amount': 10,769,631.4}, {'symbol': 'WETH', 'amount': 14,008.9} | {'lower_ticker': 200240, 'upper_ticker': 200250} |
| 2023-04-14 01:54:11 | 17042031 | burn | USDC-WETH-0.003 | 0.00048 | {'symbol': 'USDC', 'amount': 206,788.1}, {'symbol': 'WETH', 'amount': 0} | {'lower_ticker': 200100, 'upper_ticker': 200580} |

with Table 3.3, wherein we show the top 8 liquidity pools with the most events and the exchange they are a part of. We notice that nearly 300,000 of the PancakeSwap liquidity events occur on two pools: USDT-BUSD and DRIP-BUSD. We inspect each pool in turn.

Table 3.2: Number of liquidity events recorded on each DEX tracked by Kaiko between April 14, 2023 - June 28, 2023.

| **Liquidity Events** | **Exchange** |
|---|---|
| 324,894 | PancakeSwap |
| 46,574 | Uniswap v3 |
| 11,077 | Curve |
| 8,626 | Uniswap v3 |
| 3,129 | Balancer V2 |
| 2,103 | SushiSwap |
| 828 | Curve V2 |
| 28 | Balancer |

Table 3.3: Top 8 liquidity pools by the number of liquidity events between April 14, 2023 - June 28, 2023 and the DEX they belong to.

| **Liquidity Events** | **Pool Name** | **Exchange** |
|---|---|---|
| 253,513 | USDT-BUSD | PancakeSwap |
| 43,139 | DRIP-BUSD | PancakeSwap |
| 8,038 | USDC-WETH-0.0005 | Uniswap v3 |
| 5,354 | PEPE-WETH-0.003 | Uniswap v3 |
| 4,850 | USDT-WBNB | PancakeSwap |
| 4,699 | Cake-WBNB | PancakeSwap |
| 4,683 | 3pool | Curve |
| 3,523 | PEPE-WETH-0.010 | Uniswap v3 |

USDT-BUSD is a pool that exchanges two stablecoins pegged to USD: USDT [45] and BUSD [46]. As such, these tokens always have nearly the same value and investors generally swap one for another as an intermediate step to a final transaction that is of main interest. For example, if an investor holding USDT wants to purchase ETH and observes an ETH-BUSD pool has lower expected price slippage than an ETH-USDT pool, she might first swap for BUSD in the USDT-BUSD pool and then buy ETH in the ETH-BUSD pool. A PancakeSwap pool exchanging stablecoins is not an adequate environment for our prediction and classification questions, as LPs don't face a dynamic risk environment or CL choices. There is nearly no risk in being exposed to these established stablecoins and their relative price is practically constant. Moreover, once we group liquidity events by day for this pool, Figure 3.3 shows activity drops sharply before June. This is also highly undesirable, as it would lower the time frame of our predictions and marks a stark change in the process generating liquidity events, which supervised learning methods wouldn't capture.

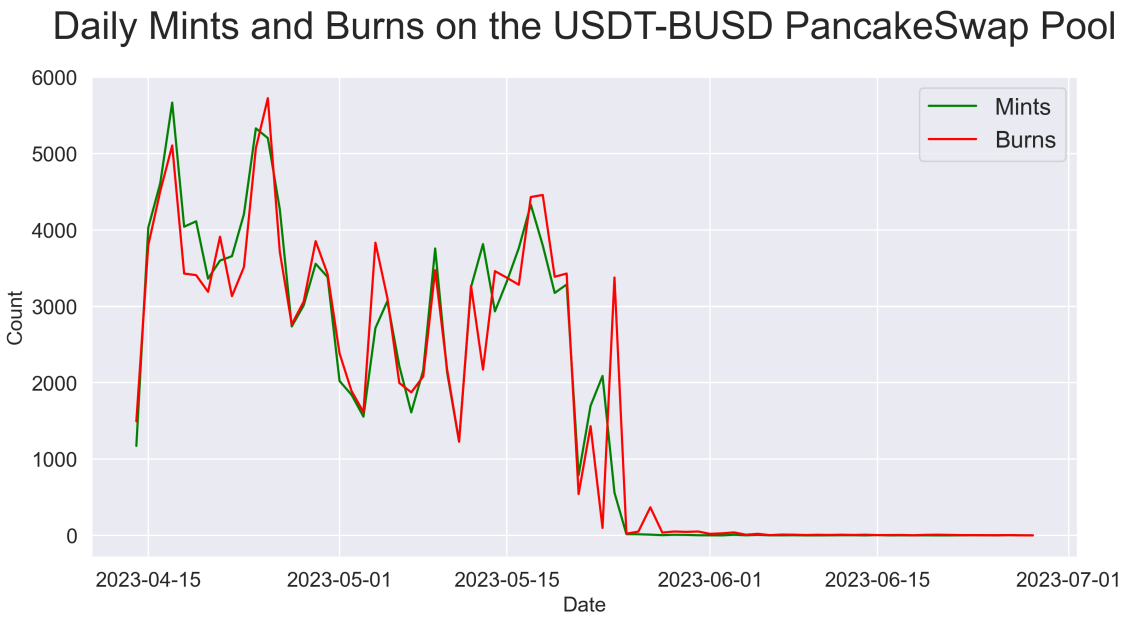



Figure 3.3: Daily number of mint and burn liquidity events recorded in the USDT-BUSD PancakeSwap pool during the period April 14, 2023 - June 28, 2023.

On the other hand, DRIP-BUSD is a pool that exchanges BUSD and DRIP [47], a cryptocurrency which has lost 99.52% of its value from September 2022 to September 2023 and has an extremely low market capitalization and trading volume as of September 2023[5]. The dominance of burning activity displayed by Figure 3.4 suggests this pool is only highly active because LPs perhaps want to end their exposure to DRIP. A burn-dominated pool with a declining token is also an inadequate environment for our analysis.

Having discounted the two most active pools in Table 3.3 as inadequate for our purposes, we are left with USP3 as the source of a plurality of the liquidity events we track. We analyze liquidity events on USP3 in Section 3.2, after describing the last category of our data.

### 3.1.3 Pool Snapshots Data

The final category of data provided by Kaiko includes liquidity snapshots, only for Uniswap v3[6]. An observation is recorded with every new block, which is every 12 seconds on the Ethereum blockchain [48]. Table 3.4 gives an example of consecutive liquidity snapshots for pool USDC-WETH-0.0005, where the first five columns are raw data, while the sixth is our engineered measure of market depth. Through snapshots, we track price and the distribution of liquidity over each tick spacing interval of a pool at block-wise granularity. We use this to build features measuring price volatility and market depth in Chapter 4.

[5] As reported by https://coinmarketcap.com/currencies/drip-network/.
[6] https://docs.kaiko.com/#uniswap-v3-liquidity-snapshots.

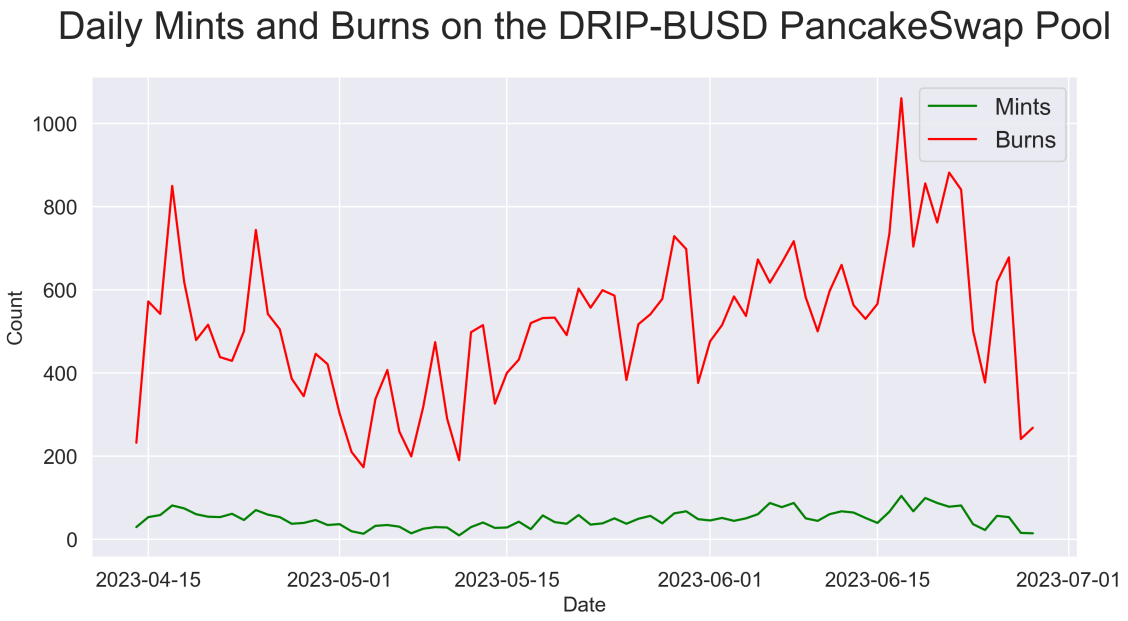



Figure 3.4: Daily number of mint and burn liquidity events recorded in the DRIP-BUSD PancakeSwap pool during the period April 14, 2023 - June 28, 2023.

Table 3.4: Consecutive liquidity snapshots collected from the Ethereum blockchain for pool USDC-WETH-0.0005. The first five columns are raw, while the sixth is engineered by the author to track market depth.

| **date** | **block** | **tick** | **price** | **snapshots** | **depth** |
|---|---|---|---|---|---|
| 2023-04-14 01:00:11 | 17041766 | 200241 | 0.0005 | {'amount0': 0, 'amount1': 99.6, 'lower_tick': 199190, 'upper_tick': 199200}, **...** {'amount0': 481,051.2, 'amount1': 0, 'lower_tick': 201190, 'upper_tick': 201200} | 480,212.9 |
| 2023-04-14 01:00:23 | 17041767 | 200241 | 0.0005 | {'amount0': 0, 'amount1': 99.6, 'lower_tick': 199190, 'upper_tick': 199200}, **...** {'amount0': 481,051.2, 'amount1': 0, 'lower_tick': 201190, 'upper_tick': 201200} | 480,212.9 |

## 3.2 Uniswap v3 Liquidity Events

Having explored our three data categories, we found that USP3 offers the most promising collection of liquidity events and also uniquely offers liquidity and price data at block-wise granularity. Thus, we proceed by exploring our 46,574 USP3 liquidity events in

more detail.

Table 3.5: Top 10 liquidity pools on Uniswap v3 by the number of liquidity events between April 14, 2023 - June 28, 2023.

| Pool Name | Liquidity Events |
|---|---|
| USDC-WETH-0.0005 | 8,038 |
| PEPE-WETH-0.003 | 5,354 |
| PEPE-WETH-0.010 | 3,523 |
| TURBO-WETH-0.010 | 2,047 |
| WETH-USDT-0.0005 | 1,755 |
| APE-WETH-0.003 | 1,563 |
| HEX-WETH-0.003 | 1,476 |
| WBTC-WETH-0.003 | 1,300 |
| USDC-USDT-0.0001 | 1,241 |
| USDC-WETH-0.003 | 1,200 |

Table 3.5 lists the top 10 USP3 pools by liquidity events. We notice that the 0.05% fee USDC-WETH pool ranks first, while its 0.3% fee sister ranks tenth. These pools exchange USDC [49], the second largest market cap stablecoin at over $26 billion USD[7], and WETH, which is equivalent to ETH [50]. In fact, the 0.05% fee pool comfortably leads trading volume and TVL on USP3, while the 0.3% fee pool is in the top 9 for both metrics[8]. These pools present a highly dynamic setting for predicting liquidity activity and provide us enough observations to capture spillover effects.

PEPE-WETH is another asset pair which has two highly active liquidity pools over the time range of our data. However, a crucial consideration is that we don't have access to the USD prices of tokens. Since we need to express the size of liquidity events and market depth in a common denomination to build features, we must rely on pools that contain one stablecoin. In such pools, we can leverage that the highly capitalized USDC is worth $1 USD and use the reported pool price to calculate the USD value of the other token.

In summary, we must narrow down our predictions to the sister USDC-WETH pools not only because of their desirable levels of activity, but equally importantly because they allow us to calculate liquidity-related features in USD. We henceforth refer to USDC-WETH-0.0005 as **Pool 5** and to USDC-WETH-0.003 as **Pool 30**. In the following subsections, we analyze them further.

### 3.2.1 USDC-WETH-0.0005 (Pool 5)

We plot the daily pattern of mints and burns for Pool 5 in Figure 3.5. While mints and burns follow a quite similar daily pattern, there is a sharp spike in both types of event compared to their averages on June 21, 2023. This is a cause for concern, as this date falls into the final 20% test data split with which we estimate our methods' generalization error and likely contributes to **dataset shift**. This would bias our estimates of the generalization

[7] According to https://coinmarketcap.com as of September 2023.
[8] According to https://info.uniswap.org/pools#/pools as of September 2023.

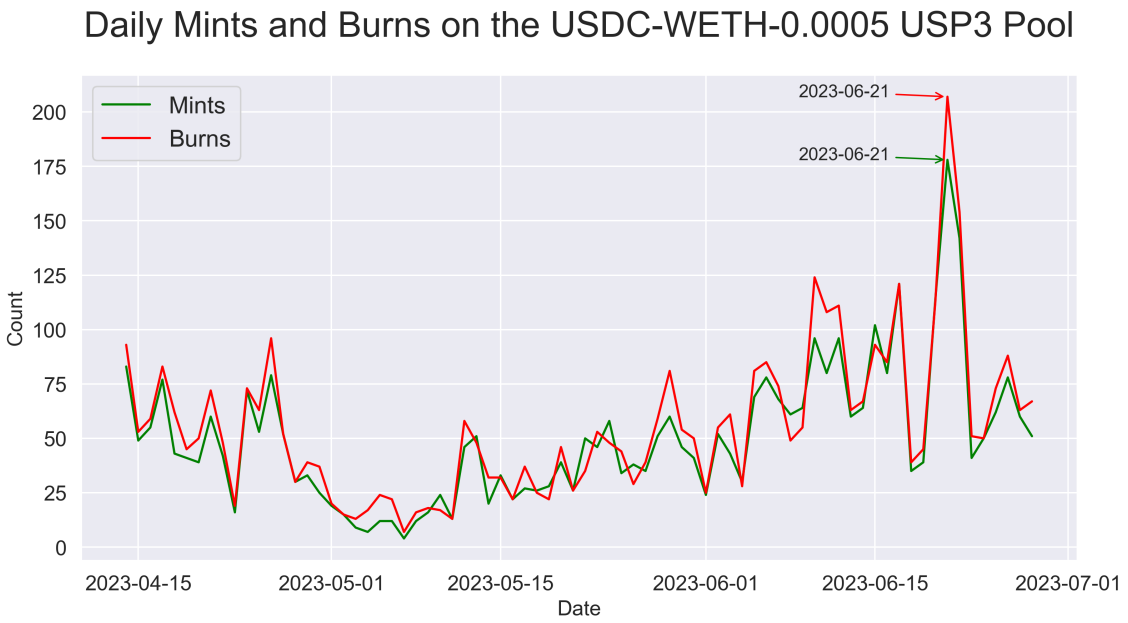



Figure 3.5: Daily number of mint and burn liquidity events recorded in the USDC-WETH-0.0005 Uniswap v3 pool during the period April 14, 2023 - June 28, 2023. An outlying peak in the number of both event types is highlighted on June 21.

performance of our supervised learning methods, which we discuss further in Chapters 6 and 7. The same date is highlighted when we plot the daily total USD sizes of mints and burns for Pool 5 in Figure 3.6. We observe a remarkable similarity in the daily mint and burn sizes, highlighting they follow a very similar and interconnected dynamic.

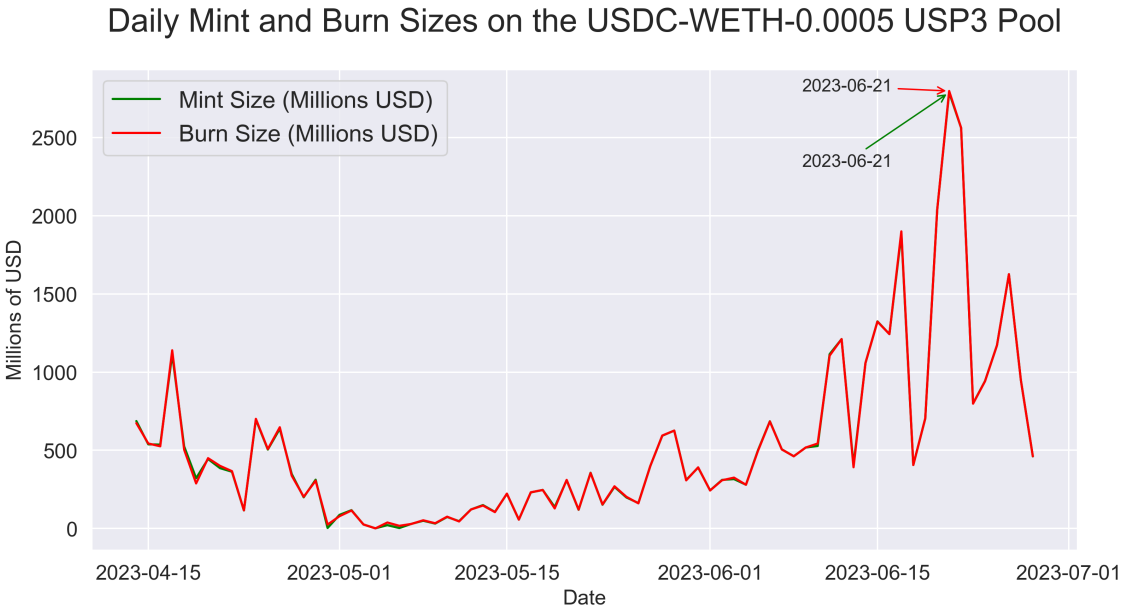



Figure 3.6: Daily USD size of mint and burn liquidity events recorded in the USDC-WETH-0.0005 Uniswap v3 pool during the period April 14, 2023 - June 28, 2023. An outlying peak in the USD size of both event types is highlighted on June 21.

### 3.2.2 USDC-WETH-0.003 (Pool 30)

Although Pool 30 offers LPs a significantly higher trade fee than Pool 5, there are nearly 8 times fewer liquidity events on this pool. Weekly trading volume on Pool 5 is almost 40 times larger than on Pool 30[9], so LPs earn swap fees much more often on Pool 5. We proceed with the same exploratory plots for Pool 30 as in Section 3.2.1.

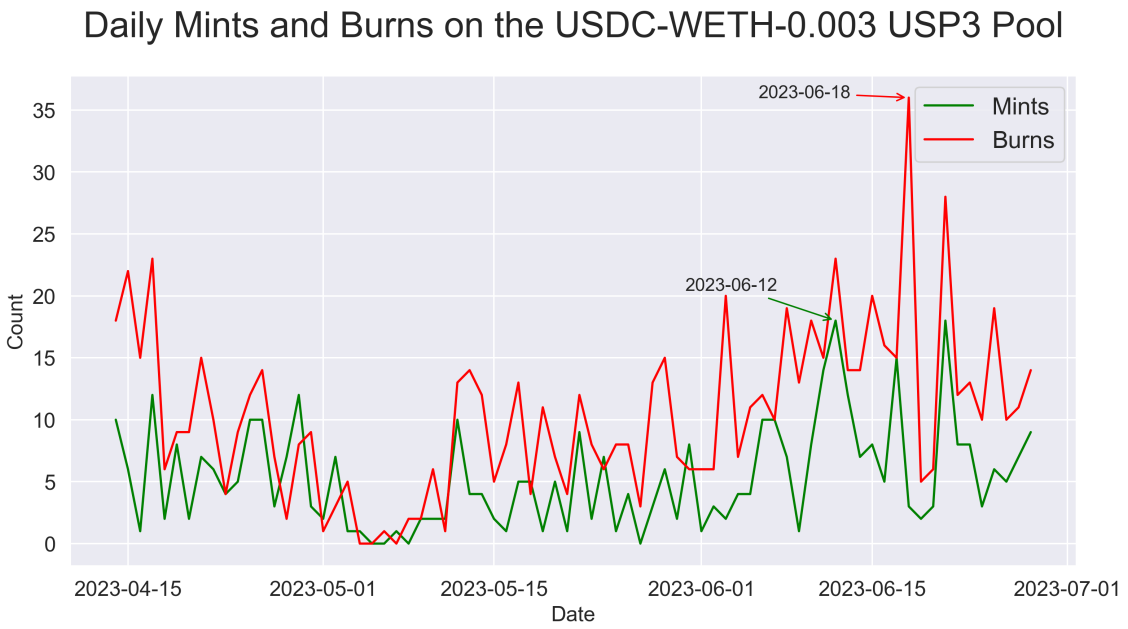



Figure 3.7: Daily number of mint and burn liquidity events recorded in the USDC-WETH-0.003 Uniswap v3 pool during the period April 14, 2023 - June 28, 2023. Peaks in the number of mint and burn events are highlighted on June 12 and June 18, respectively.

Figure 3.7 shows the pattern of daily mint and burn operations on Pool 30. The number of burn events exceeds the number of mint events nearly every day, though they still follow a very similar pattern. We have an outlying peak in burn operations on June 18 and a less extreme peak in mint operations on June 12. We also observe a bump in both types of operations on June 21, but not as sharp as for Pool 5 (Figure 3.5). Dataset shift remains a significant concern for Pool 30, as the final 20% test data split sees the most volatility in the number of daily events. Although the number of burns consistently exceeds the number of mints, Figure 3.8 shows that daily sizes largely even out and are both maximized on June 21, like in Figure 3.6. This similarly shows the shared dynamic of events on Pool 30. Together, the aforementioned figures suggest strong similarities between liquidity dynamics on Pools 5 and 30 and lead us to conclude spillover effects are highly plausible.

Finally, we plot our block-wise measure of *market depth* for both pools in Figure 3.9. We define market depth as the USD value of liquidity in the tick spacing interval including the current price, so it is no surprise market depth is usually higher on Pool 30, whose tick spacing is six times higher (Table 2.1). Notably, the Figure's jaggedness is inaccurate. We should observe liquidity snapshots every 12 seconds for each pool, but our

[9]Ibid.

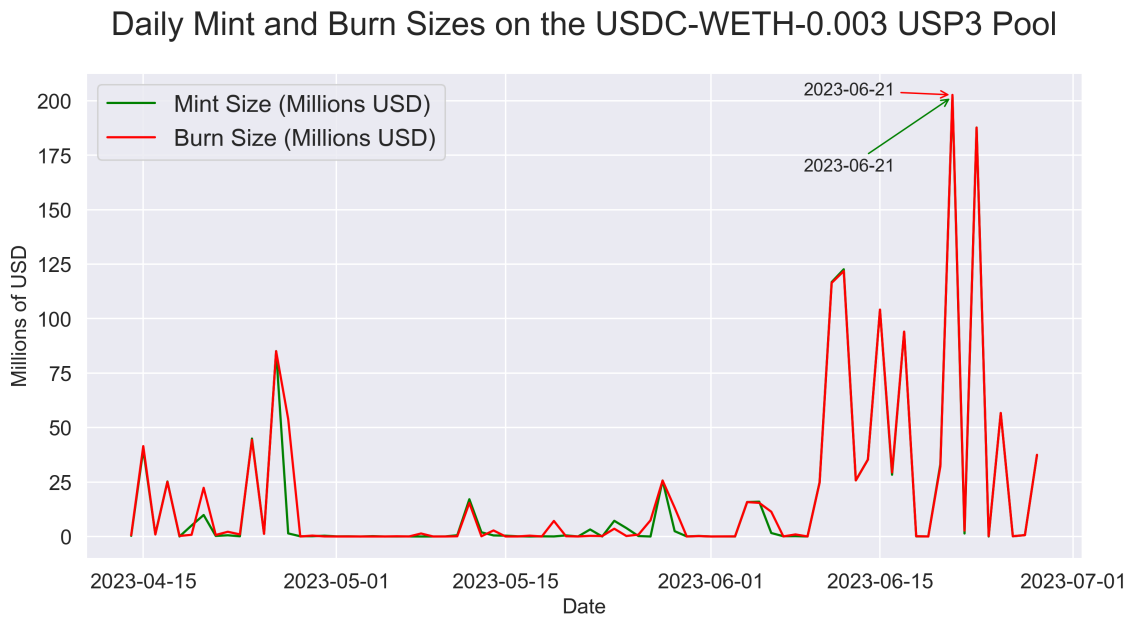



Figure 3.8: Daily USD size of mint and burn liquidity events recorded in the USDC-WETH-0.003 Uniswap v3 pool during the period April 14, 2023 - June 28, 2023. An outlying peak in the USD size of both event types is highlighted on June 21.

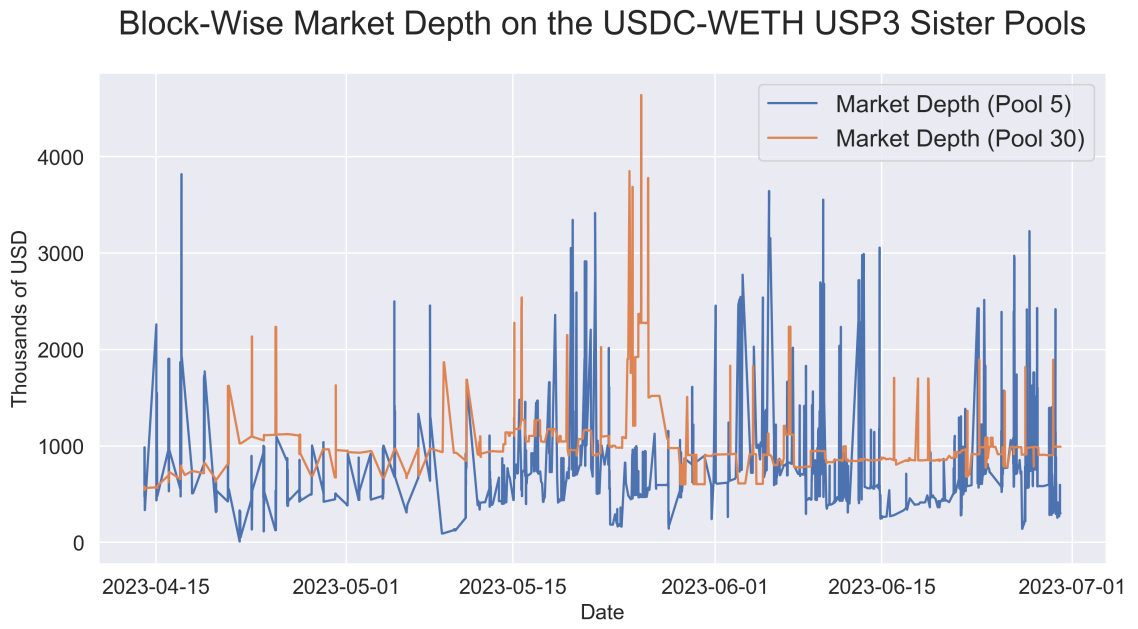



Figure 3.9: Block-wise market depth measurements for the USDC-WETH-0.0005 and USDC-WETH-0.003 Uniswap v3 pools. Only blocks for which snapshots are recorded are included, giving the measurements an artificial jaggedness whereas they are more gradually changing than depicted.

data only has 6,994 snapshots for Pool 5 and 15,921 snapshots for Pool 30 (explaining the relatively smoother appearance)[10]. The solution we implement during feature engi-

[10]We identified this problem with the external data provider, but for external reasons we did not receive

neering is forward-filling the many missing snapshots for each pool (not displayed in Figure 3.9). Alternatives were discarding snapshot-based features (which loses block-wise information about price volatility and market depth) and mean-filling the missing snapshots (which disregards their highly-frequent temporal structure).

### 3.2.3 Closing Remarks

It is worth reiterating that while there are strong theoretical and empirical motivations for focusing our analysis on USP3 and the above two pools, our data constraints are also an important factor in this decision. As we've seen in this chapter, we only have access to block-wise liquidity and price snapshots for USP3 pools, and with no access to data on the USD value of tokens, we have to focus on pools that contain one stablecoin to engineer a common denomination for valuing liquidity. As such, Pool 5 and Pool 30 are the richest settings in terms of the breadth of information we can extract from them and the number of liquidity events we can learn from.

the complete snapshot data.

# Chapter 4

# Feature & Target Engineering

In this chapter, we describe the structured process we develop to engineer blockchain panel data into the features and targets we use in our supervised learning pipelines. We then consider feature and target transformations after inspecting their distributions. Our feature and target engineering involves an extensive sequence of programming steps and combines multiple data sources carefully. We create flexible functions for all steps, so that the engineering process can be generalized to blockchain data of this format for any Uniswap v3 sister pools.

## 4.1 Dataset Engineering Process

Based on the empirical literature around USP3 dynamics discussed in Section 2.2.3, our feature space is inspired by [19]. We also introduce new features which we believe can have predictive power for liquidity dynamics. Based on our discussion about available data in Chapter 3, we can build liquidity and price-related features about Pool 5 and Pool 30, but we cannot build any trade-related features. For each of the two pools, we select it as the main pool from whose liquidity events we make predictions and the other pool as the complement from which we build features and targets that capture spillover effects. We then carry out an engineering process that yields a supervised learning-ready dataset where observations are the main pool's liquidity events. Algorithm 1 presents the engineering process we conduct for each pool. Each feature category we create is highlighted and discussed in the following subsections. The full list of features by category is in Table A.1.

### 4.1.1 Contemporaneous Liquidity Features

For both pools, we first build three contemporaneous features that store relevant information about each liquidity event. These features measure the current event's USD size (*s0*), the tick range of the current event (*w0*), and whether the tick range includes the pool's current price (*liquidity_type*). We also track the current event's *type*.

**Algorithm 1** Feature & Target Engineering Process

**Require:** `main_pool_events`, `other_pool_events`, `main_pool_snapshots`, `other_pool_snapshots`

**Ensure:** `main_pool_engineered`

1: Build **Contemporaneous Liquidity Features** for `main_pool_events` and `other_pool_events`.
2: To obtain **Aggregated Events by Block** for each pool, transform `main_pool_events` and `other_pool_events` as described in Algorithm 2.
3: Build **Lagged Main-Pool Features** for `main_pool_events`.
4: Concatenate `other_pool_events` to `main_pool_events`, ordering events by block number and breaking ties between the two pools by placing the event on the main pool second (such that the event on the other pool isn't considered a future event).
5: Build **Lagged Other-Pool Features** for the post-concatenation `main_pool_events`.
6: Build **Regression & Classification Targets**, then remove `other_pool_events` from the post-concatenation `main_pool_events`.
7: Engineer a snapshot observation for each non-event block in the block number range spanned by `main_pool_events`, then add block-wise price and liquidity information from `main_pool_snapshots` and `other_pool_snapshots`.
8: Build **Price & Market Depth Features** for the snapshot-augmented `main_pool_events`, then remove all snapshot observations.
9: **Return** `main_pool_engineered`, where each observation is a liquidity event on the main pool and all features and targets have been engineered.

### 4.1.2 Block-Wise Aggregated Events

Our lagged features and forward-looking targets use a block-wise clock to simulate the perspective of a USP3 participant. For supervised learning, we must account for the fact that multiple liquidity events can happen on the same block. When making predictions at say, block $Z$, we want to account for all information available at block $Z$. As such, for each pool, we group all liquidity events on the same block into one observation with a principled approach which minimizes the information lost about each event's contemporaneous features. Algorithm 2 represents the procedure for aggregating events block-wise.

We notice that after block-wise aggregation, the net size of liquidity events (*s0*) on a number of blocks is very low. This can at least partly be explained by the phenomenon of **just-in-time liquidity provision**, wherein an LP mints and burns (almost) the same amount of liquidity on the same block, "sandwiching" a trade order on that block to extract a large proportion of the trade's fees without permanently changing the pool's liquidity [51]. This is generally not a concern for traders, as it theoretically lowers price slippage by increasing liquidity around the trade's price. However, our target selection in Section 4.1.5 is meant to track meaningful changes in a pool's liquidity, and preserving observations with minuscule *s0* undermines this objective. Therefore, we only consider liquidity events of size $\geq$ \$100 USD to avoid noising our predictions with nominal liquidity changes. This preserves roughly 70% (80%) of the observations on Pool 5 (30), consistently with empirical findings that just-in-time liquidity provision is more prevalent on Pool 5 [52]. In fact, [52] shows that Pools 5 and 30 account for $\geq$ 70% of all just-in-time liquidity

**Algorithm 2** Aggregating Liquidity Events Block-Wise

**Require:** `pool_events`
**Ensure:** `pool_block_events`

**procedure** AGGREGATE_LIQUIDITY_EVENTS(`pool_events`)
    Group `pool_events` by block number.
    **for** each block in `pool_events` **do**
        Calculate the USD sum of mint operations (`mint_USD`) and the USD sum of burn operations (`burn_USD`).
        Set the block *s0* as $|\texttt{mint\_USD} - \texttt{burn\_USD}|$.
        **if** $\texttt{mint\_USD} > \texttt{burn\_USD}$ **then**
            Record the block as a mint event.
            Set the block *w0* as the *s0*-weighted average of *w0* of the block's mint operations.
            Set the block *liquidity_type* as the *s0*-weighted majority *liquidity_type* of the block's mint operations.
        **else**
            Record the block as a burn event.
            Set the block *w0* as the *s0*-weighted average of *w0* of the block's burn operations.
            Set the block *liquidity_type* as the *s0*-weighted majority of *liquidity_type* of the block's burn operations.
    **Return** `pool_block_events`, wherein at most one liquidity event is recorded per block.

provision on USP3, so our filter on *s0* seems well worth the data loss.

### 4.1.3 Lagged Main-Pool Features

Here, we build lags that capture recent liquidity dynamics for the block-wise aggregated main pool. For $n \in \{1, 2, 3\}$, we create features that track the USD size of, tick range of, and block distance to the $n$-th last liquidity event of each $type \in \{\text{mint, burn}\}$ on the **same** main pool. The number of lags $n$ is chosen based on [19] and the consideration that the number of lagged main-pool features scales as $6n$. We want to limit model complexity to avoid overfitting.

### 4.1.4 Lagged Other-Pool Features

Next, we concatenate the block-wise aggregated events of the other pool to the main pool dataset. We order all events by block number and break ties between the two pools by placing the observations on the other pool first. Since we make predictions from main pool events, this allows us to correctly consider events that happen on the other pool during the same block as part of our knowledge set, rather than future events. The lagged features we build here follow the same structure as the ones built in the previous subsection. Particularly, for $n \in \{1, 2, 3\}$, we create features that track the USD size of, tick range of, and block distance to the $n$-th last liquidity event of each $type \in \{\text{mint, burn}\}$ on

the **other** pool. Through this, we aim to capture predictive power for liquidity dynamics on the main pool through spillover effects from recent dynamics on the other pool.

### 4.1.5 Classification & Regression Targets

We engineer our target variables from the dataset containing liquidity events from both pools. The first category of target variables records the type of the next event in each pool, setting up a binary classification problem. The second category of target variables records the number of blocks until the next event of each type in each pool, setting up a regression problem. All target variables are described in Table 4.1.

Table 4.1: Descriptions of all classification and regression targets we make predictions for. These targets appear in both datasets we engineer, yielding 12 total targets.

| Target | Description |
|---|---|
| **Classification Targets** $\in \{mint, burn\}$ | |
| *next_type_main* | Type of the next liquidity event on the same main pool |
| *next_type_other* | Type of the next liquidity event on the other pool |
| **Regression Targets** $\in \mathbb{N}$ | |
| *next_mint_time_main* | Blocks until the next mint event on the same main pool |
| *next_burn_time_main* | Blocks until the next burn event on the same main pool |
| *next_mint_time_other* | Blocks until the next mint event on the other pool |
| *next_mint_time_other* | Blocks until the next burn event on the other pool |

A preliminary set of targets we considered included measuring the size of incoming mint and burn events in the next $x$ minutes (e.g., 5 minutes, 15 minutes, and 30 minutes). However, this yielded a sparse target setting where the range of the target variable was nevertheless very high (as millions of USD can be minted/burned in minutes). Our regression pipeline performed poorly in this setting, as methods were unable to distinguish the many occasions in which the target takes the value of 0. A potential solution is a two-step approach where we first classify whether an event will happen in the next $x$ minutes, then predict its size if our classifier makes a positive prediction. Instead, our selection of target variables captures the time element of incoming events of both types without imposing arbitrary minute limits, and foregoes the high variance prediction of event size in favor of the binary classification of incoming event type.

### 4.1.6 Price & Market Depth Features

We first drop liquidity event observations from the other pool. Then, we add artificial snapshot observations at blocks where liquidity events don't occur in the main pool, so that our dataset has an observation for each block between the first and last event. This allows us to add snapshot columns measuring each pool's block-wise price and market depth. We build two features to measure the difference (*depth_diff*) and ratio (*depth_ratio*) of market depths between Pool 30 and Pool 5. A third feature measures the price difference (*price_diff*) between the main pool and the other pool. The final features measure price standard deviation (*vol_n*) between the current block and the block at which

the previous $n$-th liquidity event occurs for $n \in \{1, 2, 3\}$. Notably, these features are distorted by our need to forward fill missing snapshots, but we hope they hold informational value nonetheless. Having engineered all features and targets, we remove the artificial snapshot observations, yielding a final dataset where observations are the liquidity events in the main pool.

## 4.2 Feature Analysis

For each of our two datasets, we have a collection of 46 features, of which 44 are numerical and 2 are binary. We analyze features in each dataset, highlighting the motivations for our numerical feature transformations in Chapter 5.

### 4.2.1 The USDC-WETH-0.0005 Dataset

Table 4.2: Summary statistics for selected numerical features in the USDC-WETH-0.0005 dataset, highlighting different feature scales and the presence of outliers.

| **Count: 3,526** | **s0** | **w0** | **b1_mint_main** | **b3_burn_other** | **depth_diff** | **depth_ratio** |
|---|---|---|---|---|---|---|
| **mean** | 572,768.552 | 34,661.487 | 283.348 | 5,565.131 | 251,221.552 | 2.207 |
| **std** | 2,283,654.947 | 236,734.973 | 470.9 | 7,681.062 | 673,410.361 | 3.632 |
| **min** | 100 | 10 | 1 | 49 | -2,512,011.337 | 0.22 |
| **25%** | 3,987.512 | 160 | 44.25 | 1,739.5 | 93,093.504 | 1.115 |
| **50%** | 34,548.002 | 780 | 130 | 3551 | 349,815.819 | 1.646 |
| **75%** | 188,902.026 | 2,077.5 | 308.75 | 6,063.75 | 543,596.046 | 2.292 |
| **max** | 33,640,357.232 | 1,774,540 | 6,796 | 60,731 | 3,365,403.051 | 117.556 |

Table 4.2 shows summary statistics for selected features to highlight their very different scales. We also notice the presence of top-end (e.g., for *w0*) or bottom-end (e.g., for *depth_diff*) outliers for all features. We have no reason to believe any outliers are data errors, as the data underlying the features is gathered directly from the Ethereum blockchain and no values are impossible. As such, we don't consider removing them or applying feature transformations that completely silence them[1].

While tree-based estimators are robust to feature scales, other estimators often require features to have similar scales and are sensitive to outliers. Hence, one of the feature transformations we consider is a `StandardScaler`, which subtracts each numerical feature's mean and scales it to unit variance. The effect of this is shown in Figure 4.1, where two features with vastly different raw scales are converted to comparable scales.

However, this transformation doesn't address skewness. All numerical features except for *depth_diff* are highly skewed compared to a Gaussian distribution. Approximately normally distributed features often improve the performance of many linear models (as the models' key assumptions are more likely to be satisfied), so we also consider a

[1] We experimented with winsorization and it did not improve the test performance of our methods, so we don't include it in our feature transformations.

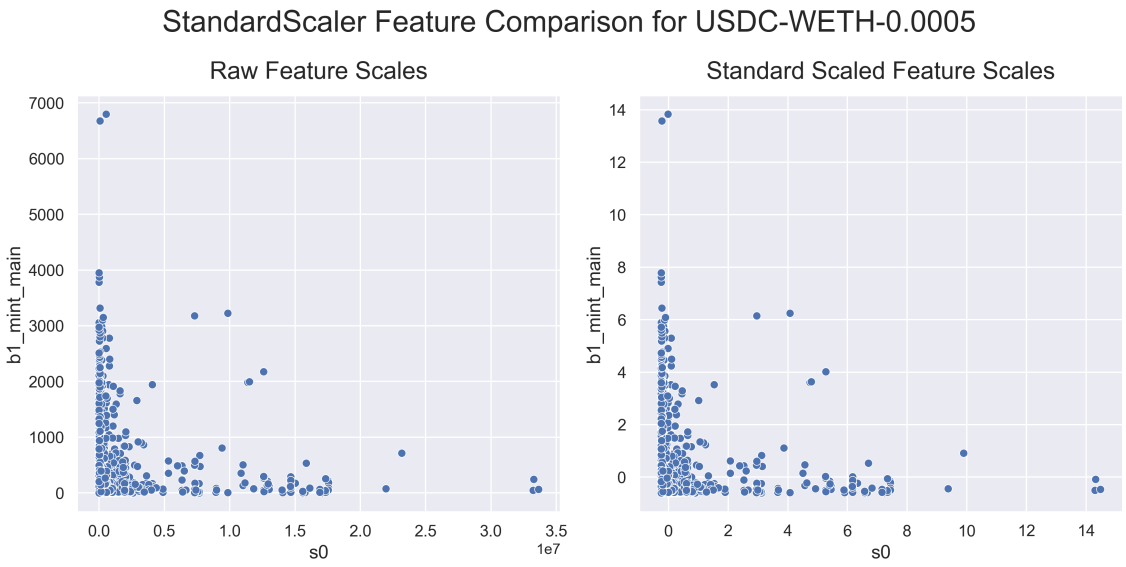



Figure 4.1: Scale comparison for features *s0* and *b1_mint_main* in the USDC-WETH-0.0005 dataset before and after the application of a `StandardScaler`.

`PowerTransformer` in our pipelines. This applies Yeo-Johnson [53] power transformations to features to make them more Gaussian-like and is robust to features that take on negative values (e.g., *depth_diff*), unlike a basic log transformation. Figure 4.2 shows the effect of power transforming the same features as before. We notice both comparable scales and much more Gaussian-like feature distributions. For each (non-tree-based) method in our pipelines, the feature transformation with better time series cross-validation performance is selected.

Finally, we inspect the correlation structure of this pool's numerical features in Figure 4.3. The vast majority of feature correlations are near zero, so we need not worry about collinearity. Other than correlations we expect by construction (e.g., *b1_mint_main* and *b2_mint_main*), we notice positive correlations between the distances of past events in different pools, which hint at spillover effects.

### 4.2.2 The USDC-WETH-0.003 Dataset

Table 4.3: Summary statistics for selected numerical features in the USDC-WETH-0.003 dataset, highlighting different feature scales and the presence of outliers.

| **Count: 672** | **s0** | **w0** | **b1_mint_main** | **b3_burn_other** | **depth_diff** | **depth_ratio** |
|---|---|---|---|---|---|---|
| **mean** | 276,147.892 | 27,174.037 | 1,751.135 | 1,034.641 | 231,412.151 | 2.127 |
| **std** | 2,241,258.119 | 185,225.941 | 2,189.084 | 959.92 | 650,606.956 | 2.545 |
| **min** | 100 | 60 | 1 | 5 | -3,024,826.78 | 0.208 |
| **25%** | 929.579 | 1380 | 327.25 | 368.5 | 75,848.034 | 1.106 |
| **50%** | 5,396.447 | 3450 | 952 | 689.5 | 359,789.587 | 1.653 |
| **75%** | 74,367.426 | 8160 | 2187 | 1418.25 | 517,926.759 | 2.224 |
| **max** | 53,884,543.185 | 1,774,440 | 20,494 | 5,674 | 3,365,403.051 | 19.667 |

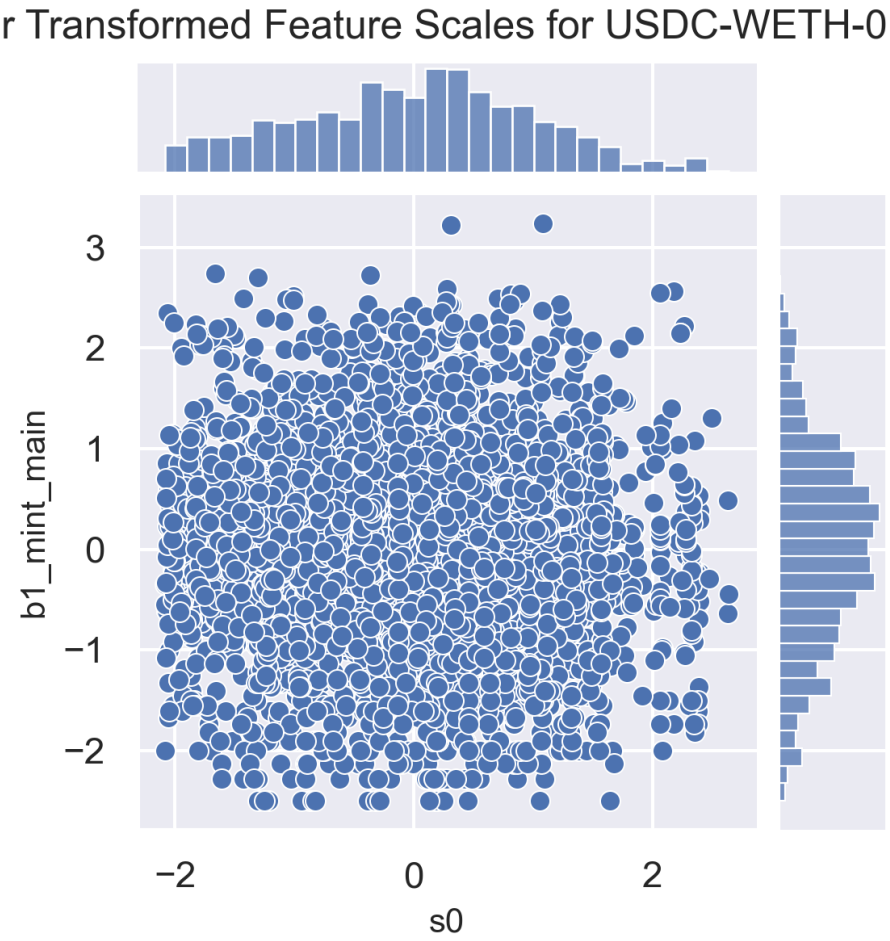



Figure 4.2: Marginal distributions and scatterplot of features *s0* and *b1_mint_main* in the USDC-WETH-0.0005 dataset after applying a Yeo-Johnson `PowerTransformer`.

We proceed with the same numerical feature analysis for Pool 30. A summary statistics inspection (Table 4.3) unsurprisingly reveals the same feature characteristics as for Pool 5: vastly different scales and top-end and bottom-end outliers. Figure 4.4 visualizes the effect of applying a `StandardScaler` to two features in Pool 30 with vastly different raw scales, *w0* and *depth_ratio*. This transformation brings the features to remarkably similar ranges, but does not address high skewness, which again is present in the vast majority of features.

In comparison, Figure 4.5 shows the scale and distribution effect of `PowerTransformer` on these features. The transformed features are on remarkably similar scales and display more Gaussian-like marginal distributions, though less closely so than the marginal distributions in Figure 4.2. This is likely because we have fewer data points for this pool, hinting that this transformation might perform better with more data.

Finally, we display the correlation heatmap of numerical features in Pool 30 in Figure 4.6. We observe an almost identical correlation structure to Figure 4.3, with the caveat that positive correlations between the distances of past events in different pools are slightly weaker. This might be a consequence of dataset size and is not concerning.

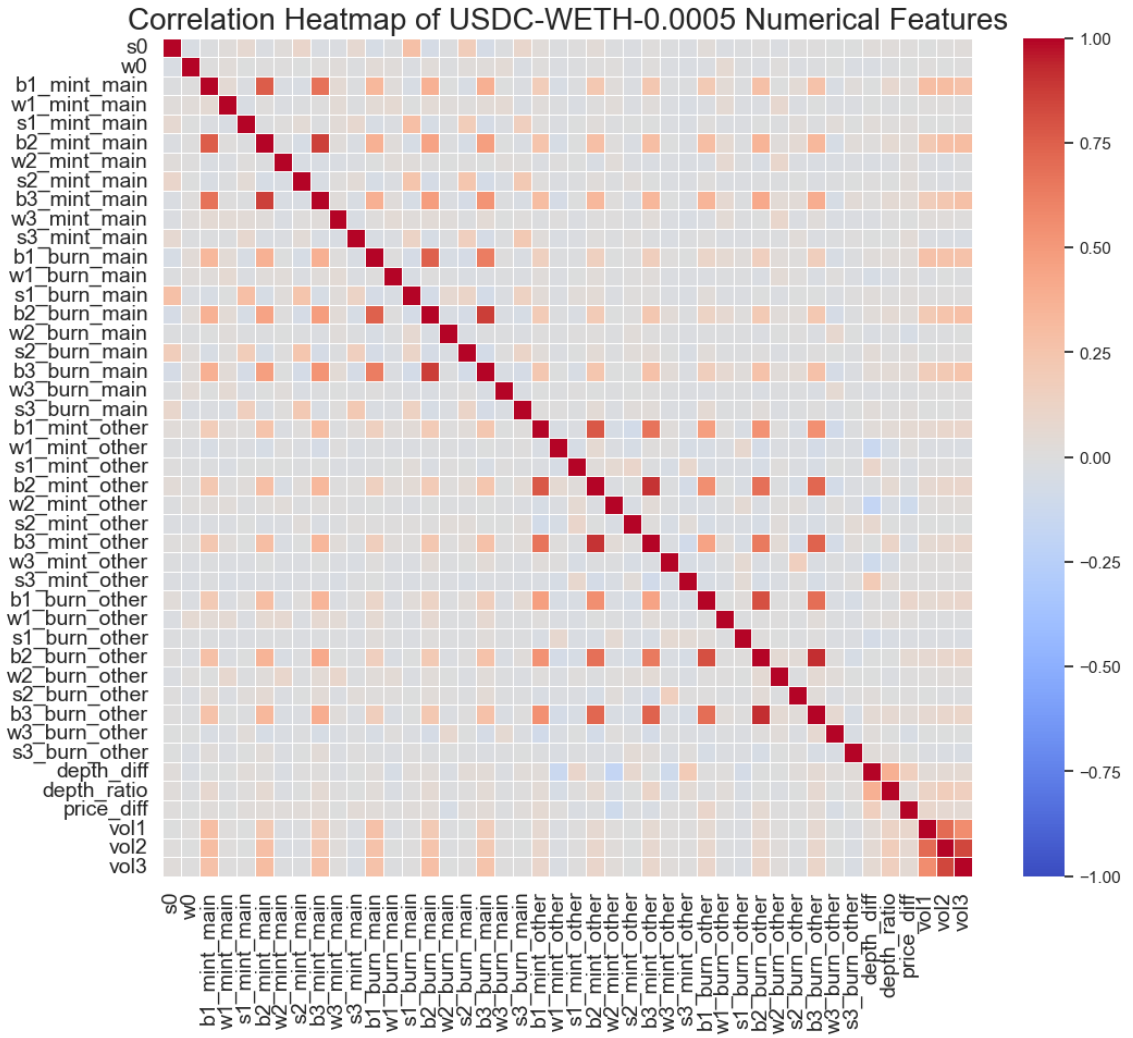



Figure 4.3: Correlation heatmap of numerical features in the USDC-WETH-0.0005 dataset.

## 4.3 Target Analysis

Finally, we analyze the distributions of our engineered targets and consider transformations for regression targets.

### 4.3.1 Binary Classification Targets

For each pool, we have two binary classification targets. We summarize their distributions in Table 4.4 and observe a fairly balanced distribution for each target. Thus, we don't need to worry about preventing any issues due to class imbalance.

### 4.3.2 Regression Targets

For each dataset, we have four regression targets to predict, all of which are natural numbers. We provide summary statistics for the regression targets in Pool 5 in Table 4.5, and

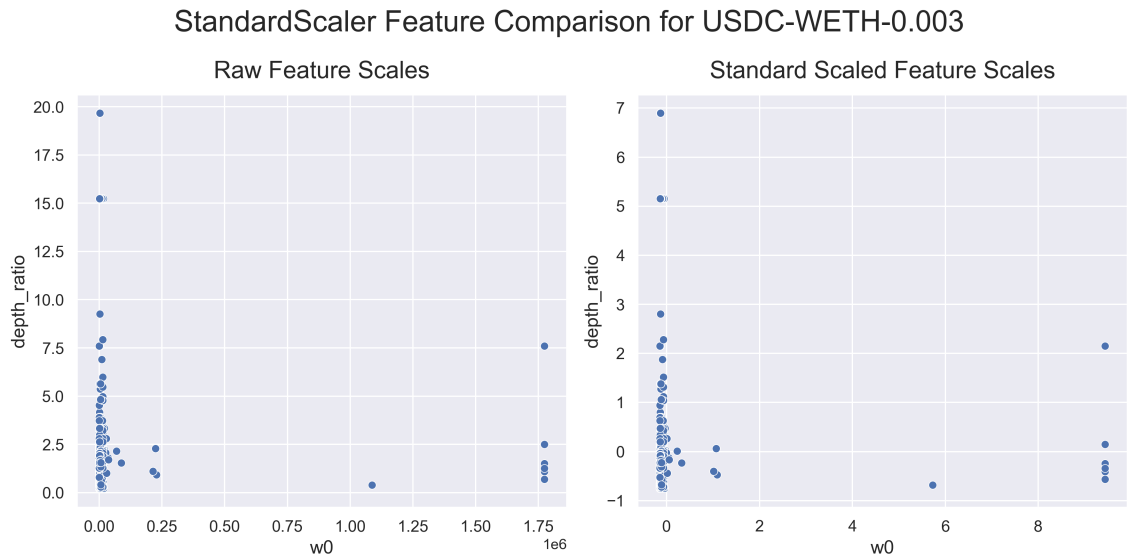



Figure 4.4: Scale comparison for features *w0* and *depth_ratio* in the USDC-WETH-0.003 dataset before and after the application of a `StandardScaler`.

Table 4.4: Distribution of the two classification targets in dataset USDC-WETH-0.0005 and dataset USDC-WETH-0.003.

| **Target** | **Mint Counts** | **Burn Counts** |
|---|---|---|
| Main Pool: **USDC-WETH-0.0005** | | |
| *next_type_main* | 1,881 | 1,645 |
| *next_type_other* | 1,458 | 2,068 |
| Main Pool: **USDC-WETH-0.003** | | |
| *next_type_main* | 304 | 368 |
| *next_type_other* | 332 | 340 |

Table 4.5: Summary statistics for the four regression targets in the USDC-WETH-0.0005 dataset. Actual target names are shortened for cleanliness.

| **Pool 5** | *next_mint_main* | *next_burn_main* | *next_mint_other* | *next_burn_other* |
|---|---|---|---|---|
| **mean** | 257.493 | 333.802 | 1,916.573 | 1,937.682 |
| **std** | 458.205 | 515.408 | 2,354.456 | 4,279.999 |
| **min** | 1 | 1 | 1 | 1 |
| **25%** | 31 | 50 | 351 | 294 |
| **50%** | 95 | 151.5 | 992 | 800 |
| **75%** | 279 | 378.75 | 2,645.75 | 2,077 |
| **max** | 6,796 | 5,478 | 20,423 | 40,562 |

the same for Pool 30 in Table 4.6. These summary statistics reflect that liquidity events of either type happen more often on Pool 5 and more consistently, yielding a lower sample mean and variance for targets tracking events in this pool. Interestingly, on both pools, a mint operation is expected to arrive before a burn operation, though for the Pool 30 we can

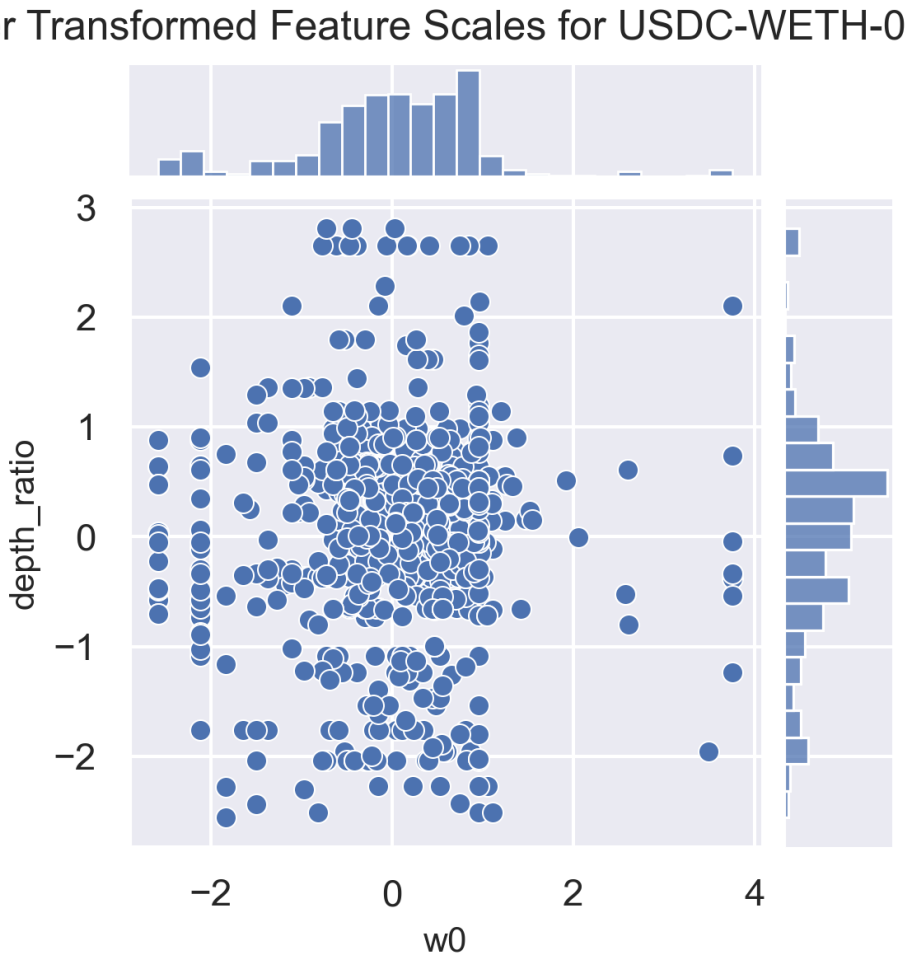



Figure 4.5: Marginal distributions and scatterplot of features *w0* and *depth_ratio* in the USDC-WETH-0.003 dataset after applying a Yeo-Johnson `PowerTransformer`.

see this higher burn arrival expectation is influenced by the maximum value of $\approx 40,560$, which is more than 5 days.

Table 4.6: Summary statistics for the four regression targets in the USDC-WETH-0.003 dataset. Actual target names are shortened for cleanliness.

| **Pool 30** | *next_mint_main* | *next_burn_main* | *next_mint_other* | *next_burn_other* |
|---|---|---|---|---|
| mean | 1,706.374 | 1,827.518 | 279.682 | 316.850 |
| std | 2,238.166 | 4,338.125 | 472.480 | 401.266 |
| min | 2 | 1 | 1 | 1 |
| 25% | 253.75 | 244 | 49.75 | 58 |
| 50% | 853.5 | 760 | 141 | 164 |
| 75% | 2,205 | 1,915.25 | 308 | 428 |
| max | 20,494 | 40,565 | 5,266 | 2,539 |

Although these regression targets are counts, the above statistics suggest overdispersion

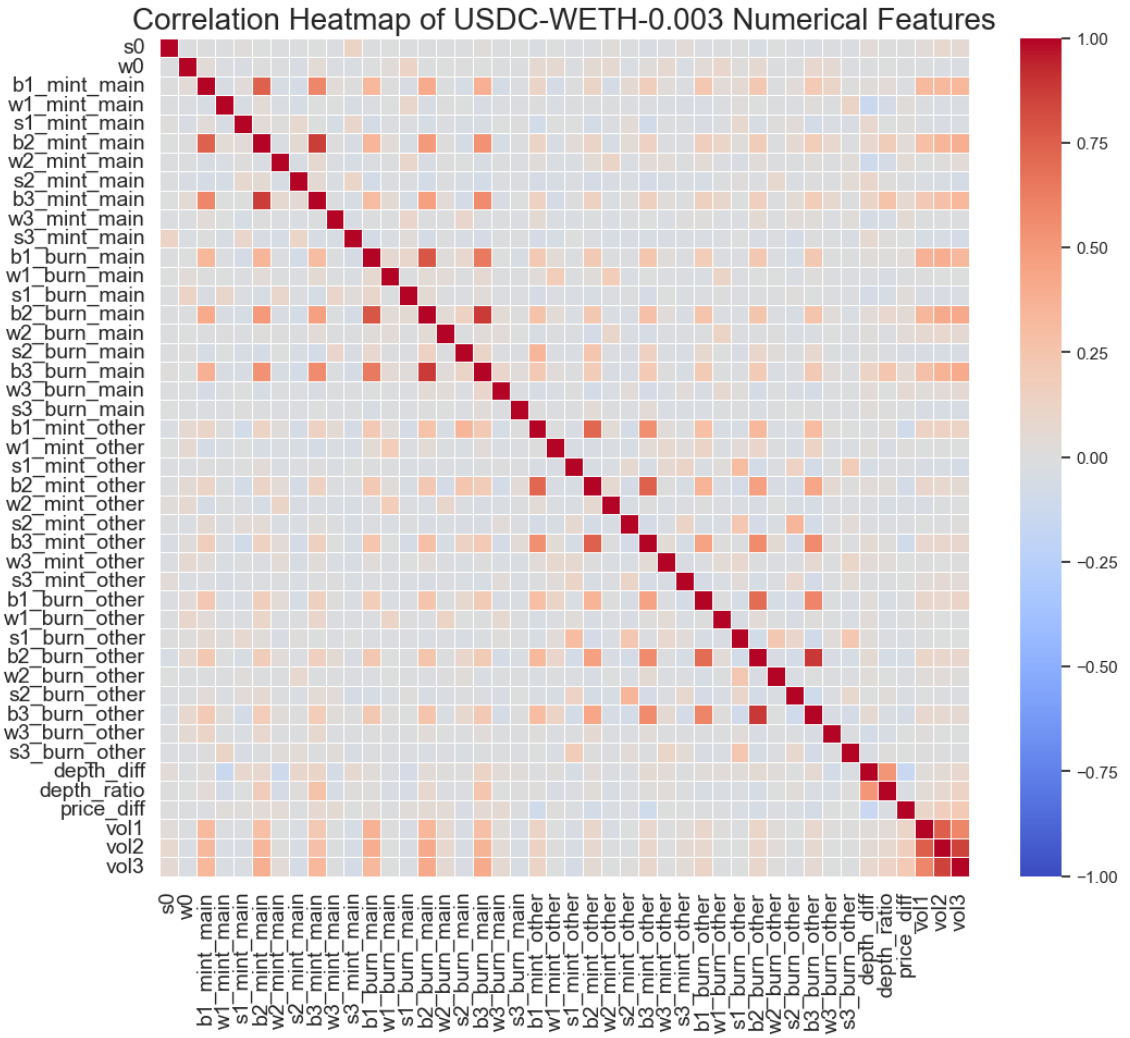



Figure 4.6: Correlation heatmap of numerical features in the USDC-WETH-0.003 dataset.

compared to a Poisson distribution, so using a Poisson generalized linear model (GLM) for prediction may not provide a good fit. We could instead use a negative binomial GLM which accommodates overdispersion, but this assumes a linear relationship between features and the log of expected counts. Because our focus is prediction rather than inference, we take another approach that allows much more flexibility in modelling assumptions.

Specifically, applying a log1p ($y \longrightarrow \log(1+y)$) transformation to our regression targets makes their distribution much more Gaussian-like, especially for targets in Pool 5. Figure 4.7 displays this for the Pool 5 dataset, while Figure 4.8 shows this for the Pool 30 dataset. Normality is desirable for our prediction tasks, as it's likely to improve performance for many estimators and satisfy the residual normality assumption that linear models rely on. The log1p transformation is preferable to one that must be fitted on the data, like Box-Cox, as there are no data leakage concerns and we preserve interpretability in our predictions. Thus, we apply this transformation to our regression targets and explore a comprehensive range of supervised learning methods in the next chapter. Using this approach, we navigate models that assume a linear relationship between features and

the log of expected counts, but also many more flexible methods like tree-based ensembles and neural networks.

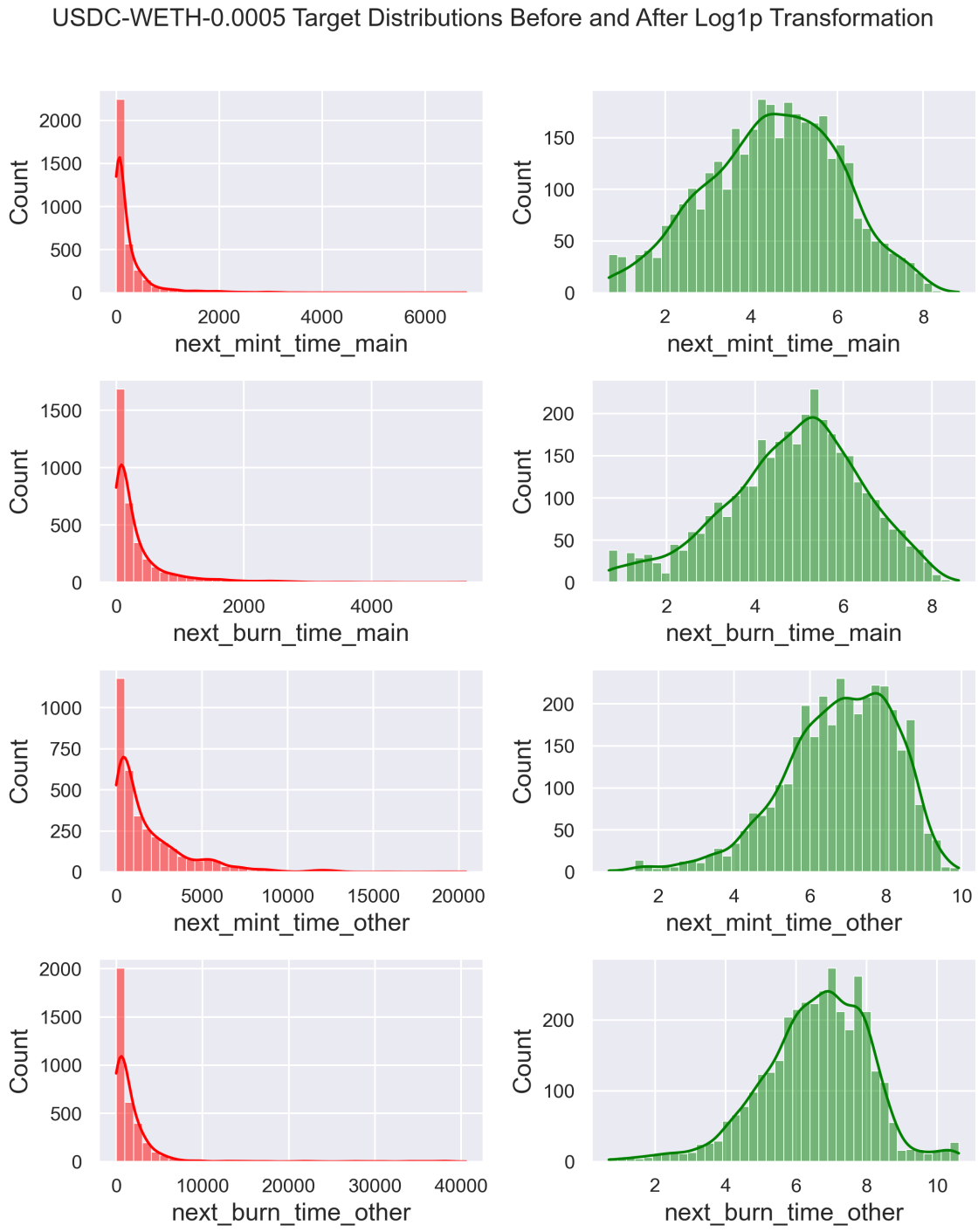



Figure 4.7: Marginal distributions of the four regression targets in the USDC-WETH-0.0005 dataset before and after applying a log1p ($y \longrightarrow \log(1+y)$) transformation.

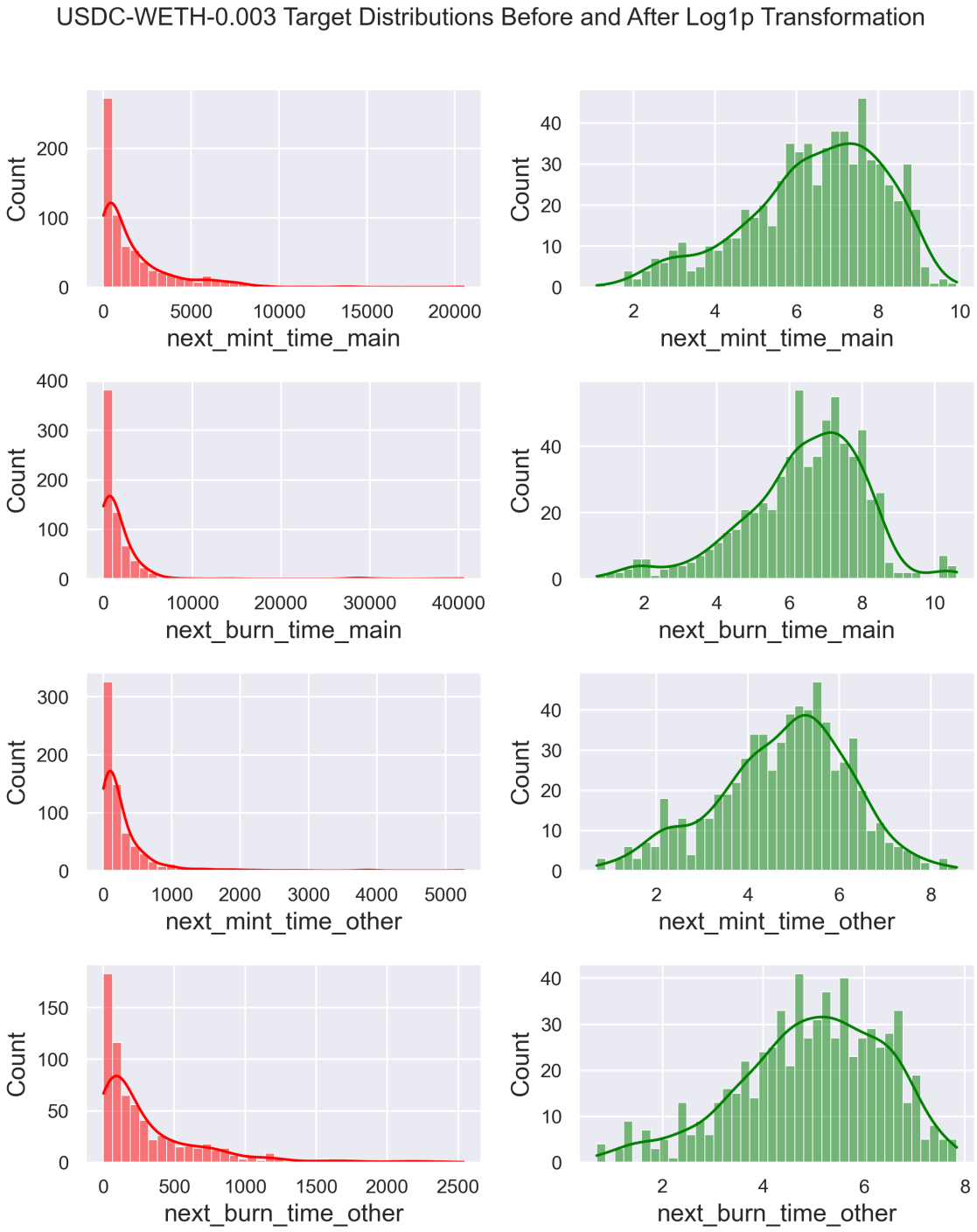



Figure 4.8: Marginal distributions of the four regression targets in the USDC-WETH-0.003 dataset before and after applying a log1p ($y \longrightarrow \log(1+y)$) transformation.

# Chapter 5

# Methodology

In this chapter, we describe the supervised learning pipelines we build for our classification and regression tasks in order to generate unbiased estimates for the generalization performance of a comprehensive range of methods. We then discuss the individual methods we train, giving attention to their merits and hyperparameter spaces.

## 5.1 Supervised Learning Pipelines

We apply the procedures discussed in this section to all 12 of our feature-target combinations. For notation, we consider an $(n \times p)$ feature matrix $X$ and an $(n \times 1)$ target vector $y$.

### 5.1.1 Time Series Data Splits & Cross-Validation

Our observations are temporally ordered by block number and our targets pose a forecasting problem. Hence, we use the first 80% of observations as training/validation data and hold the final 20% as test data to estimate generalization performance on. To tune hyperparameters, we opt for `TimeSeriesSplit` cross-validation with 5 folds on the training/validation split, which preserves the setting's time structure by simulating forecasting, as we depict for a toy example in Figure 5.1. These choices prevent temporal leakages when training estimators and when estimating their generalization performance. They seamlessly integrate into our pipelines, which we describe next.

### 5.1.2 Model Selection & Training

For each target, we compare the performance of a comprehensive set of appropriate methods in a systematic way. The classification pipeline includes 11 methods, while the regression pipeline includes 13, all of which are discussed in Section 5.2. The end-to-end pipeline for any $X - y$ pair is described by Algorithm 3, where the inputs `models` and `param_grids` are specific to whether $y$ is a classification or regression target. For each method, we carefully specify hyperparameter distributions to search over on the training/validation split using `RandomizedSearchCV`, which also implements `TimeSeriesSplit`

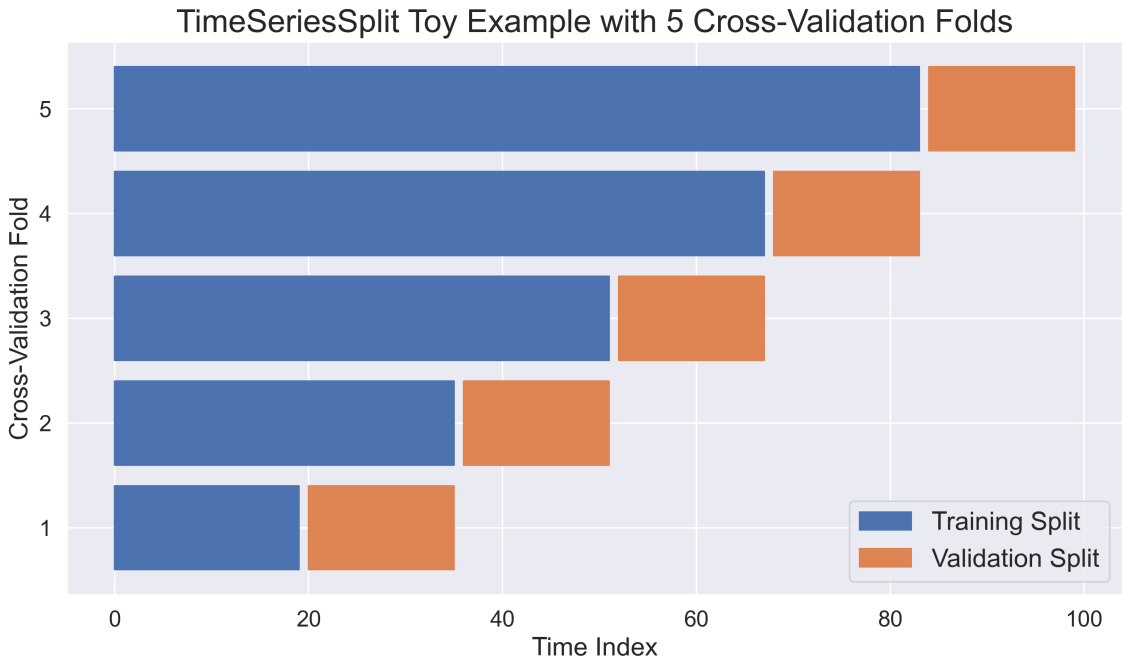



Figure 5.1: Toy example demonstrating how `TimeSeriesSplit` validation preserves the time structure of a forecasting problem during hyperparameter tuning.

cross-validation. Compared to exhaustive grid search where we must define exact hyperparameter values, `RandomizedSearchCV` samples from hyperparameter distributions for a specified number of iterations (100 in our case[1]). Thus, we navigate a much broader hyperparameter space to find optimal configurations while having explicit control over computational expenses [54]. For non-tree-based methods, `RandomizedSearchCV` also includes feature transformations `StandardScaler` and `PowerTransformer` based on Section 4.2.

For each method, `RandomizedSearchCV` saves the estimator with the best `TimeSeriesSplit` cross-validation performance, refitted on the entire training/validation split. We make predictions with this `model_best_estimator` on the test split to estimate generalization performance and on the training/validation split to track potential dataset shift and/or overfitting. Our chosen performance metrics for classification and regression are described in Table 5.1. Accuracy as the performance metric for binary classification is a natural choice. For regression, we opt for mean squared logarithmic error (MSLE) instead of the standard $R^2(y, \hat{y}) = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$, where $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$. $R^2$ is not comparable across our regression $X - y$ pairs, as many target variances are quite different (see Table 4.5 and Table 4.6). We use the MSE metric for our log1p-transformed targets, which corresponds to MSLE for liquidity arrival times, directly translating to the setting we are interested in, unlike other metrics. This metric penalizes under-predictions more than over-predictions [55], which is desirable in some scenarios (e.g., a trader programs a swap after a predicted mint operation to reduce price slippage). More importantly, using MSLE as our regression metric balances fitted methods to yield similar percentage errors in settings like ours,

[1]With 5 time series cross-validation folds, this yields 500 fits for each method, which balances tuning performance with limiting search time well.

**Algorithm 3** Supervised Learning Pipelines (Classification & Regression)

**Require:** `X`, `y`, `models`, `param_grids`
**Ensure:** `pipeline_results`

**procedure** SUPERVISED_LEARNING_PIPELINE(`X`, `y`, `models`, `param_grids`)
    Create 1 dummy column in `X` for each binary categorical feature
    **if** $\texttt{y} \in \mathbb{N}$ **then**
        $\texttt{y} := \log(1 + \texttt{y})$
    Reserve the first 80% of observations in `X`, `y` as `X_train_val`, `y_train_val` and assign the last 20% as `X_test`, `y_test`
    Create an empty `pipeline_results` DataFrame
    **for** `model` in `models` **do**
        **if** `model` is not tree-based **then**
            Add feature transformations `StandardScaler` and `PowerTransformer` for numerical (non-dummy) columns to `model_param_grid`
        Train the `model` via `RandomizedSearchCV` (100 iterations) over `model_param_grid` with a `TimeSeriesSplit` using data `X_train_val`, `y_train_val`
        Store `model_best_params`, `model_param_search_time`, and `model_best_estimator` (which has been refitted on the whole training/validation split)
        Make predictions on `X_train_val`, `y_train_val` using `model_best_estimator` and store `model_train_val_score`
        Make predictions on `X_test`, `y_test` using `model_best_estimator` and store `model_prediction_time` and `model_test_score`
        Append `model_name`, `model_test_score`, `model_train_val_score`, `model_param_search_time`, `model_prediction_time`, and `model_best_params` as a row to `pipeline_results`
    **Return** `pipeline_results`

where the target spans different orders of magnitude [56].

Table 5.1: Performance metric choices for the classification and regression setting, alongside their respective formulae for a dataset $d = \{x_i, y_i\}_{i=1}^n$, where $\hat{y}$ are predictions.

| **Setting** | **Performance Metric** | **Formula** |
|---|---|---|
| Classification | Accuracy | $\frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\{\hat{y}_i = y_i\}$ |
| Regression | Mean Squared (Logarithmic) Error | $\frac{1}{n}\sum_{i=1}^{n}(\log(\hat{y}_i + 1) - \log(y_i + 1))^2$ |

At the end of the loop through methods in Algorithm 3, we have the hyperparameter search time, test split prediction time, and best estimator training/validation split and test split accuracy/MSLE for all methods considered for target $y$. We compare methods in these dimensions and note that their test split accuracy/MSLE is an unbiased estimate of their generalization accuracy/MSLE [57]. Thus, for each target, we select the tuned estimator with the best test split accuracy/MSLE and conduct model evaluation.

### 5.1.3 Model Evaluation

For all targets, we plot the target distribution and the chosen estimator's prediction distribution for the test split and training/validation split. This sheds light on potential dataset shift/overfitting and whether the estimator makes predictions in a different range/shape than the target. To quantify distribution differences for regression targets, we compute Kullback-Leibler (KL) divergence:

$$\mathcal{D}_{KL}(P \,||\, Q) = \int_{\mathcal{X}} p(x) \log\left(\frac{p(x)}{q(x)}\right) dx \tag{5.1}$$

from kernel density estimates (KDEs) of target distributions ($P$) to KDEs of prediction distributions ($Q$) for the test split and training/validation split. For all targets, we also plot top 10 mean feature permutation importances after 50 permutation loops for the test split. A feature's permutation importance is the change in accuracy/MSLE from its values being randomly shuffled [58]. This sheds light on whether a target's best estimator identifies features with high predictive power and contextualizes its predictions with respect to intuition and the literature.

For classification targets, we evaluate estimator predictions on the test split by building unnormalized and normalized confusion matrices and reporting key classification metrics. For each class $\in$ {mint, burn}, we report Precision $= \Pr(Y = \text{class} | h(X) = \text{class})$, Recall $= \Pr(h(X) = \text{class} | Y = \text{class})$, and F1 $= \frac{2}{\frac{1}{\text{Precision}} + \frac{1}{\text{Recall}}}$, where $h(X)$ denotes an estimator's prediction. Finally, we plot the chosen estimator's ROC curve and measure AUC, where the positive class is mint. AUC is interpreted as the probability the estimator assigns a higher mint class probability to a random mint observation than to a random burn observation [57].

For regression targets, we evaluate estimator predictions on the test split by plotting residuals and target values against predictions. A desirable residuals vs. predictions plot displays residuals randomly and homoskedastically scattered around 0, so as to not indicate model misspecification which lowers prediction accuracy and/or robustness. A desirable target vs. predictions plot displays points scattered tightly around a $45°$ line, indicating predictions match the target direction precisely.

## 5.2 Supervised Learning Methods

In this section, we discuss the merits and hyperparameter configurations of the methods we include in our supervised learning pipelines. Appropriate supervised learning methods explored during the MSc are included alongside gradient boosting methods known for their state-of-the-art performance in tabular settings. Methods are implemented in `scikit-learn`, seamlessly integrating into our pipeline. For notation, we consider dataset $d = \{x_i, y_i\}_{i=1}^{n}$, where $x_i$ is $(1 \times p)$.

### 5.2.1 Classification Methods

For classification, we map $y \in$ (burn, mint) to $y \in (-1, 1)$ (except for logistic loss) and seek to learn a prediction rule for $x \in \mathcal{X}$:

$$h(x) = \begin{cases} 1 & f(x) \geq 0 \\ \text{-1} & f(x) < 0, \end{cases} \tag{5.2}$$

where $f : \mathcal{X} \to \mathbb{R}$ is a discriminant function. The method we are implementing defines a hypothesis class $\mathcal{H}$ restricting the form of $h$.

#### Tree-Based Methods

Tree-based methods are robust to features that vary on different scales, allowing us to forgo feature transformations when implementing them and effectively doubling the amount of hyperparameter combinations we sample for a feature representation compared to non-tree-based methods. The most basic estimator we consider is a `DecisionTreeClassifier`, which partitions $\mathcal{X}$ into $R$ disjoint regions $\mathcal{P} = \{\mathcal{R}_1, ..., \mathcal{R}_R\}$ such that $h(x) = \beta_j$ $\forall x \in \mathcal{R}_j$. Split quality is measured by Gini impurity:

$$2\eta_1(1 - \eta_1), \tag{5.3}$$

where $\eta_1 = \frac{\sum_i \mathbb{1}\{y_i=1\}\mathbb{1}\{x_i \in \mathcal{R}\}}{\sum_i \mathbb{1}\{x_i \in \mathcal{R}\}}$ for some region $\mathcal{R}$. Decision trees are quickly trained and easily interpretable, but prone to overfitting. Hence, our hyperparameter search explores maximum tree depth and the minimum number of observations per leaf for regularization.

Our next estimator is a `RandomForestClassifier` making predictions through an ensemble of independently trained decision trees with the Gini measure of node impurity. Instead of taking the majority vote of tree classifications like [58], `scikit-learn` averages the trees' probability predictions, incorporating more information. Random forests are easy to use and known for their strong performance in many tabular settings [59]. We regularize by making trees use a bootstrap 75% observation subsample and split nodes considering a random subset of $p_{max}$ features. No tree pruning is necessary [60]; instead, we search over the number of trees and whether $p_{max}$ is $\sqrt{p}$, $log_2(p)$, or $p$ (bagging), as `scikit-learn` recommends.

The last `scikit-learn` tree-based method we implement is `AdaBoostClassifier`, where the discriminant function takes the form $f(x) = \sum_{t=1}^{T} \beta_t h_t(x)$ and $h_t : \mathcal{X} \to \{-1, 1\}$ are weak classifiers. The prediction rule is a weighted majority vote of $T$ weak classifiers $\{h_t\}_{t=1}^{T}$ fitted in a sequence, with each classifier's weight $\beta_t$ determined by weighted accuracy. $h_1$ is trained using uniform observation weights, and new classifiers increase (lower) the weights of misclassified (correctly classified) observations from the previous iteration [61]. Thus, subsequent classifiers focus on "difficult" observations. Following `scikit-learn`'s advice, we search over $T$ and whether $\{h_t\}_{t=1}^{T}$ are decision stumps or depth 3 trees.

The other tree-based methods we consider implement *gradient boosting* [62], which also involves an ensemble of sequentially-built weak classifiers. New weak classifiers are built such that their correlation with the negative gradient of the loss function of the current

ensemble is maximized [63], so that the ensemble improves through a process like functional gradient descent. A mathematical description of gradient boosting is in [64]. Gradient boosting is known for its leading performance ([65], [66]) and ubiquity [67] in tabular settings. We implement the three most popular variants: XGBoost [68] (`XGBClassifier`), LightGBM [69] (`LGBMClassifier`), and CatBoost [70] (`CatBoostClassifier`). In each variant, we use trees as weak classifiers and optimize logistic loss,

$$L(y, h(x)) = \sum_{i=1}^{n} [y_i \log(1 + \exp(-h(x_i)) + (1 - y_i) \log(1 + \exp(h(x_i))], \tag{5.4}$$

where $y \in \{0, 1\}$. Previous studies have compared their training speed and accuracy for a specific dataset [71] and for a wide variety of datasets, with attention to hyperparameter tuning [72]. They generally find similar performance and notice that the top performer is dataset-specific. [72] highlights hyperparameter choices are impactful for these methods, so we incorporate literature insights into all hyperparameter distributions. Particularly, following [72], we upper bound the number of trees (`n_estimators`) at 200 for computational speed for all methods. Following discussion with an author from [41], we upper bound tree depth at 6 for computational speed and search over L1 and L2 regularization for all methods.

An attractive feature of `XGBClassifier` is that it incorporates regularization in the objective function by controlling tree complexity, which helps make it highly scalable. We fix the subsample of training observations to 75% and $p_{max} = \sqrt{p}$ as [72], [73] find beneficial. Aside from previously mentioned hyperparameters, we search (based on [72]) over the `learning_rate` with which new trees contribute to the ensemble and the `min_split_loss` required to further split a leaf node, which is part of XGBoost's distinctive regularization term.

`LGBMClassifier` grows trees leaf-wise [74] rather than the standard depth-wise growth, which tends to achieve lower loss, but can cause overfitting in small datasets [75]. We leverage *Gradient-based One-Side Sampling* (GOSS), LightGBM's subsampling innovation meant to boost training speed without sacrificing accuracy [69], and fix related hyperparameters according to [72]. Instead of depth, we search over the maximum number of leaves in a tree (*num_leaves*), which controls tree depth under leaf-wise growth. Alongside the other 3 common hyperparameters, we search over `learning_rate`, the maximum number of discrete bins (`max_bins`) that features are bucketed into to reduce training time (and accuracy), and `feature_fraction_bynode`, the leaf-wise growth equivalent of $p_{max}$.

A main focus of CatBoost is *ordered boosting*, which combats *prediction shift*, a kind of target leakage endemic to gradient boosting that increases generalization error [70]. Notably, weak classifiers are decision tables: decision trees restricted to use the same split criterion for all nodes on a level. Compared to classic trees, they have empirically provided superior accuracy and speed as weak classifiers for boosted ensembles in classification and regression problems [76]. We implement `CatBoostClassifier` using ordered boosting and search over `learning_rate`, $p_{max}$, and the number of gradient steps used to calculate leaf values (*leaf_estimation_iterations*) alongside the usual `n_estimators`, depth, and L2 (there is no L1 regularization for CatBoost). [72] finds

that tuning `learning_rate` and depth is sufficient, so our hyperparameter search is extra robust.

### Other Methods

Here we consider a range of linear and distance-based classifiers and neural networks. All of these methods can benefit from our feature transformations, so we include them in their hyperparameter searches.

We first consider two linear classifiers with discriminant functions of the form $f(x_i) = x_i w$, where $w \in \mathbb{R}^p$ (intercept term included). Linear classifiers are attractive for their interpretability and resistance to overfit due to simplicity. We implement `RidgeClassifier`, which optimizes the objective:

$$w^* = \min_w \|Xw - y\|_2^2 + \alpha \|w\|_2^2, \tag{5.5}$$

and we search over a log-uniform distribution of $\alpha$. We then implement `LogisticRegression` with L2 regularization, which numerically solves:

$$w^* = \min_w C \sum_{i=1}^{n} (-y_i \log(\hat{p}(X_i)) - (1 - y_i) \log(1 - \hat{p}(X_i))) + \frac{1}{2}\|w\|_2^2, \tag{5.6}$$

where $y \in \{0, 1\}$ and

$$\Pr(y_i = 1 | X_i) = \hat{p}(X_i) = \text{sig}(X_i w) = \frac{1}{1 + \exp(-X_i w)}.$$

This can be interpreted as empirical risk minimization under logistic loss or as a Bernoulli GLM with a logit link. Our hyperparameter search is over regularization strength $C$ and the maximum number of iterations for the robust quasi-Newton-Raphson solver used by `scikit-learn`.

Next, we consider two distance-based classifiers which can generate highly flexible non-linear decision boundaries. `SVC` aims to find a decision boundary that maximizes the margin of separation between observations from opposite classes and minimize the amount of misclassified observations. For some *feature map* $\phi(\cdot)$, we have prediction rule

$$h(x) = \text{sign}(\phi(x)w + b) = \text{sign}(\sum_{i=1}^{n} \alpha_i y_i k(x, x_i) + b), \tag{5.7}$$

where $w, b$ are parameters in the primal problem which balances margin maximization with misclassification, $\{\alpha_i\}_{i=1}^n$ are dual variables, and $k(x, x_i) = \phi(x)\phi(x_i)^T$ via the *kernel trick* [77], allowing us to introduce infinitely-dimensional non-linear feature maps of $x$. We implement `SVC` searching over the RBF and polynomial (degree 3) kernels. It is particularly important [77] to search over hyperparameters `C`, which balances the simplicity of the decision function against misclassification, and `gamma`, which manages the radius of influence of support vectors. Secondly, we implement `KNeighborsClassifier`, where an observation's predicted target class is a majority vote of the target classes of its k-nearest neighbors. Hyperparameter search over $k$ is paramount for performance. We

also search over uniformly-weighted neighbor votes or distance-weighted neighbor votes and whether distance is defined as Euclidean distance ($\rho(x, x') = \|x - x'\|_2$) or Manhattan distance ($\rho(x, x') = \|x - x'\|_1$).

Finally, we implement neural networks trained by backpropagation via `MLPClassifier`. While deep learning outperforms other methods for huge datasets with high dimensional homogeneous inputs [78], tree ensemble methods provide superior performance and training speed for smaller tabular settings ([65], [66]) like ours, so we only consider this method loosely and refer to [79] for its parameterization. We use the relu activation function, defined as $f(x) = max(0, x)$, for the hidden layer(s), and the ADAM optimizer [78] for determining weights. We search over different architectures, including networks with one hidden layer and deep feedforward neural networks, and the L2 regularization parameter `alpha` following `scikit-learn` guidelines.

### 5.2.2 Regression Methods

For regression, we have $y \in \mathbb{R}$ (after the log1p transformation), so prediction rules are $h(x) \in \mathbb{R}$ and don't require a discriminant function.

#### Tree-Based Methods

All of the tree-based methods we employ for classification are extended to regression. We first implement a `DecisionTreeRegressor` which generates the same prediction rule as its classification counterpart, and where split quality is measured by squared error. Like in classification, we search over maximum depth and the minimum number of observations per leaf for regularization. Then we use a `RandomForestRegressor` wherein each tree uses a squared error splitting criterion, a bootstrap 75% observation subsample, and a random subset of $p_{max}$ features for each split. Like for our `RandomForestClassifier`, we search over `n_estimators` and $p_{max}$. Finally, we implement `AdaBoostRegressor` with squared loss, yielding L2-Boosting [61]. As in classification, we tune `n_estimators` and the depth of these base trees.

All of the gradient boosting libraries we use have regressor variants, namely `XGBRegressor`, `LGBMRegressor`, and `CatBoostRegressor`. In each variant, we use trees as weak regressors and optimize squared loss,

$$L(y, h(x)) = \sum_{i=1}^{n} (y_i - h(x_i))^2. \tag{5.8}$$

Identically to the classification case, we take advantage of the distinctive regularization term for `XGBRegressor`, of GOSS for `LGBMRegressor`, and of ordered boosting for `CatBoostRegressor`. The hyperparameter searches for these methods are identical to the classification case, so we refer to the discussions about `XGBClassifier`, `LGBMClassifier`, and `CatBoostClassifier` above.

#### Other Methods

Similarly to the classification case, we consider here a range of linear models, distance-based regressors, and neural networks. Feature transformations are important for each of

these methods, so they are included in the hyperparameter search.

Linear models have prediction rule $h(x_i) = x_i w$, where $w \in \mathbb{R}^p$ (intercept term included). We first consider `Ridge`, `scikit-learn`'s implementation of ridge regression, where we optimize the same objective (Equation (5.5)) as in `RidgeClassifier` and search over a log-uniform distribution of $\alpha$. We then implement the estimator `Lasso`, which optimizes:

$$w^* = \min_w \frac{1}{2n} \|Xw - y\|_2^2 + \alpha \|w\|_1 \tag{5.9}$$

and similarly search over regularization parameter $\alpha$. This model combats overfitting by estimating sparse coefficients, while ridge regression only shrinks their magnitude. Hence, we combine L1 and L2 regularization via `ElasticNet`, optimizing:

$$w^* = \min_w \frac{1}{2n} \|Xw - y\|_2^2 + \alpha\rho \|w\|_1 + \frac{\alpha(1-\rho)}{2} \|w\|_2^2 \tag{5.10}$$

and search over both $\alpha$ and $\rho$, the hyperparameter controlling the relative strength of L1 and L2 regularization.

Distance-based regressors introduce highly flexible prediction rules. We first implement `KernelRidge` and `SVR`. Both methods use kernel functions to capture complex non-linear relationships between features and the target, alongside L2 regularization. However, they use different loss functions. `KernelRidge` optimizes:

$$h^* = \arg\min_{h \in \mathcal{H}_k} (\sum_{i=1}^{n} (y_i - h(x_i))^2 + \alpha \|h\|_{\mathcal{H}_k}^2), \tag{5.11}$$

where $\mathcal{H}_k$ is a reproducing kernel Hilbert space (RKHS) implicitly defined by our kernel choice [77]. It's clear this objective minimizes the squared loss. Our hyperparameter search is over regularization strength $\alpha$, over whether we use an RBF or polynomial (degree 3) kernel, and over the kernel hyperparameter *gamma*. On the other hand, `SVR` uses $\epsilon$-insensitive loss:

$$L(y_i, h(x_i)) = \begin{cases} 0, \text{ if } |y_i - h(x_i)| \leq \epsilon \\ |y_i - h(x_i)| - \epsilon, \text{otherwise} \end{cases} \tag{5.12}$$

for a specified tolerance $\epsilon$, which influences the number of support vectors determining the method's predictions. Thus, for `SVR`, we conduct a search over values of $\epsilon$ alongside all hyperparameters we also search over for `KernelRidge`. Finally, we implement `KNeighborsRegressor`, where an observation's predicted target is the mean target of its k-nearest neighbors. Similarly to `KNeighborsClassifier`, we search over $k$, whether to weigh neighbor contributions uniformly or inversely to their distance, and whether distance should be Euclidean or Manhattan.

Finally, we implement neural networks via `MLPRegressor`, which trains by backpropagation and targets the squared error, and we again refer to [79] for the parameterization. Like in the classification case, we use the relu activation function for the hidden layer(s) and ADAM for optimizing weights. We conduct the same hyperparameter search as for `MLPClassifier`, navigating between single hidden layer and deep feedworward neural networks and varying L2 regularization strength.

# Chapter 6

# Classification Results & Model Evaluation

In this chapter, we analyze the results of the supervised learning pipeline described in Chapter 5 for our 4 classification targets. For each target, we select the estimator with the best test split accuracy and evaluate its predictions compared to the true target, then carry out a feature importance analysis. We compare our predictions $\hat{y}_i = \hat{h}(x_i)$ against a naive persistence model:

$$\hat{y}_i = y_{i-1}, \tag{6.1}$$

which is a standard forecasting baseline preferred to predicting the most common class, which ignores the data's time structure [80]. Detailed supervised learning performance tables for each target are in Appendix B. We provide 95% Wald confidence intervals (C.I.s) for each best estimator test accuracy, but consider them loosely because they wrongly assume liquidity events in the test split are independent. Tables also list the training/validation accuracy of best estimators to identify potential dataset shift/overfitting issues. Information about computational speed[1] and optimal hyperparameter configurations is recorded too.

## 6.1 Main Pool: USDC-WETH-0.0005 (Pool 5)

In this section, we predict the next type of liquidity event in both pools from liquidity events in Pool 5.

Table 6.1: Test split accuracy of the best estimator and persistence baseline model for targets *next_type_main* and *next_type_other*, where the main pool is USDC-WETH-0.0005.

| Target | Best Estimator | Best Acc. | Baseline Acc. |
|---|---|---|---|
| *next_type_main* | `RidgeClassifier` | 0.5354 | 0.4618 |
| *next_type_other* | `DecisionTreeClassifier` | 0.5567 | 0.4802 |

[1] Computational speed is recorded in the tables for running the supervised learning pipelines of all targets at the same time.

### 6.1.1 Target: *next_type_main*

Table B.1 gives a comprehensive summary of the performance of each classifier described in Section 5.2.1 for predicting *next_type_main* from the Pool 5 dataset. All estimators display practically low generalization accuracy. A simple `RidgeClassifier` offers the highest accuracy (0.5354), hinting at this setting's difficulty. A simple linear model is less negatively affected by potential dataset shift than methods which learn more complex decision rules on the training/validation split. Our `RidgeClassifier` outperforms the persistence model test accuracy of 0.4618 in Table 6.1 (though so does random classification). It opts for a `StandardScaler` transformation, as do all but one non-tree-based methods for this dataset. The `RidgeClassifier` also benefits from the fastest hyperparameter search time among non-tree-based methods, while tree-based methods perform poorly on the test split, with all but one always predicting mints.

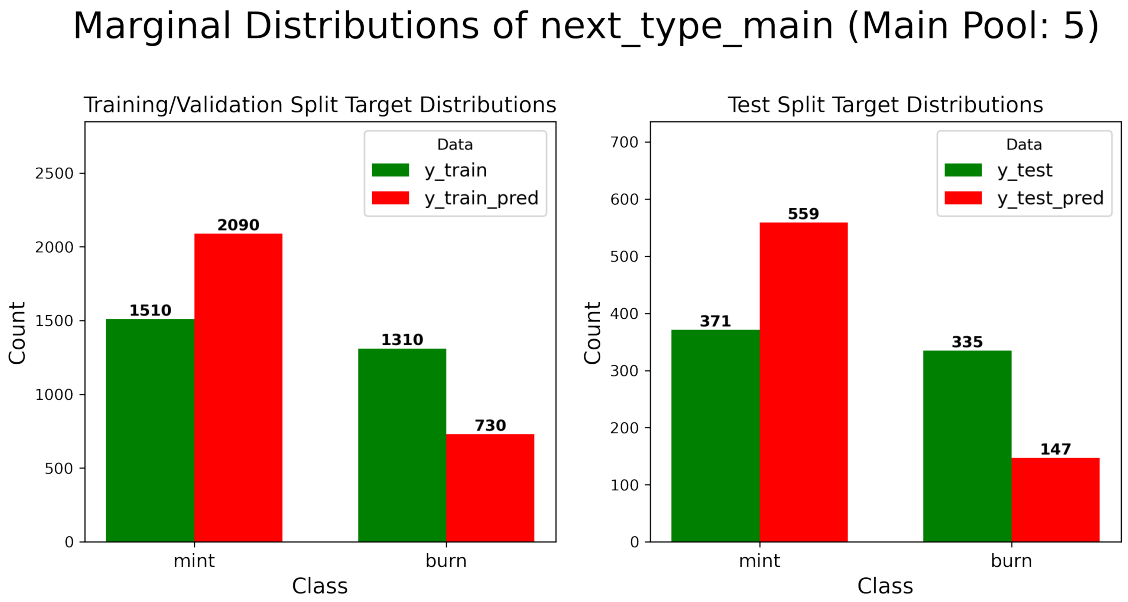



Figure 6.1: Actual and predicted distributions of target *next_type_main*, where the main pool is USDC-WETH-0.0005 and predictions are made with an optimized `RidgeClassifier` estimator.

We compare the marginal distribution of our `RidgeClassifier` predictions with the true target's distribution for the training/validation and test splits in Figure 6.1. For each split, the estimator over-predicts the total amount of mints by similar proportions. However, this doesn't inform how well it performs at individual predictions. To explore this, Figure 6.2 plots unnormalized and normalized confusion matrices for our estimator's test split predictions. We observe that it recalls 81% of incoming mint operations, but performs much worse at recalling burns (23% of the time). This is consistent with the over-prediction of mints in Figure 6.1. Further, we list key classification metrics for our `RidgeClassifier` in Table 6.2. While the difference in class recall is highlighted, we notice closer performance in precision, though both measures are nearly as bad as randomly guessing. The F1-score balances rewarding making accurate predictions of a type (precision) and capturing all instances of that type (recall), so it is significantly higher for mints. *Macro Avg* takes simple class averages of each metric, while *Weighted Avg* weights averages by class

support, yielding slightly higher metrics as mint is the majority class. Moreover, we plot the test split ROC curve in Figure 6.3a and measure AUC for predicting mints with our `RidgeClassifier`. Our estimator's AUC is slightly superior to the AUC of randomly guessing, as the TPR-FPR ratio is higher over a wide range of discrimination thresholds.

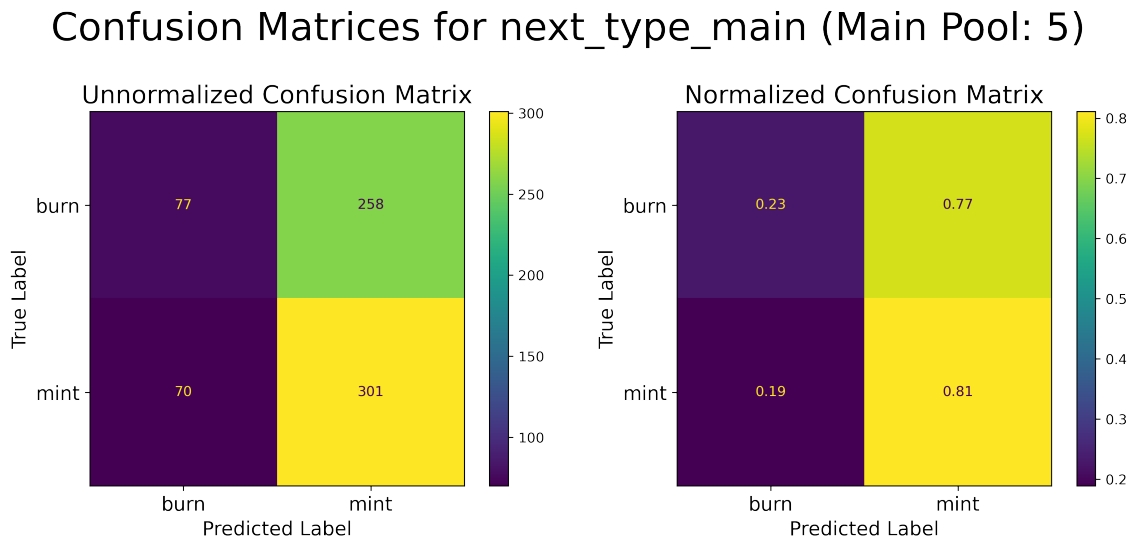
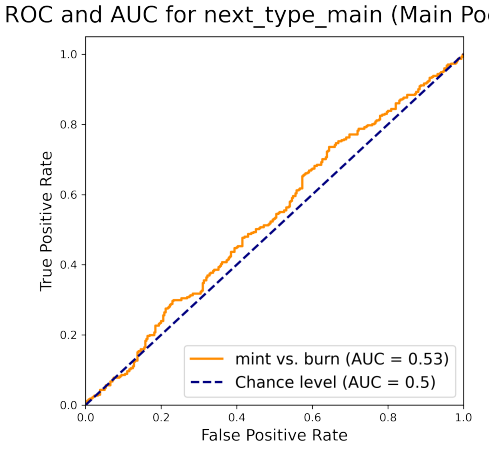



Figure 6.2: Unnormalized and normalized confusion matrices for target *next_type_main*, where the main pool is USDC-WETH-0.0005 and predictions are made on the test split with an optimized `RidgeClassifier` estimator.

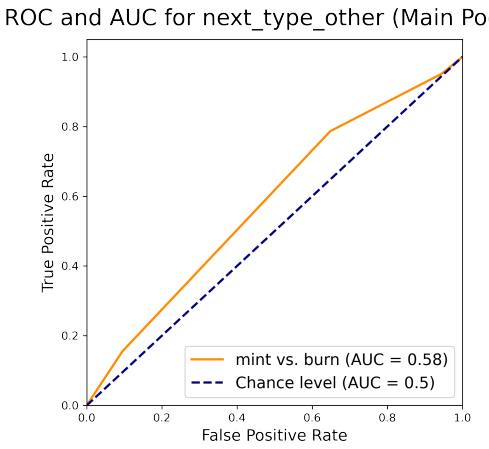
Table 6.2: Classification metrics report for target *next_type_main*, where the main pool is USDC-WETH-0.0005 and predictions are made on the test split with an optimized `RidgeClassifier` estimator.

| | **Precision** | **Recall** | **F1-Score** | **Support** |
|---|---|---|---|---|
| burn | 0.5238 | 0.2299 | 0.3195 | 335 |
| mint | 0.5385 | 0.8113 | 0.6473 | 371 |
| Accuracy | | | 0.5354 | 706 |
| Macro Avg | 0.5311 | 0.5206 | 0.4834 | 706 |
| Weighted Avg | 0.5315 | 0.5354 | 0.4918 | 706 |

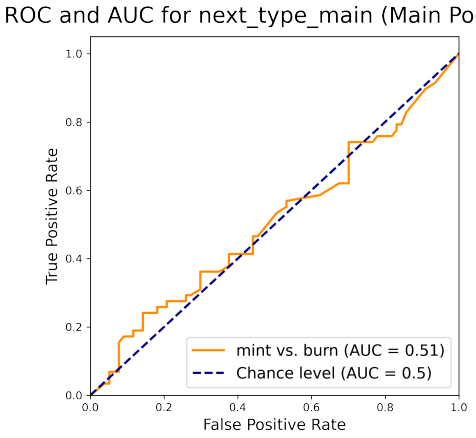
Finally, we plot the top 10 feature permutation importances for the `RidgeClassifier` on the test split in Figure 6.6a. The current event's type (*type_mint*) has relatively high predictive power for the next event type, hinting at some persistence. What is more interesting is the predictive power of *b2_burn_other* and *b1_burn_other*, which measure the distance to past burns on Pool 30, providing some evidence for spillover effects. However, no feature greatly affects the classifier's accuracy individually, which highlights its weakness.



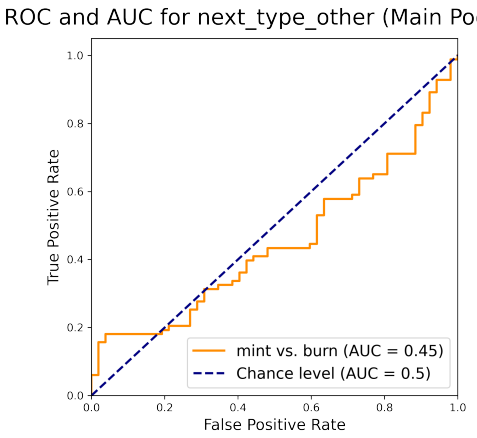

(a) Estimator: `RidgeClassifier`.




(b) Estimator: `DecisionTreeClassifier`.




(c) Estimator: `LGBMClassifier`.




(d) Estimator: `KNeighborsClassifier`.

Figure 6.3: ROC and corresponding AUC measure for predicting the mint class of *next_type_main* and *next_type_other* on the test split with the corresponding best estimator, where the main pool is USDC-WETH-0.0005 or USDC-WETH-0.003.

### 6.1.2 Target: *next_type_other*

Table B.2 summarizes classifier performance in predicting *next_type_other* from Pool 5. Surprisingly, a `DecisionTreeClassifier` of depth 2 provides the best test accuracy (0.5567), beating the persistence baseline of 0.4802 (Table 6.1). Overall, classifiers tend to perform better than for the previous target, but accuracy is still practically low. Gradient boosting methods perform as well as any non-tree-based method, but they all once again always predict the same class (burn). Figure 6.4 shows that the `DecisionTreeClassifier` stands out compared to the gradient boosting methods because it also generates mint predictions. Nevertheless, it also reveals this estimator highly over-predicts burns on both data splits.

Consistently with the above, from Figure 6.5 we see our `DecisionTreeClassifier` has a 90% recall for burns at the expense of a 16% recall for mints on the test split. As a

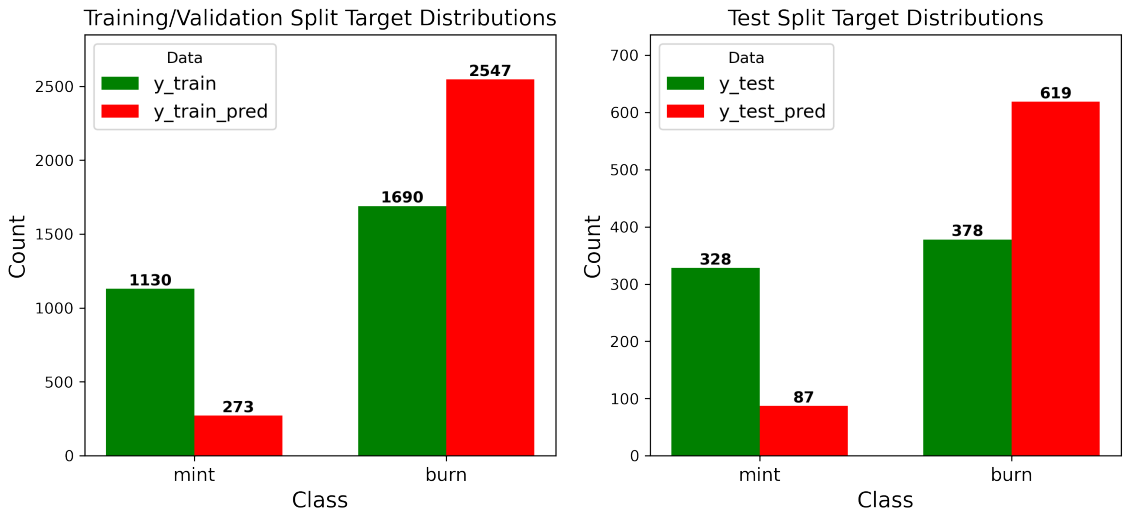



Figure 6.4: Actual and predicted distributions of target *next_type_other*, where the main pool is USDC-WETH-0.0005 and predictions are made with an optimized `DecisionTreeClassifier` estimator.

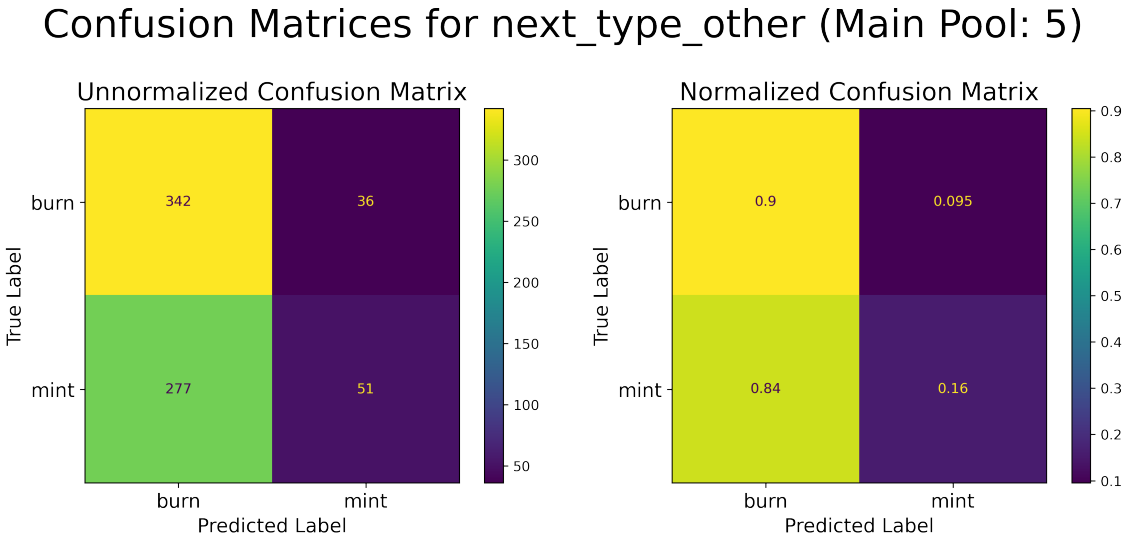


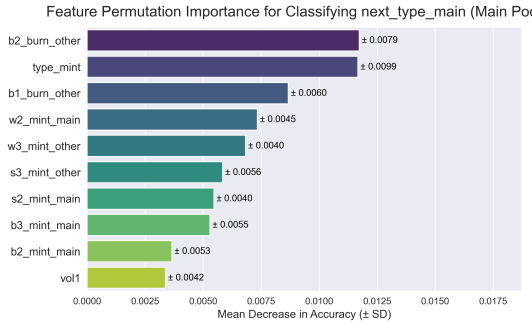

Figure 6.5: Unnormalized and normalized confusion matrices for target *next_type_other*, where the main pool is USDC-WETH-0.0005 and predictions are made on the test split with an optimized `DecisionTreeClassifier` estimator.

result of the over-prediction of burns, precision is similar for the two classes, as seen in Table 6.3. Moreover, the ROC curve for mint predictions in Figure 6.3b consistently shows superior performance compared to chance, as false mint predictions are rare in this setting. Finally, for our `DecisionTreeClassifier` of depth 2, only 2 features matter (Figure 6.6b). These measure the USD size of the third last mint (*s3_mint_other*) and the width of the second last mint (*w2_mint_other*) on Pool 30, considering no information about burns. Feature importances are higher than in Figure 6.6a, but this is an artifact of

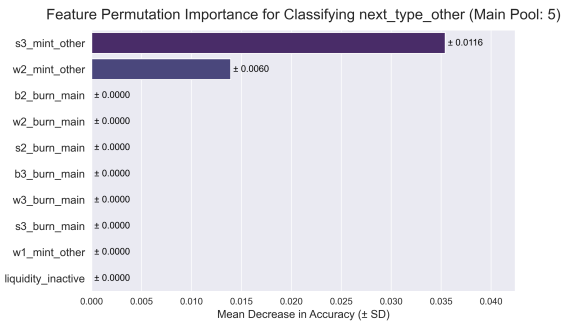
Table 6.3: Classification metrics report for target *next_type_other*, where the main pool is USDC-WETH-0.0005 and predictions are made on the test split with an optimized `DecisionTreeClassifier` estimator.

| | **Precision** | **Recall** | **F1-Score** | **Support** |
|---|---|---|---|---|
| burn | 0.5525 | 0.9048 | 0.6861 | 378 |
| mint | 0.5862 | 0.1555 | 0.2458 | 328 |
| Accuracy | | | 0.5567 | 706 |
| Macro Avg | 0.5694 | 0.5301 | 0.4659 | 706 |
| Weighted Avg | 0.5682 | 0.5567 | 0.4815 | 706 |

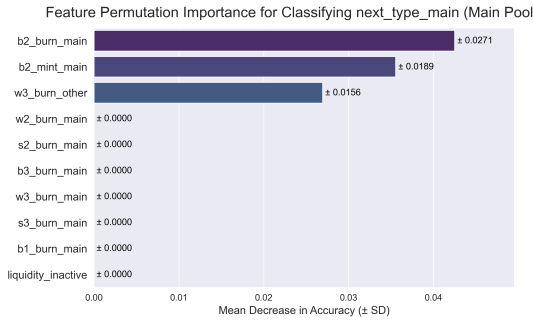
there only being two features. This is a weak classifier which doesn't split according to intuitive features.



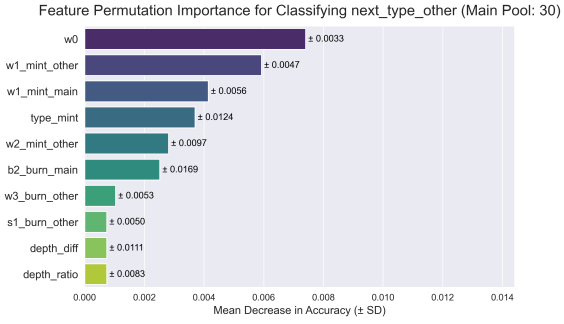

(a) Estimator: `RidgeClassifier`.




(b) Estimator: `DecisionTreeClassifier`.




(c) Estimator: `LGBMClassifier`.




(d) Estimator: `KNeighborsClassifier`.

Figure 6.6: Mean permutation importance (with standard deviation) over 50 iterations for the 10 most important features in classifying *next_type_main* and *next_type_other* on the test split with the corresponding best estimator, where the main pool is USDC-WETH-0.0005 or USDC-WETH-0.003.

## 6.2 Main Pool: USDC-WETH-0.003 (Pool 30)

In this section, we predict the next type of liquidity event in both pools from liquidity events in Pool 30.

Table 6.4: Test split accuracy of the best estimator and persistence baseline model for each classification target where the main pool is USDC-WETH-0.003.

| Target | Best Estimator | Best Acc. | Persistence Acc. |
|---|---|---|---|
| *next_type_main* | `LGBMClassifier` | 0.5926 | 0.5407 |
| *next_type_other* | `KNeighborsClassifier` | 0.4889 | 0.5259 |

### 6.2.1 Target: *next_type_main*

Table B.3 summarizes classifier performance in predicting *next_type_main* where the main pool is Pool 30. An `LGBMClassifier` has the greatest generalization accuracy (0.5926), which exceeds the persistence baseline of 0.5407 (Table 6.4), though the lower end of its 95% Wald C.I. (0.5097) doesn't. This estimator has the longest hyperparameter search time even though it leverages GOSS [69], likely due to our search over hyperparameter `bin_size`. Figure 6.7 reveals the `LGBMClassifier` suffers from the same problem as our previous `DecisionTreeClassifier`, highly over-predicting burns for both data splits. Hence, we notice a familiarly poor recall of mints in Figure 6.8 alongside a slightly better recall of burns, explaining this estimator's higher test accuracy for essentially the same target. The classification report in Table 6.5 paints a similar picture, where the `LGBMClassifier` makes similarly precise predictions for both classes and outperforms the `DecisionTreeClassifier` on both mint and burn precision. Nevertheless, we notice an AUC performance for predicting mints similar to random classification in Figure 6.3c. This may be influenced by the combination of a small test split (135 observations) and rare mint predictions.

Table 6.5: Classification metrics report for target *next_type_main*, where the main pool is USDC-WETH-0.003 and predictions are made on the test split with an optimized `LGBMClassifier` estimator.

| | Precision | Recall | F1-Score | Support |
|---|---|---|---|---|
| burn | 0.5917 | 0.9221 | 0.7208 | 77 |
| mint | 0.6000 | 0.1552 | 0.2466 | 58 |
| Accuracy | | | 0.5926 | 135 |
| Macro Avg | 0.5958 | 0.5386 | 0.4837 | 135 |
| Weighted Avg | 0.5952 | 0.5926 | 0.5171 | 135 |

Strikingly, only 3 features affect the predictions of our `LGBMClassifier` according to Figure 6.6c, even though it is an ensemble of 158 decision trees. This speaks to the difficulty of the classification task, as this estimator's hyperparameter search doesn't force it to over-regularize. The important features measure the distance to the second last burn

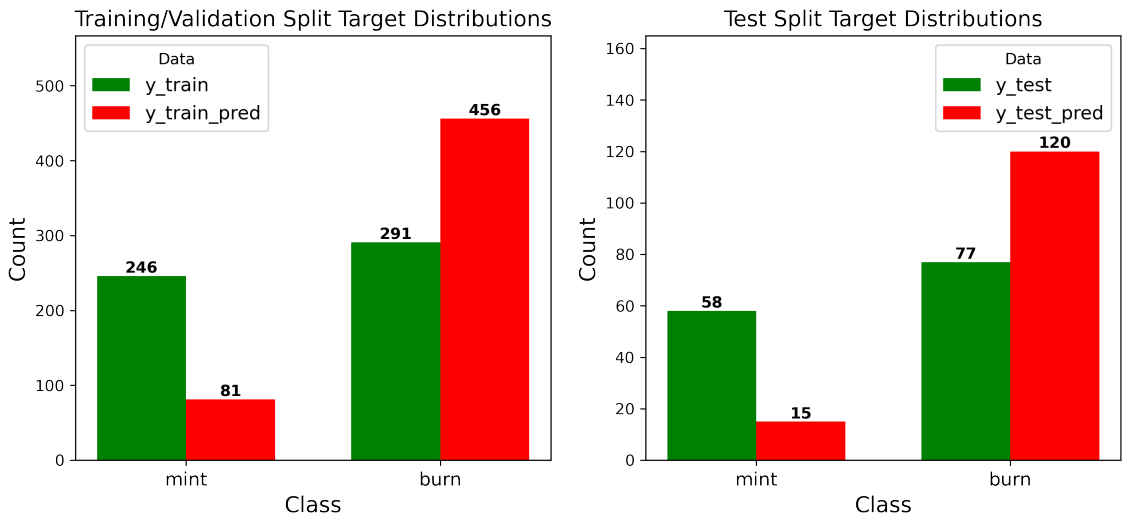



Figure 6.7: Actual and predicted distributions of target *next_type_main*, where the main pool is USDC-WETH-0.003 and predictions are made with an optimized `LGBMClassifier` estimator.

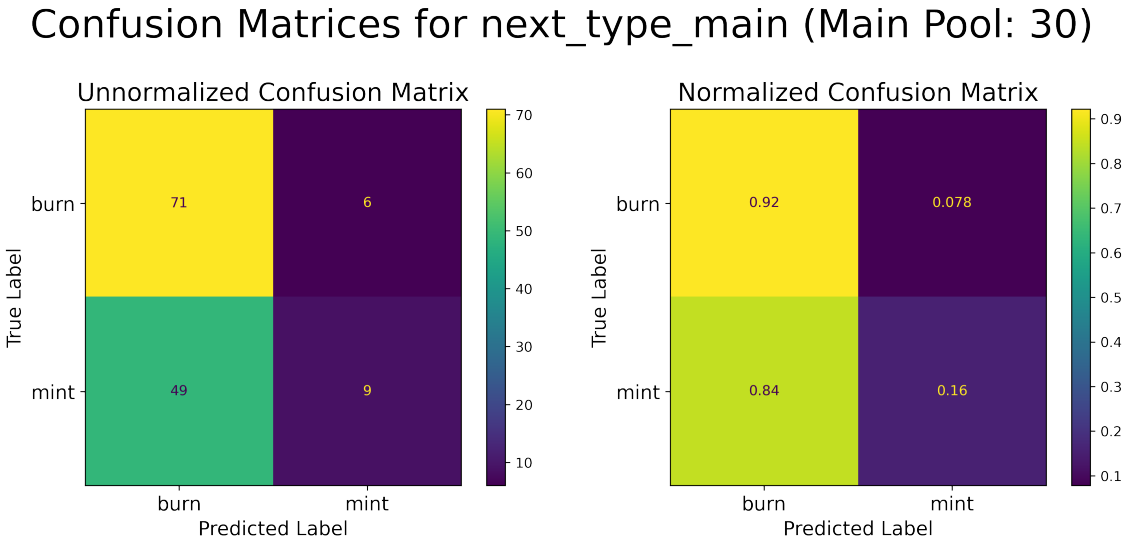



Figure 6.8: Unnormalized and normalized confusion matrices for target *next_type_main*, where the main pool is USDC-WETH-0.003 and predictions are made on the test split with an optimized `LGBMClassifier` estimator.

(*b2_burn_main*) and mint (*b2_mint_main*) on Pool 30 and the width of the third last burn on Pool 5 (*w3_burn_other*). This is some more evidence of spillover effects, but the Pool 5 feature is the lowest contributor to accuracy.

### 6.2.2 Target: *next_type_other*

Finally, Table B.4 displays classifier performance in predicting *next_type_other* from events in Pool 30. This is the only classification target where no optimized classifier has better test accuracy than guessing. The best estimator is a `KNeighborsClassifier` with test accuracy 0.4889, lagging behind the baseline of 0.5259 (Table 6.4). This estimator has the fastest hyperparameter search among non-tree-based methods and opts for a `StandardScaler`, as do the majority of non-tree-based methods for this dataset. The estimator's reconstruction of target distributions in both splits is impressive (Figure 6.9), and we notice from Table B.4 it actually predicts the training/validation split perfectly. It uses 24 `n_neighbors`, which is high relative to the test split and cross-validation folds, but yields the best time series cross-validation performance.

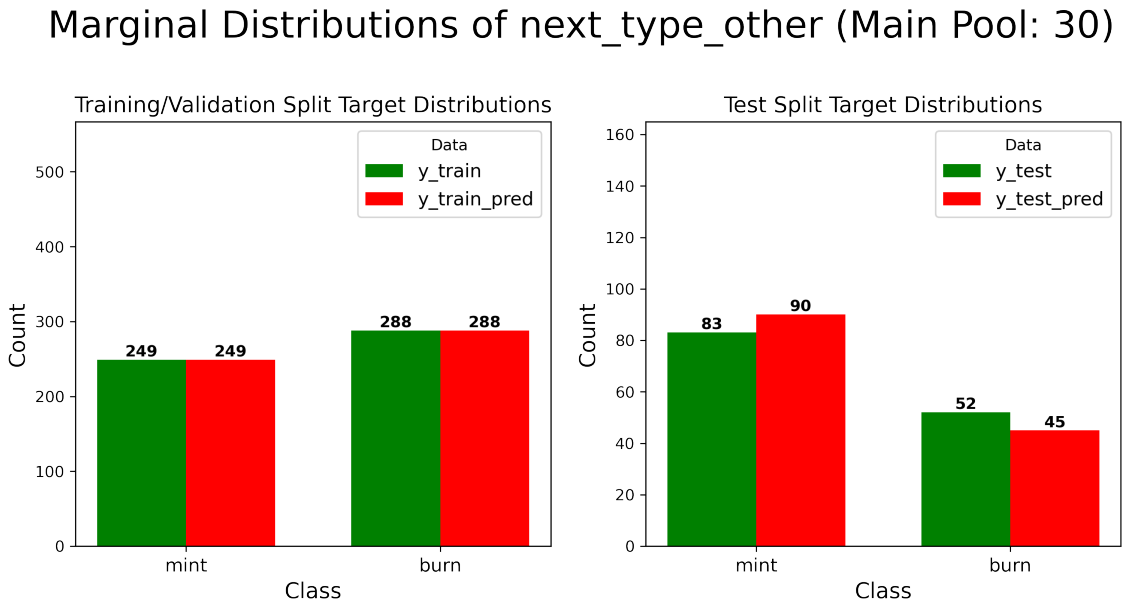



Figure 6.9: Actual and predicted distributions of target *next_type_other*, where the main pool is USDC-WETH-0.003 and predictions are made with an optimized `KNeighborsClassifier` estimator.

Nevertheless, Figure 6.10 shows that on the test split, the `KNeighborsClassifier` recalls mints at a mediocre 63% rate and burns at a poor 27% rate. Unlike the other best estimators, this estimator doesn't capture either class particularly well. Moreover, we notice uniquely poor precision for burns as well in Table 6.6, helping explain this estimator's poor performance. Figure 6.3d completes the picture, highlighting we would be better off randomly classifying mints instead of using the estimator on the test split. Finally, Figure 6.6d shows that the most important features for test accuracy are event width-related, which is not intuitive. Since more intuitive features are missing, it is plausible that the `KNeighborsClassifier` learns rules about them on the training/validation split which don't apply on the test split owing to dataset shift (discussed in Section 6.3), so these features don't have high permutation importance. We clearly prefer the `RidgeClassifier` from Section 6.1.1 to classify the next event on Pool 5.

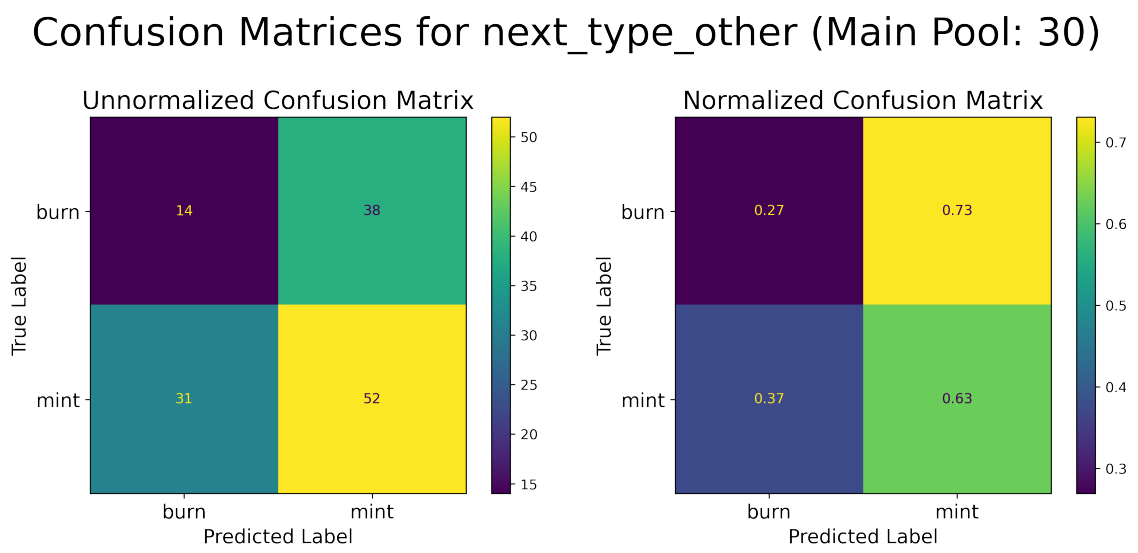



Figure 6.10: Unnormalized and normalized confusion matrices for target *next_type_other*, where the main pool is USDC-WETH-0.003 and predictions are made on the test split with an optimized `KNeighborsClassifier` estimator.

Table 6.6: Classification metrics report for target *next_type_other*, where the main pool is USDC-WETH-0.003 and predictions are made with an optimized `KNeighborsClassifier` estimator.

| | **Precision** | **Recall** | **F1-Score** | **Support** |
|---|---|---|---|---|
| burn | 0.3111 | 0.2692 | 0.2887 | 52 |
| mint | 0.5778 | 0.6265 | 0.6012 | 83 |
| Accuracy | | | 0.4889 | 135 |
| Macro Avg | 0.4444 | 0.4479 | 0.4449 | 135 |
| Weighted Avg | 0.4751 | 0.4889 | 0.4808 | 135 |

## 6.3 Discussion

We identified estimators that exceed the test split accuracy of the baseline persistence model for all classification targets except for *next_type_other* in the Pool 30 dataset. Nevertheless, the test accuracy of our best estimators is practically low. Their `TimeSeriesSplit` cross-validation accuracy is similarly low, but we notice in the tables of Appendix B that the training/validation accuracy of some classifiers is near 1. For example, we see this for our `KNeighborsClassifier` discussed in Section 6.2.2. The hyperparameter space of each method is built to avoid overfitting, so the combination of high training/validation accuracy and low test accuracy for *tuned* estimators leads us to suspect **dataset shift**, where the joint feature-target distribution differs between the training/validation and test split. This may be an empirical phenomenon due to our limited sample size, or a general consequence of operating in a non-stationary environment [81] (which cryptocurrency trading liquidity likely is).

We explore dataset shift by calculating the KL-divergence from KDEs of training/validation

Table 6.7: Kullback-Leibler divergence from the training/validation distribution of selected features to their test distribution for each dataset.

| **Feature** | **USDC-WETH-0.0005 Dataset** | **USDC-WETH-0.003 Dataset** |
|---|---|---|
| *s0* | 2.404 | inf |
| *s1_mint_main* | inf | inf |
| *s1_burn_main* | 4.232 | inf |
| *b1_burn_main* | 0.1532 | inf |
| *w1_burn_other* | inf | 0.1349 |
| *depth_diff* | 3.1891 | 4.5702 |
| *price_diff* | inf | inf |

feature distributions to KDEs of test feature distributions. Results for both datasets are in Table 6.7, where we notice KL-divergences of $inf$ for many features, meaning their training/validation distribution places probability mass somewhere their test distribution doesn't. This could help explain our classifiers' poor test performance. Flexible models like `KNeighborsClassifier` and `RandomForestClassifier` learn complex rules about features in the first 80% training/validation split, but applying those rules to the final 20% test split may perform poorly because feature distributions are different. Under dataset shift, our generalization accuracy estimates are negatively biased. Simultaneously, feature importance plots showed no feature drastically affects an estimator's test accuracy, so it is plausible we require a more informative feature space for better classification performance.

# Chapter 7

# Regression Results & Model Evaluation

In this chapter, we analyze the results of the supervised learning pipeline described in Chapter 5 for our 8 regression targets. For each target, we select the estimator with the best test split mean squared log error (MSLE) and evaluate its predictions compared to the true target, then carry out a feature importance analysis. Like in Chapter 6, the baseline we compare our predictions against is the naive persistence model predicting $\hat{y}_i = y_{i-1}$. Detailed supervised learning performance tables for each target are in Appendix C, taking the same structure as the Appendix B tables we discussed in the previous chapter.

## 7.1 Main Pool: USDC-WETH-0.0005 (Pool 5)

In this section, we forecast the arrival time of the next mint and burn event in both pools from liquidity events in Pool 5.

Table 7.1: Test split MSLE of the best estimator and persistence baseline model for each regression target where the main pool is USDC-WETH-0.0005.

| Target | Best Estimator | Best MSLE | Baseline MSLE |
|---|---|---|---|
| *next_mint_time_main* | `KernelRidge` | 1.9379 | 3.4051 |
| *next_burn_time_main* | `KNeighborsRegressor` | 2.1615 | 3.9844 |
| *next_mint_time_other* | `KNeighborsRegressor` | 1.9772 | 3.3366 |
| *next_burn_time_other* | `XGBRegressor` | 1.8832 | 3.8002 |

### 7.1.1 Target: *next_mint_time_main*

Table C.1 summarizes the performance of each regressor described in Section 5.2.2 for predicting target *next_mint_time_main* from the Pool 5 dataset. The best performing estimator is a `KernelRidge` with an RBF kernel, achieving a test split MSLE of 1.9379, which comfortably beats the persistence baseline of 3.4051 Table 7.1. Given MSLE measures relative error, a drop of 1.5 is a real improvement. Our `KernelRidge` uses a `PowerTransformer`, as the vast majority of non-tree-based regressors do for the Pool 5 dataset. It features one of the longer hyperparameter searches, while a much faster

`XGBRegressor` achieves a marginally higher test MSLE. Nevertheless, we analyze the `KernelRidge` further.

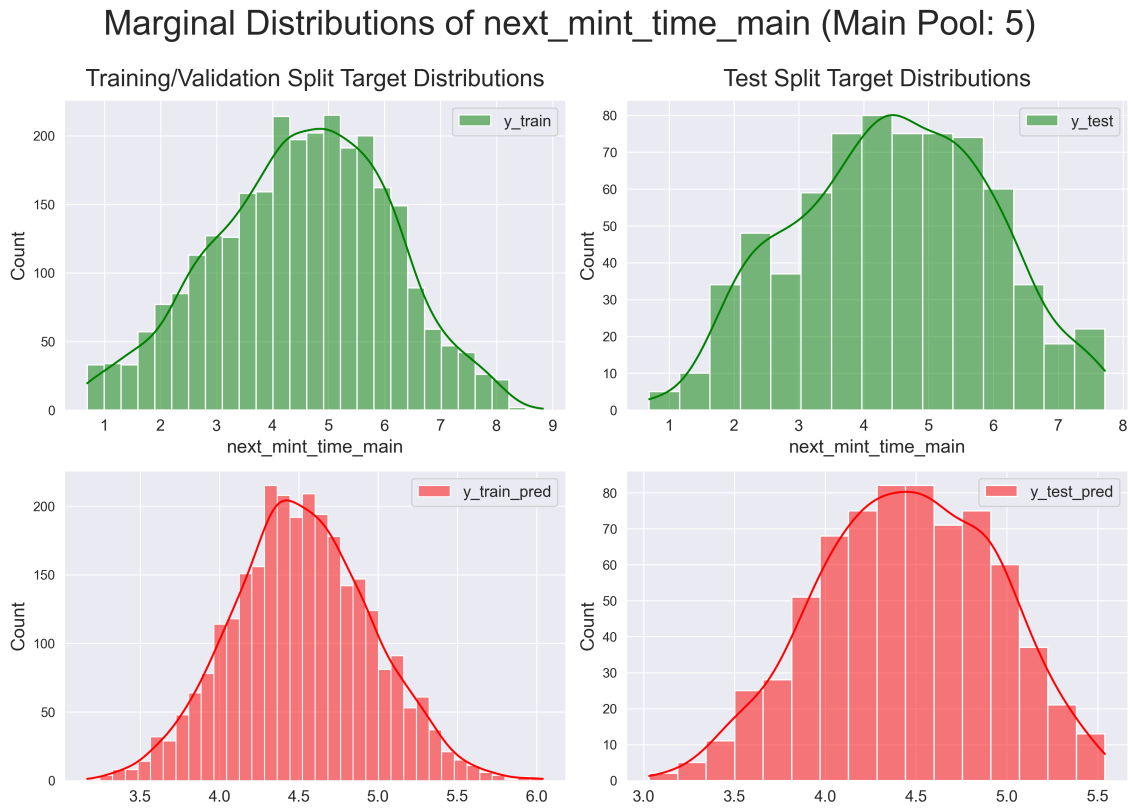



Figure 7.1: Actual and predicted distributions of target *next_mint_time_main*, where the main pool is USDC-WETH-0.0005 and predictions are made with an optimized `KernelRidge` estimator.

Table 7.2: Training/validation split and test split Kullback-Leibler divergence from the target distribution to its best estimator's prediction distribution, for each regression target where the main pool is USDC-WETH-0.0005.

| Target | Best Estimator | Training/Validation KL | Test KL |
|---|---|---|---|
| **next_mint_time_main** | `KernelRidge` | 0.0691 | 0.0262 |
| **next_burn_time_main** | `KNeighborsRegressor` | 0.0 | 0.0436 |
| **next_mint_time_other** | `KNeighborsRegressor` | 1.8801 | 0.8848 |
| **next_burn_time_other** | `XGBRegressor` | 2.2342 | 1.6492 |

In Figure 7.1, we visualize the marginal distributions of the (log1p-transformed) predicted and actual target for the training/validation and test split. Our `KernelRidge` impressively reconstructs the shape of *next_mint_time_main* for each split, but in both cases predicts only an inner subset of the values the target can take on. For continuous distributions it is harder to visually gauge how well estimators replicate the target distribution, so we calculate the KL-divergence from a KDE of the true target distribution to a KDE of the prediction distribution for both splits in Table 7.2. We find a lower KL-divergence for the

test split. Moreover, Figure 7.2 shows residuals plotted against predictions scatter around 0 with a homoskedastic pattern, suggesting no obvious model misspecification that we could correct to improve our model's predictions. Nevertheless, the left side of the plot highlights that the `KernelRidge` matches the direction of *next_mint_time_main* but fails to identify its tail values.

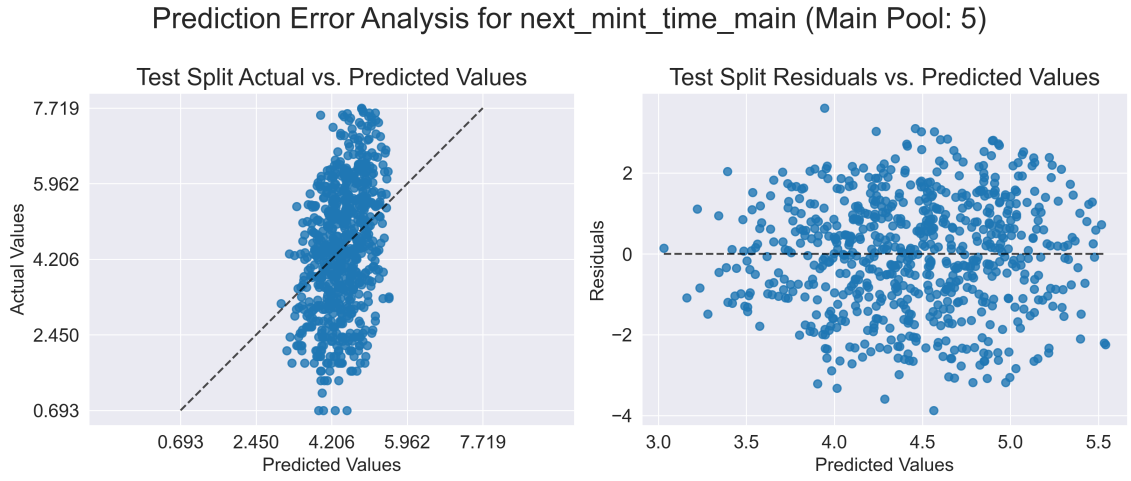



Figure 7.2: Plots of target vs. predicted values and residuals vs. predicted values for regression target *next_mint_time_main*. The main pool is USDC-WETH-0.0005 and predictions are made on the test split with an optimized `KernelRidge` estimator.

Finally, Figure 7.9a highlights features which are intuitively important for predicting target *next_mint_time_main*, as they track recent liquidity trends on Pool 5. Interestingly, the distance to previous burn events on Pool 5 appears more important than the distance to previous mint events, and spillover effects from Pool 30 appear weak. No single feature has a dominant impact on MSLE, but *type_mint* stands out relatively.

### 7.1.2 Target: *next_burn_time_main*

The regressor performance summary for predicting *next_burn_time_main* on Pool 5 is in Table C.2. Test split MSLEs are uniformly higher than for the previous target, and the estimator achieving the lowest is a `KNeighborsRegressor` with an MSLE of 2.1615, almost half of the baseline MSLE of 3.9844 (Table 7.1). It uses a `PowerTransformer` and a high `n_neighbors` of 88 and its hyperparameter search is as fast as any non-tree-based method's. As Figure 7.3 shows, it perfectly reconstructs the training/validation target distribution (with a training/validation MSLE of 0), but once again its test split predictions don't reach the target's tails. The test KL-divergence in Table 7.2 is relatively low compared to future targets.

Figure 7.4 shows a residual plot that similarly doesn't hint at model misspecification, while the left plot highlights the restricted range of predictions the `KNeighborsRegressor` makes (though in the right direction). Figure 7.9b once again emphasizes intuitively important features for predicting *next_burn_time_main*. Distances to past burns and mints on Pool 5 are most important, alongside a stronger indication of spillover effects from the

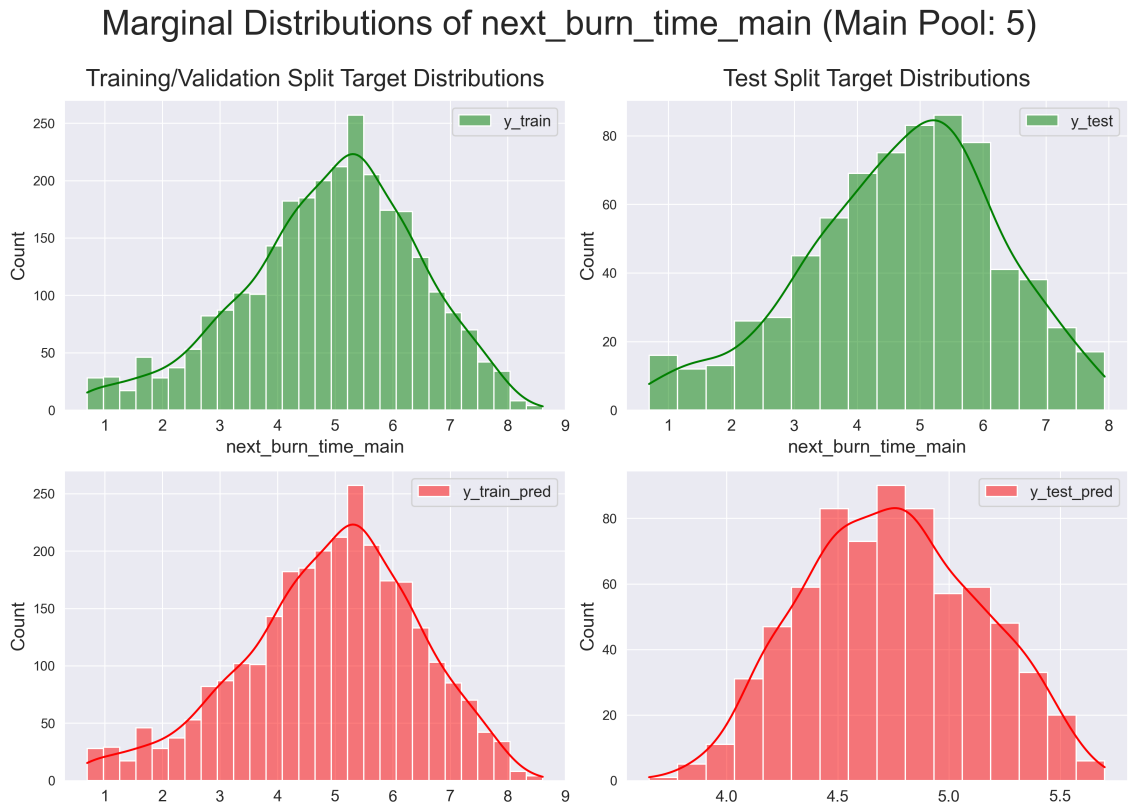



Figure 7.3: Actual and predicted distributions of target *next_burn_time_main*, where the main pool is USDC-WETH-0.0005 and predictions are made with an optimized `KNeighborsRegressor` estimator.

second highest importance of *b2_mint_other*. Nevertheless, no single feature dominates MSLE.

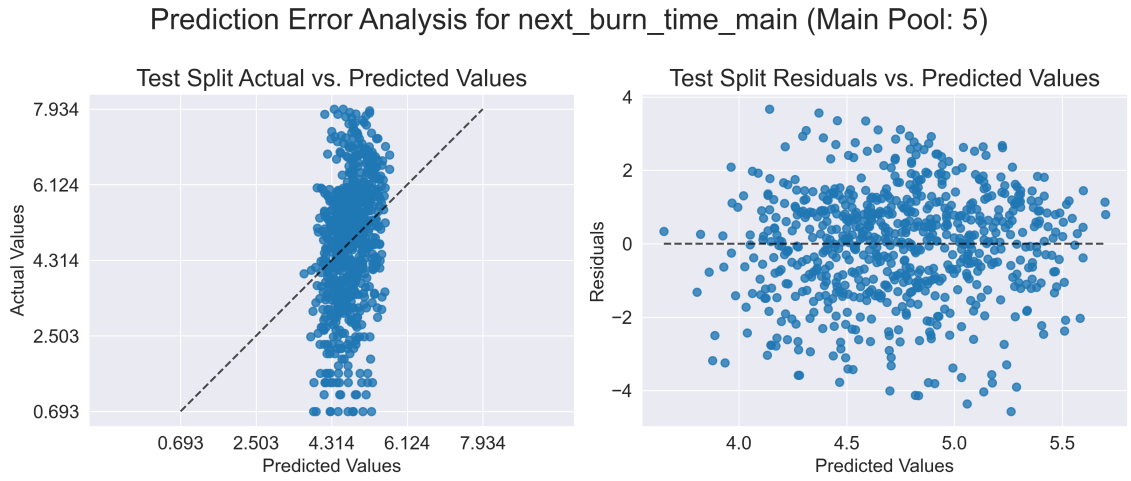



Figure 7.4: Plots of target vs. predicted values and residuals vs. predicted values for regression target *next_burn_time_main*. The main pool is USDC-WETH-0.0005 and predictions are made on the test split with an optimized `KNeighborsRegressor` estimator.

### 7.1.3 Target: *next_mint_time_other*

Table C.3 summarizes regressor performance for predicting *next_mint_time_other* on Pool 5. A `KNeighborsRegressor` once again achieves the lowest test MSLE at 1.9772, this time using a `StandardScaler` and an even higher `n_neighbors` of 98. Though its performance exceeds the baseline MSLE of 3.3366, its predictions don't perfectly reconstruct the target's training/validation split (Figure 7.5). We see our test predictions are closer to reaching the right end of the target's distribution, but ignore its long left tail, yielding the highest test KL-divergence we've seen so far in Table 7.2.

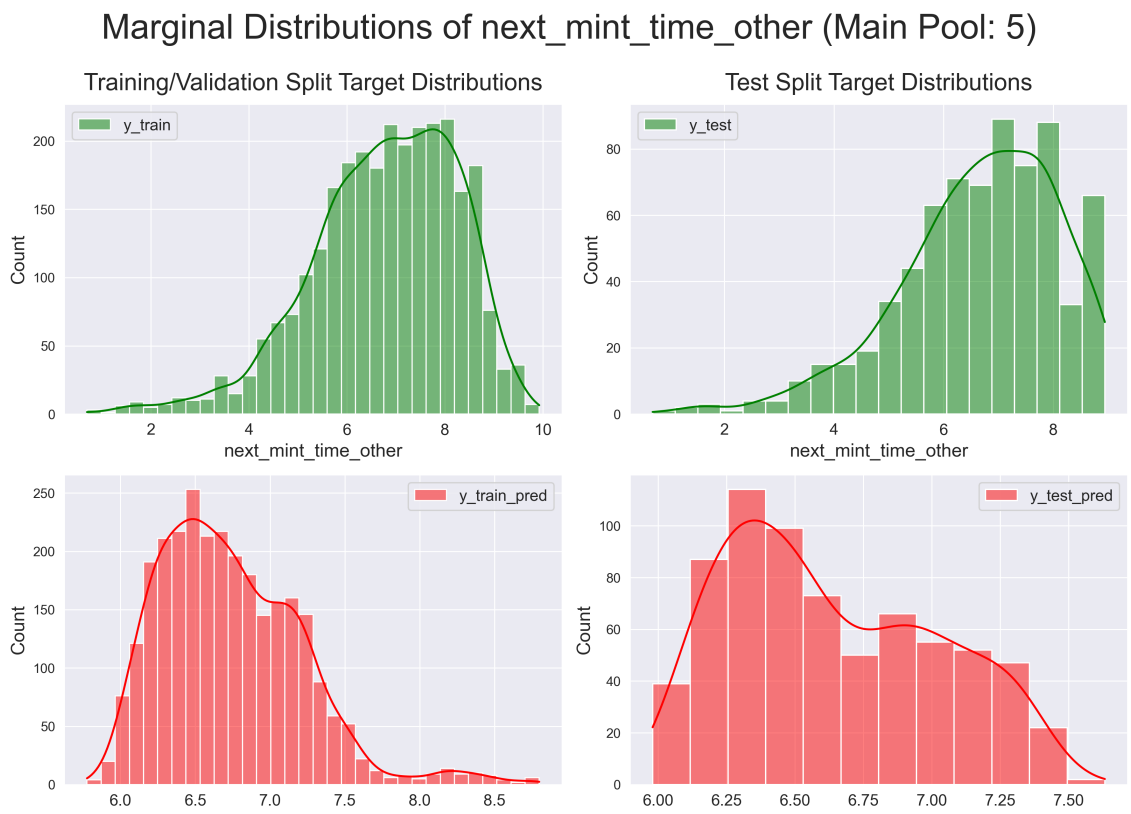



Figure 7.5: Actual and predicted distributions of target *next_mint_time_other*, where the main pool is USDC-WETH-0.0005 and predictions are made with an optimized `KNeighborsRegressor` estimator.

The prediction error plots in Figure 7.6 now show some unbalanced negative residuals, reflecting the deficiency of the `KNeighborsRegressor` in identifying tail values. If the tail of a target distribution is longer, our estimator for that target produces more extreme residuals by failing to capture it. Finally, Figure 7.9c shows intuitively important features for predicting *next_mint_time_other* in the distances of past mints and burns on Pool 30. More interestingly, the price difference between pools appears important, hinting at trading-related spillovers between pools.

### 7.1.4 Target: *next_burn_time_other*

Finally, Table C.4 summarizes regressor performance for forecasting *next_burn_time_other* from Pool 5. The best estimator is an `XGBRegressor` made up of 184 decision stumps,

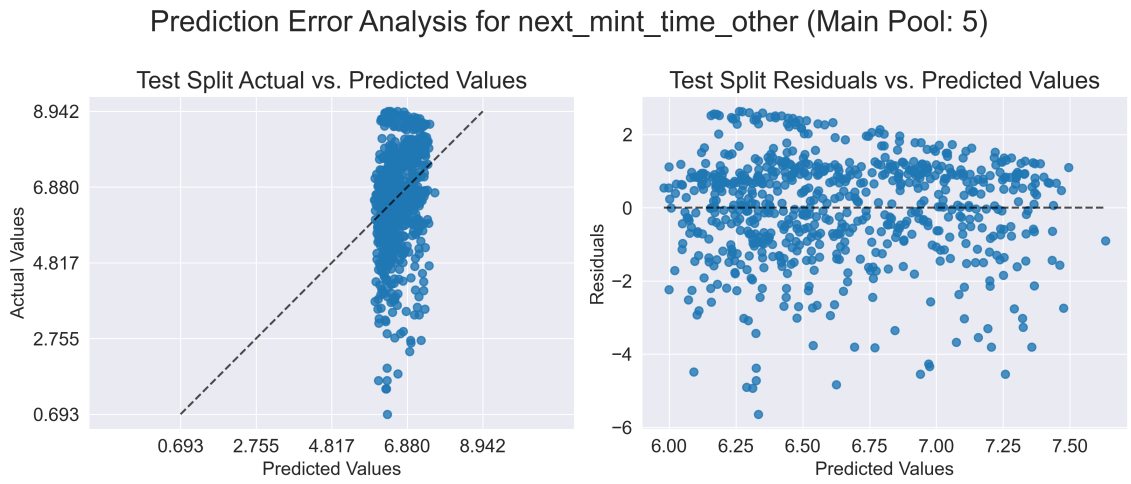



Figure 7.6: Plots of target vs. predicted values and residuals vs. predicted values for regression target *next_mint_time_other*. The main pool is USDC-WETH-0.0005 and predictions are made on the test split with an optimized `KNeighborsRegressor` estimator.

which achieves a test MSLE of 1.8832, less than half of the baseline MSLE of 3.8002 (Table 7.1). This estimator also has the fastest hyperparameter search among tree ensemble methods. Nevertheless, we notice the same issues in predictions reaching the target's long left tail in Figure 7.7 as for the previous target. The `XGBRegressor`'s train/validation and test KL-divergence in Table 7.2 are the largest for any regression target in the Pool 5 dataset.

Figure 7.8 highlights the same negative residual issues as Figure 7.6, so an ensemble tree method doesn't solve the problem we've seen with our non-tree estimators of never predicting tail target values. Finally, Figure 7.9d highlights the importance of Pool 30 lagged features for the success of our `XGBRegressor` in forecasting *next_burn_time_other*. We notice liquidity spillover effects through `b3_burn_main`, though `b3_burn_other` is at least twice more impactful for test MSLE than any other feature.

## 7.2 Main Pool: USDC-WETH-0.003 (Pool 30)

In this section, we forecast the arrival time of the next mint and burn event on both pools from liquidity events in Pool 30.

Table 7.3: Test split MSLE of the best estimator and persistence baseline model for each regression target where the main pool is USDC-WETH-0.003.

| Target | Best Estimator | Best MSLE | Baseline MSLE |
|---|---|---|---|
| *next_mint_time_main* | `XGBRegressor` | 2.2297 | 4.9481 |
| *next_burn_time_main* | `RandomForestRegressor` | 2.4751 | 5.1637 |
| *next_mint_time_other* | `Ridge` | 1.8563 | 3.5054 |
| *next_burn_time_other* | `KNeighborsRegressor` | 1.8429 | 3.3967 |

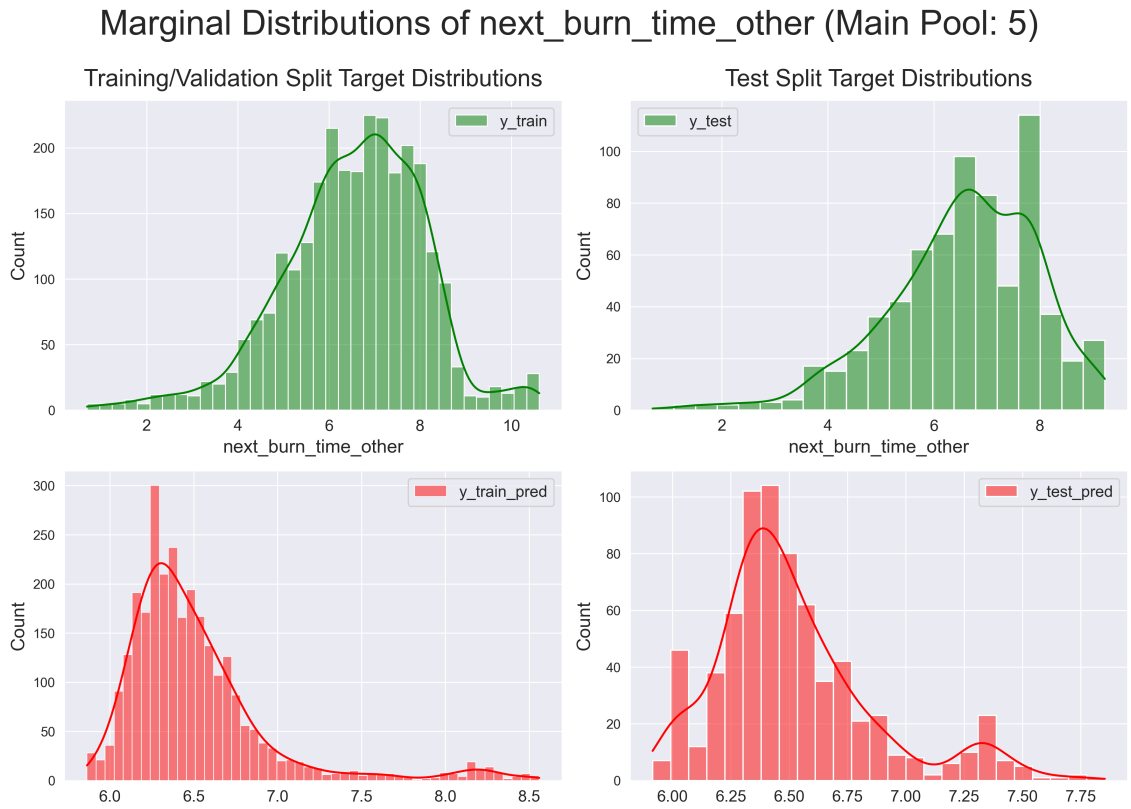



Figure 7.7: Actual and predicted distributions of target *next_burn_time_other*, where the main pool is USDC-WETH-0.0005 and predictions are made with an optimized `XGBRegressor` estimator.

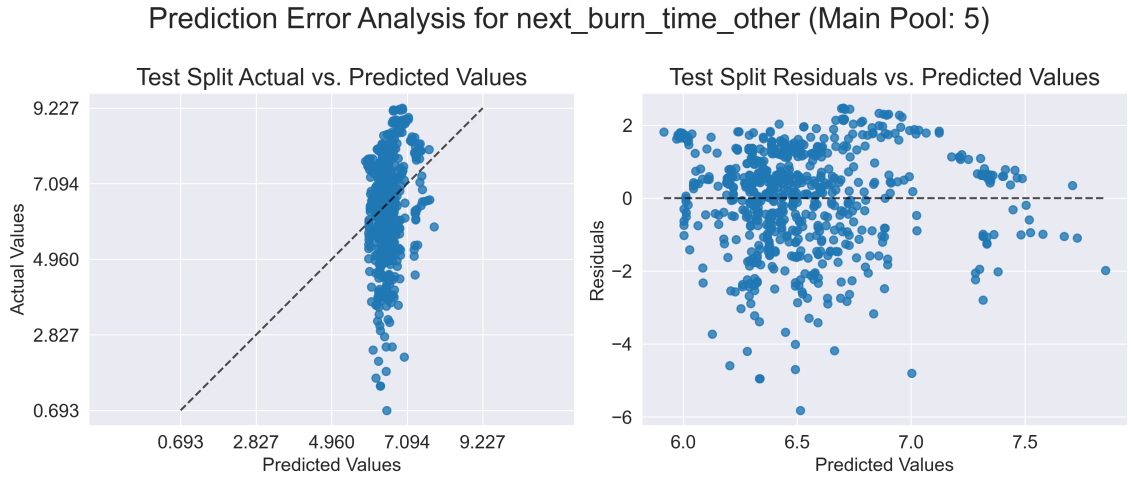
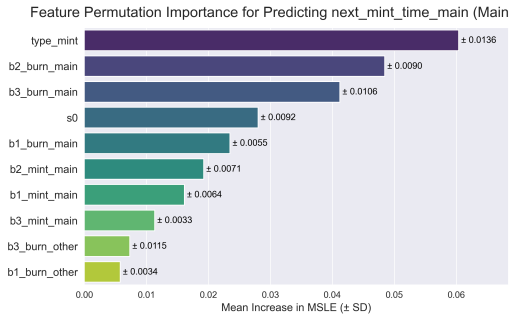
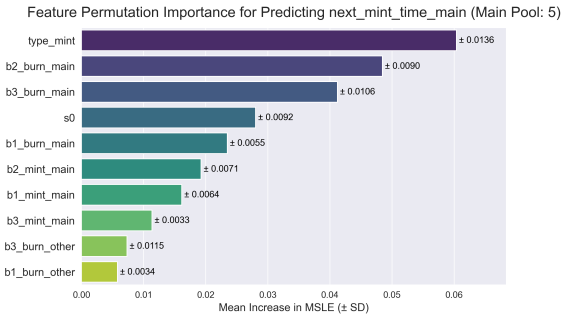


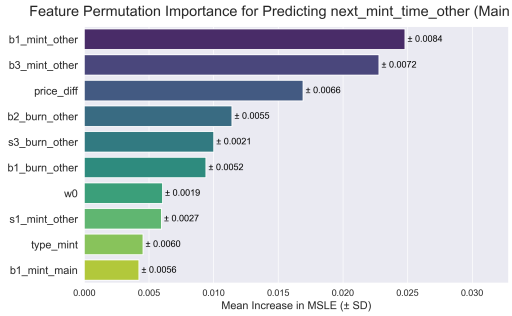

Figure 7.8: Plots of target vs. predicted values and residuals vs. predicted values for regression target *next_burn_time_other*. The main pool is USDC-WETH-0.0005 and predictions are made on the test split with an optimized `XGBRegressor` estimator.



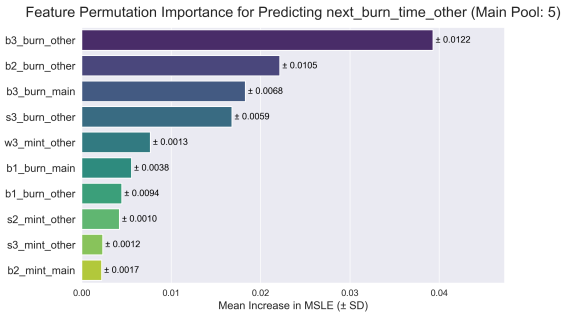

(a) Estimator: `KernelRidge`.




(b) Estimator: `KNeighborsRegressor`.




(c) Estimator: `KNeighborsRegressor`.




(d) Estimator: `XGBRegressor`.

Figure 7.9: Mean permutation importance (with standard deviation) over 50 iterations for the 10 most important features in predicting *next_mint_time_main*, *next_burn_time_main*, *next_mint_time_other*, and *next_burn_time_other* on the test split with the corresponding best estimator, where the main pool is USDC-WETH-0.0005.

### 7.2.1 Target: *next_mint_time_main*

Table C.5 summarizes regressor performance for predicting *next_mint_time_main* from Pool 30. This is the second hardest regression task as measured by the best estimator's test MSLE, which is 2.2297 for an `XGBRegressor` made up of 184 decision stumps, like our previous `XGBRegressor`. This is a notable improvement over the baseline MSLE of 4.9481 (Table 7.3). Interestingly, while this target is left-skewed like the previous two, the estimator's predictions now match its shape for both data splits, yielding lower training/validation and test KL-divergence in Table 7.4.

Table 7.4: Training/validation split and test split Kullback-Leibler divergence from the target distribution to its best estimator's prediction distribution, for each regression target where the main pool is USDC-WETH-0.003.

| Target | Best Estimator | Training/Validation KL | Test KL |
|---|---|---|---|
| *next_mint_time_main* | `XGBRegressor` | 0.2539 | 0.162 |
| *next_burn_time_main* | `RandomForestRegressor` | 0.4526 | 0.5444 |
| *next_mint_time_other* | `Ridge` | 0.264 | 0.1309 |
| *next_burn_time_other* | `KNeighborsRegressor` | 0.0 | 0.1301 |

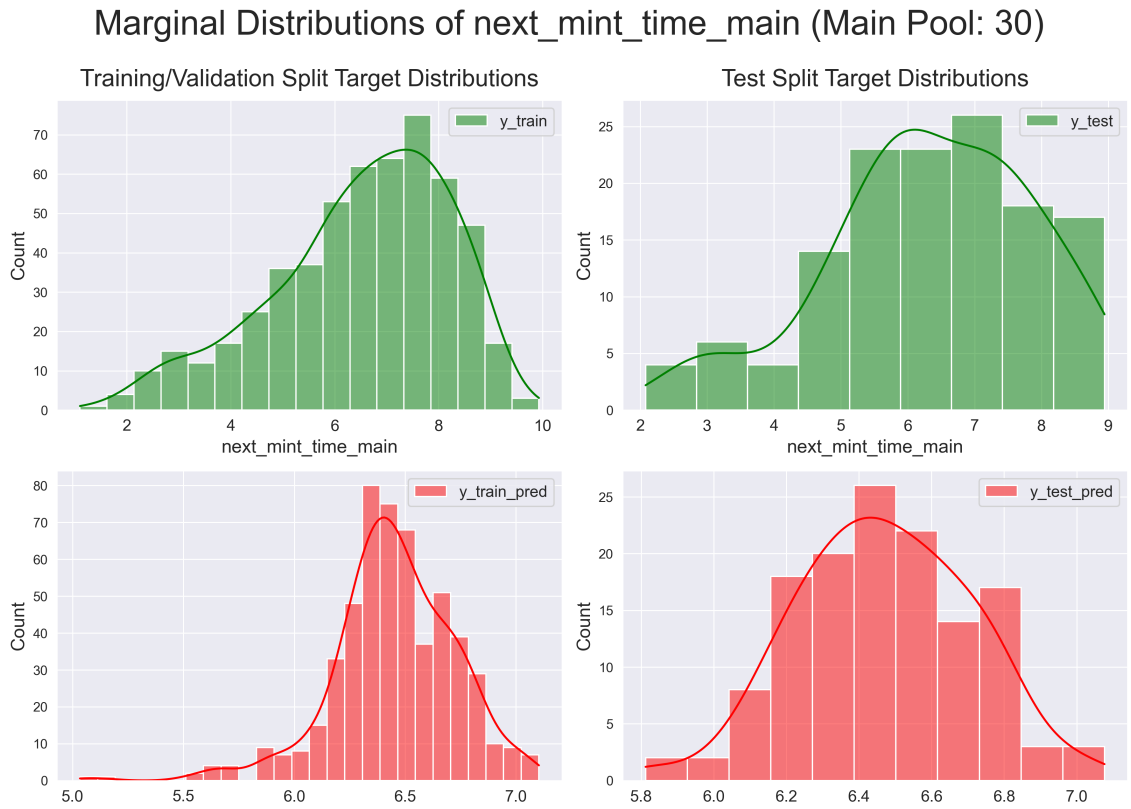



Figure 7.10: Actual and predicted distributions of target *next_mint_time_main*, where the main pool is USDC-WETH-0.003 and predictions are made with an optimized `XGBRegressor` estimator.

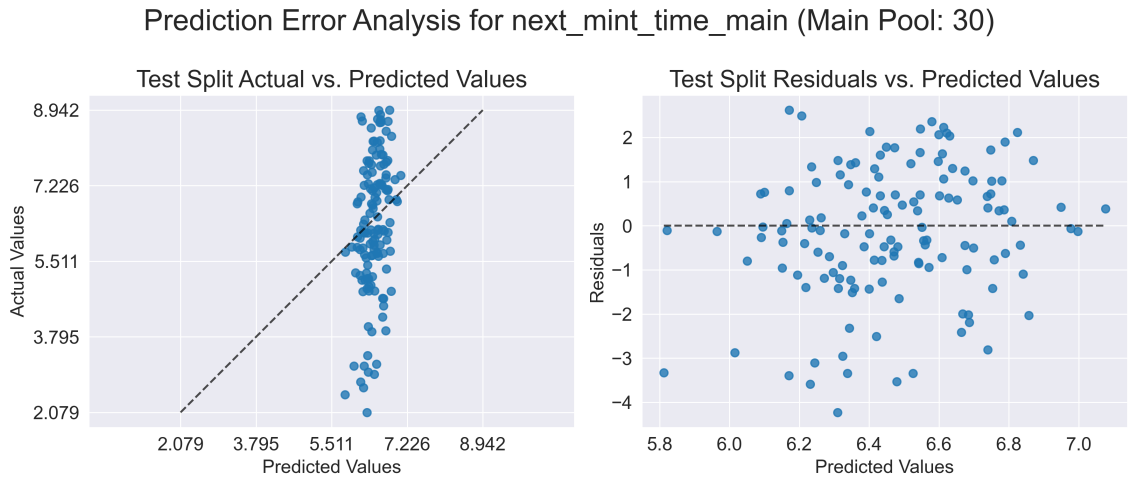



Figure 7.11: Plots of target vs. predicted values and residuals vs. predicted values for regression target *next_mint_time_main*. The main pool is USDC-WETH-0.003 and predictions are made on the test split with an optimized `XGBRegressor` estimator.

Nevertheless, we encounter the familiar problem of failing to predict tail target values. This is reflected in the residual plot of Figure 7.11, though extreme residuals here are slightly less negative than for the last two targets in the Pool 5 dataset, as the left tail of

*next_mint_time_main* in Pool 30 is less pronounced. Finally, the feature importance plot in Figure 7.18a attributes the greatest contributions to *b1_mint_other* and *s3_burn_main*, hinting at strong spillover effects from the more active pool.

### 7.2.2 Target: *next_burn_time_main*

Table C.6 summarizes regressor performance for predicting *next_burn_time_main* from Pool 30. A `RandomForestRegressor` of 248 decision trees has the best test MSLE of 2.4751, less than half of the baseline of 5.1637. This target is the hardest predict based on test MSLE, though we observe in Figure 7.12 that our estimator reconstructs its distribution particularly well for the training/validation split, including the tails. This doesn't extend to the test split, where our `RandomForestRegressor` makes predictions particularly far from the left tail of *next_burn_time_main*, yielding a familiar residual plot in Figure 7.13 where negative residuals reach more extreme values than positive residuals. Finally, Figure 7.18b highlights the importance of *s0*, which is not intuitively connected to *next_burn_time_main*. This could relate to the estimator's relatively poor performance, which we address in the final section.

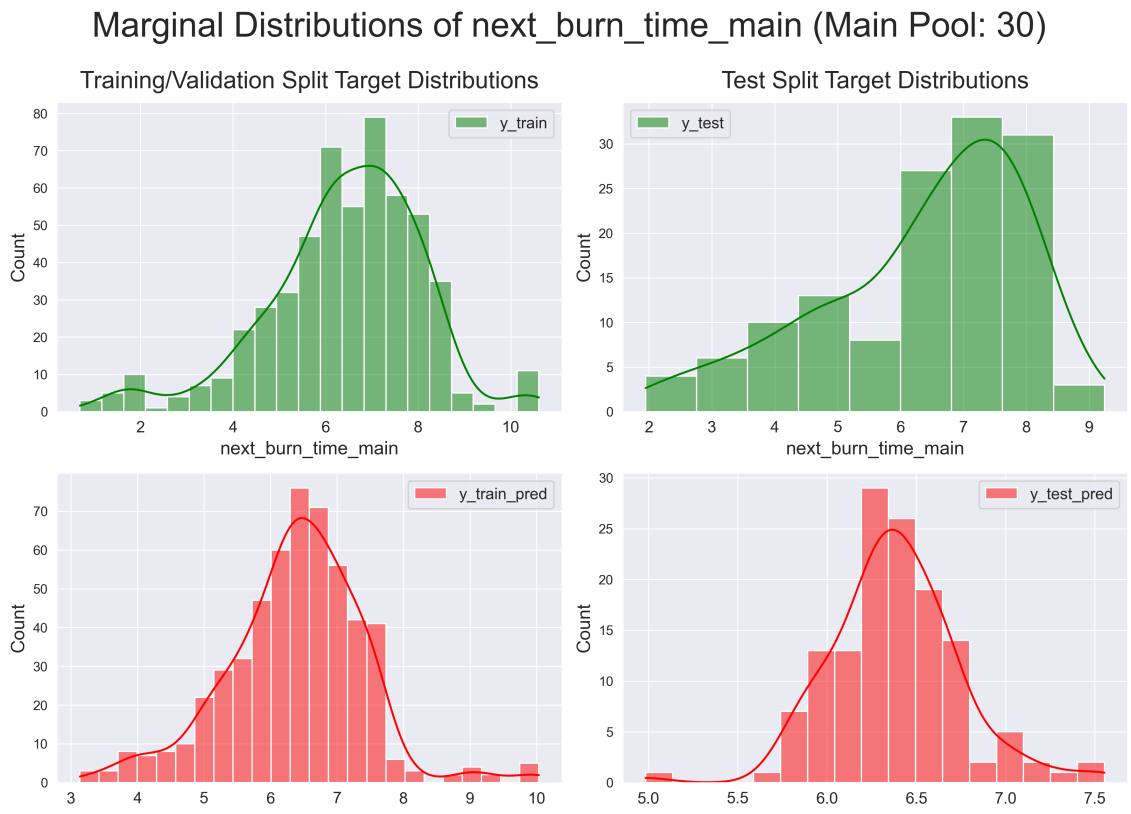



Figure 7.12: Actual and predicted distributions of target *next_burn_time_main*, where the main pool is USDC-WETH-0.003 and predictions are made with an optimized `RandomForestRegressor` estimator.

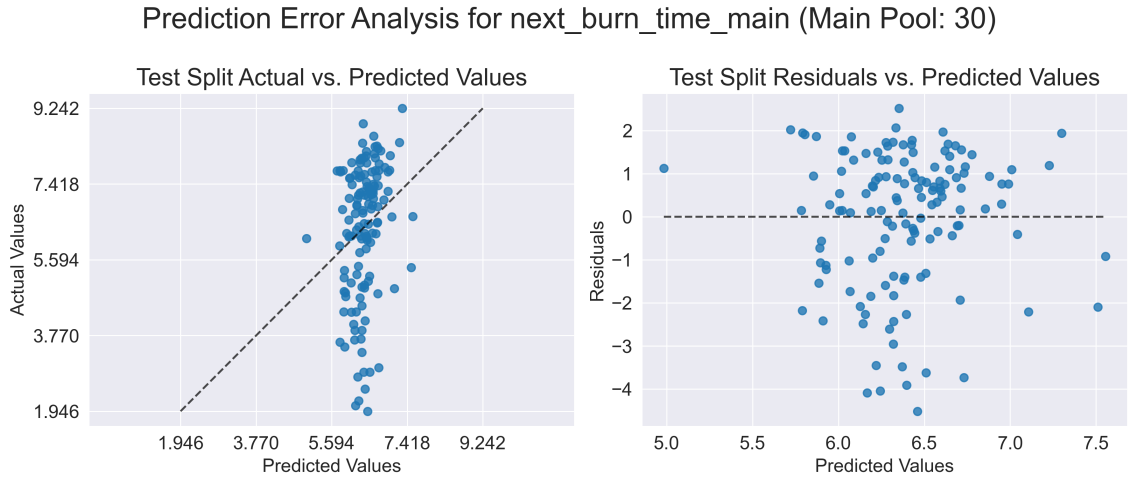



Figure 7.13: Plots of target vs. predicted values and residuals vs. predicted values for regression target *next_burn_time_main*. The main pool is USDC-WETH-0.003 and predictions are made on the test split with an optimized `RandomForestRegressor` estimator.

### 7.2.3 Target: *next_mint_time_other*

Table C.7 summarizes regressor performance for predicting *next_mint_time_other* from Pool 30. Interestingly, a simple `Ridge` regressor using a `PowerTransformer` achieves the best test MSLE, with 1.8563. All linear methods outperform tree-based methods on the test split, hinting that simpler prediction rules are more generalizable in this setting. From Figure 7.14 we notice a decent reconstruction of the target across both splits, though with the usual problem of not predicting the target's tails. Because the distribution of *next_mint_time_other* from Pool 30 isn't as skewed as some previous distributions, the residual plot in Figure 7.15 isn't as unbalanced towards negative residuals. This helps explain the lower test MSLE we achieve for this target compared to our previous two targets. Finally, from Figure 7.18c we identify *b1_mint_main* as the biggest contributor to the performance of our `Ridge` estimator. Though it doesn't greatly affect test MSLE, we note the interconnectedness in liquidity provision it highlights between sister pools.

### 7.2.4 Target: *next_burn_time_other*

Finally, Table C.8 summarizes regressor performance for predicting *next_burn_time_other* from Pool 30. A `KNeighborsRegressor` with a `StandardScaler` and 18 `n_neighbors` performs best with a test MSLE of 1.8429. From Table 7.3, we notice that predicting arrival times for events on Pool 30 appears harder and tree ensemble methods perform best, while predicting arrival times for events on Pool 5 is easier and best handled by non-tree-based methods. The `KNeighborsRegressor` boasts a training/validation MSLE of 0, which reflects in Figure 7.16. This doesn't help it predict extreme target values on the test split, with slightly worse identification of left-tail events per Figure 7.17. Finally, Figure 7.18d attributes the most importance to features tracking the type of the current event in Pool 30 and the distance to past events of both types on both pools, which is reasonable.

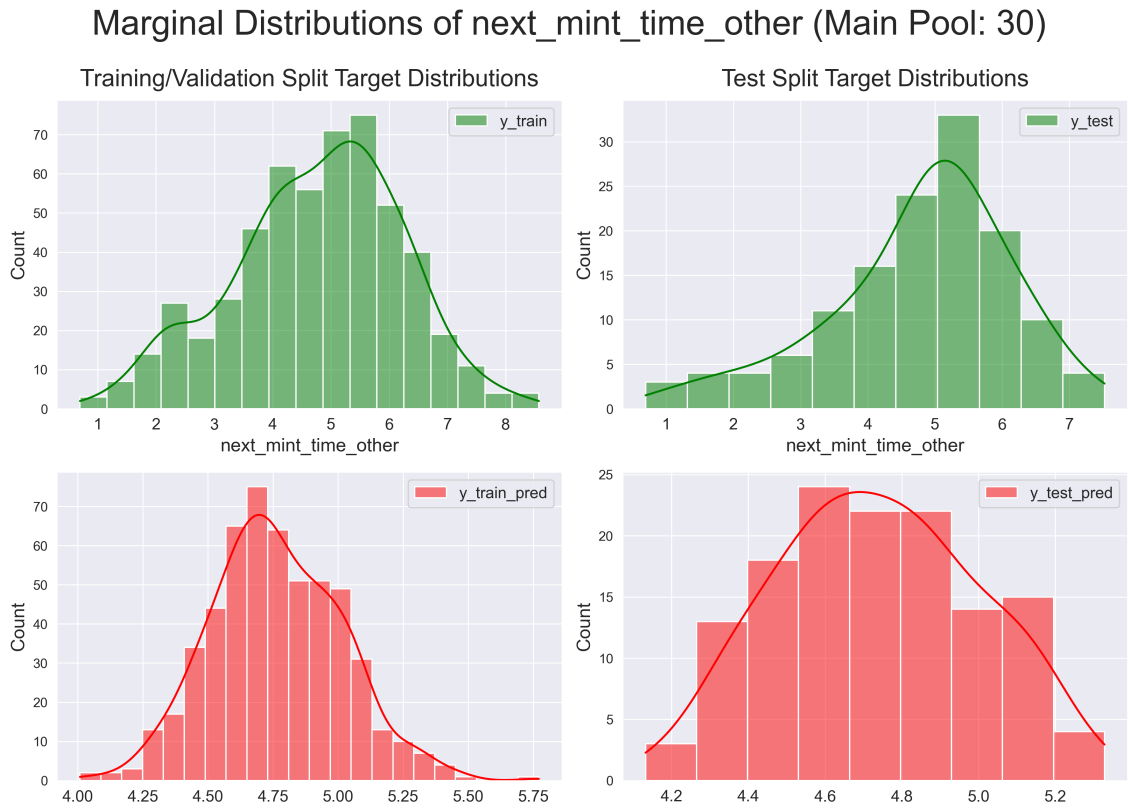



Figure 7.14: Actual and predicted distributions of target *next_mint_time_other*, where the main pool is USDC-WETH-0.003 and predictions are made with an optimized `Ridge` estimator.

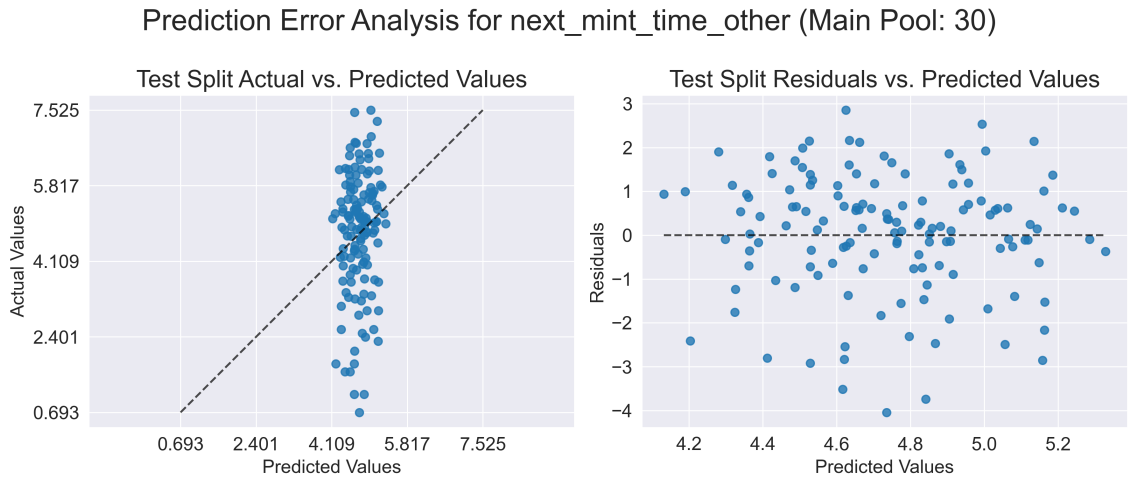



Figure 7.15: Plots of target vs. predicted values and residuals vs. predicted values for regression target *next_mint_time_other*. The main pool is USDC-WETH-0.003 and predictions are made on the test split with an optimized `Ridge` estimator.

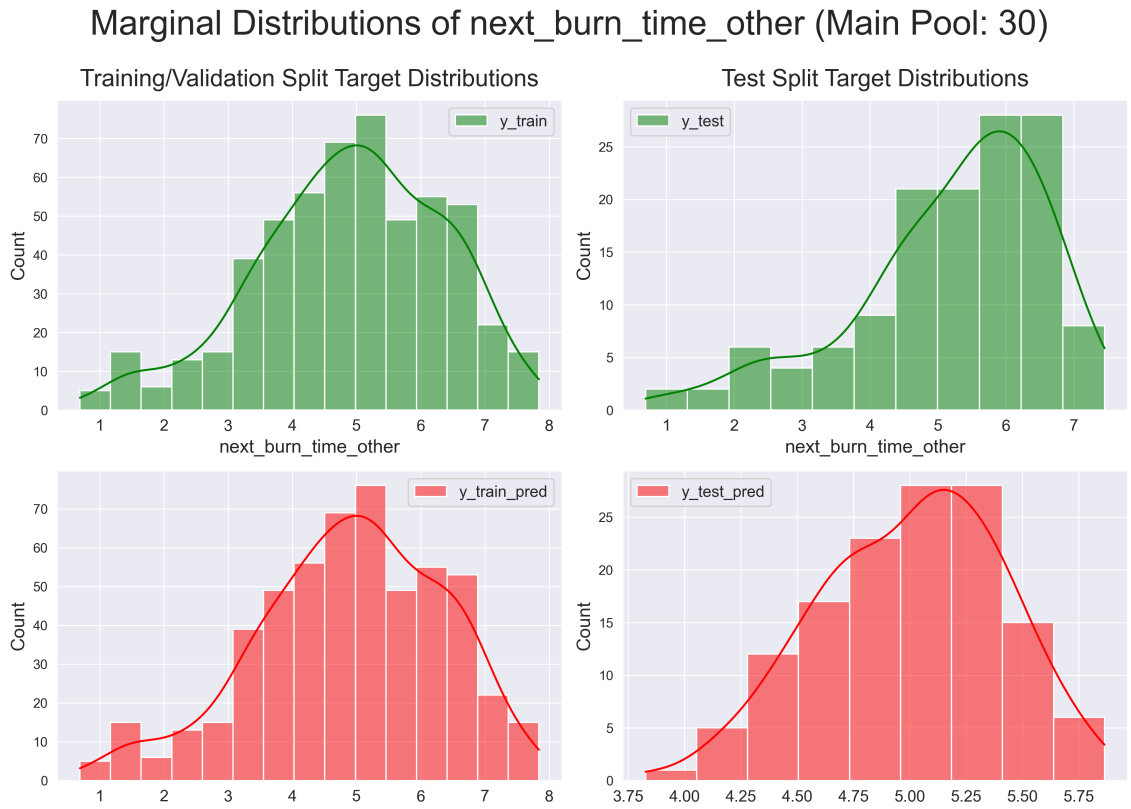



Figure 7.16: Actual and predicted distributions of target *next_burn_time_other*, where the main pool is USDC-WETH-0.003 and predictions are made with an optimized `KNeighborsRegressor` estimator.

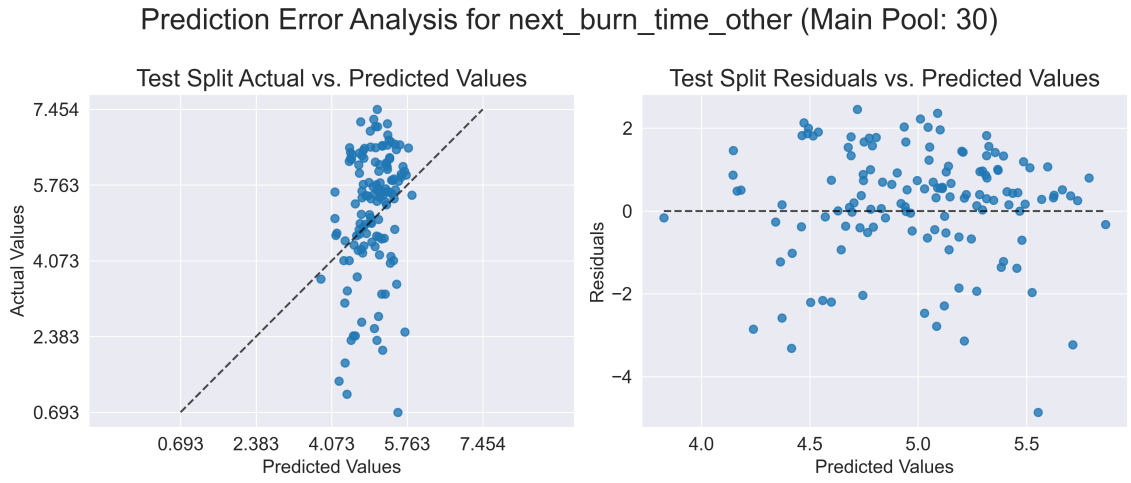
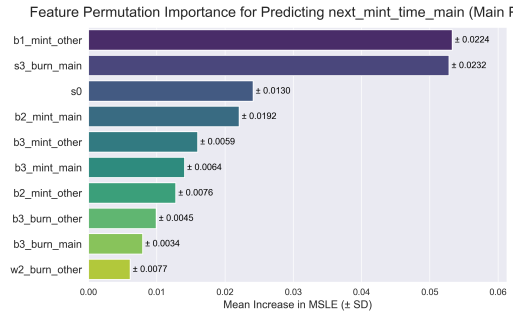


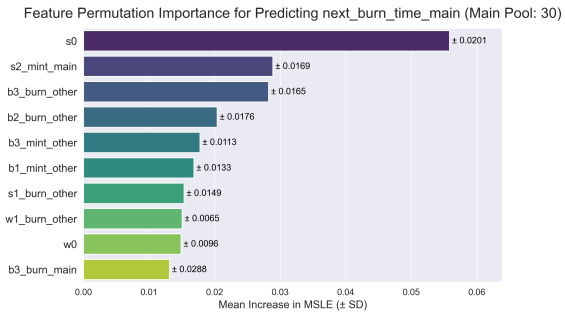

Figure 7.17: Plots of target vs. predicted values and residuals vs. predicted values for regression target *next_burn_time_other*. The main pool is USDC-WETH-0.003 and predictions are made on the test split with an optimized `KNeighborsRegressor` estimator.

## 7.3 Discussion

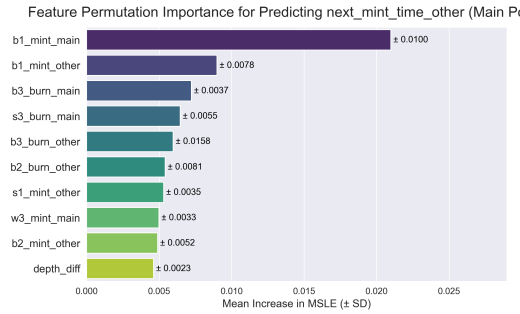
For each regression target, we found estimators that achieved a significantly lower test MSLE than the baseline persistence model. Non-tree-based methods were selected to



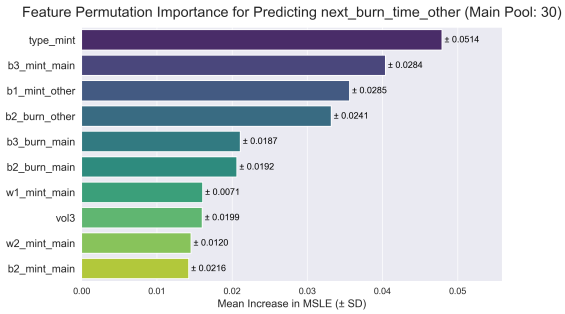

(a) Estimator: `XGBRegressor`.




(b) Estimator: `RandomForestRegressor`.




(c) Estimator: `Ridge`.




(d) Estimator: `KNeighborsRegressor`.

Figure 7.18: Mean permutation importance (with standard deviation) over 50 iterations for the 10 most important features in predicting *next_mint_time_main*, *next_burn_time_main*, *next_mint_time_other*, and *next_burn_time_other* on the test split with the corresponding best estimator, where the main pool is USDC-WETH-0.003.

forecast arrival times of events on Pool 5 for both datasets, while tree-based methods were chosen to forecast arrival times of Pool 30 mints and burns in 3 of 4 occasions. At the same time, we consistently found our best estimators fail to predict tail values of their respective target's test distribution. Given that some tuned estimators in Appendix C achieved near-zero training/validation MSLE, but considerably higher test MSLE, it's plausible the dataset shift explored in Table 6.7 positively biased their generalization MSLE estimates. It could be that the estimator behavior in Figure 7.3, where target tails are recovered for the training/validation but not for the test split, is because certain features in the test split don't take on key values they take on in the training/validation split. Simultaneously, the feature importance plots in this chapter don't highlight features which greatly affect test MSLE upon permutation, so a more informative feature space may also be required for better regression performance.

# Chapter 8

# Conclusion

## 8.1 Synthesis

In this paper, we produced the first direct forecasts of liquidity dynamics in Uniswap v3 liquidity pools through a comprehensive supervised learning framework. We first provided a technical description of the market making and liquidity mechanisms of the leading DEX Uniswap v3, highlighting the complex interplay between liquidity providers and traders that makes this a unique setting with respect to liquidity dynamics. We identified a literature gap concerning the aggregate flow of liquidity over time, which is a salient issue given its direct impact on price slippage. Hence, we set out to address this by leveraging our access to blockchain panel data concerning Uniswap v3 liquidity events and snapshots for the period April 14, 2023 - June 28, 2023. We devised a systematic procedure to preprocess blockchain panel data for supervised learning tasks, engineering features inspired by the literature which track previous liquidity and price trends in a liquidity pool and its sister pool, capturing widely supported spillover effects. We applied this procedure to create two datasets: one where observations are liquidity operations in pool USDC-WETH-0.0005 (the largest Uniswap v3 pool by trading volume and TVL), and one where observations are liquidity operations in pool USDC-WETH-0.003 (its inextricably linked sister pool).

To represent incoming liquidity dynamics, we chose as targets the **type** (mint or burn) of the next liquidity operation in each pool and the **arrival time** of the next operation of each type in each pool, predicting these from each dataset. This yielded 4 classification targets and 8 regression targets. We devised a classification and a regression supervised learning pipeline that tunes a comprehensive range of appropriate methods using randomized search over carefully-built hyperparameter spaces and a validation method that preserves the data's temporal structure, simulating forecasting. For each classification and regression target, we made predictions on a held-out final 20% test split with the best time series cross-validated estimator of each model. We then compared the estimator with the best test performance for each target against a persistence baseline and evaluated it by inspecting its feature importances and strengths and weaknesses in reconstructing the target.

For classification targets, our scoring metric was accuracy. Our best estimators achieved higher test-split accuracy than the baseline for every task except predicting the type of

the next operation in USDC-WETH-0.0005 from operations in USDC-WETH-0.003. We produced more accurate forecasts of the next type of operation in a pool by predicting from that same pool's operations, but we observed frequent evidence of spillover effects through feature importance. The four best estimators were each different methods of various complexities, from a simple ridge classifier to a LightGBM classifier made up of 158 weak learners. Although our supervised learning approach outperformed the baseline in 3 of 4 cases, test split accuracies were practically low (under 60%), which we address in Section 8.2.

For regression targets, our scoring metric was mean squared log error (MSLE). Our best estimators achieved materially lower test-split MSLE than the baseline for every target. Here, we produced lower MSLE forecasts for the arrival time of the next operation of both types in a pool by predicting from the other pool's operations, highlighting the importance of spillover effects. The 8 best estimators represented a roughly equal mix of tree and non-tree-based methods. Linear and distance-based methods were the best estimators for the arrival time of operations in USDC-WETH-0.0005, while ensemble tree methods outperformed in test MSLE for the arrival time of operations in USDC-WETH-0.003. Though our estimators comfortably outperformed their baselines, we identified consistent shortcomings in their predictions. We elaborate on these findings and their implications next.

## 8.2 Limitations & Future Work

A limitation of our best regressors is that they universally failed to predict values at the tails of their target's test distribution, even when their predictions on the training/validation split perfectly matched the target. Thus, the more skewed the target's test destribution, the higher MSLE its best estimator achieved. A similar problem for classification was that the classifier which didn't outperform the baseline made perfect predictions on the training/validation split, meaning it learned poorly generalizable rules during time series cross-validation. Flexible estimators (e.g., k-nearest neighbors and random forests) often made near-perfect predictions on the training/validation split while producing poor cross-validation and test scores, even though their hyperparameter spaces did not encourage overfitting. Computing Kullback-Leibler divergences from training/validation feature distributions to test split feature distributions signalled dataset shift. This is reasonable in a setting as volatile as cryptocurrency trading. A key assumption of supervised learning methods is no dataset shift, and our finding can help explain why our best estimators struggle to identify tail target values in the test split and/or learn effective rules on the training/validation split which don't generalize well to test data.

A concurrent limitation is that across forecasting tasks, our feature importance analysis did not identify features which greatly affect an estimator's test score. Indeed, our feature selection is inspired by a previous study predicting trading volumes in sister Uniswap v3 pools, so there is no theoretical guarantee for their predictive power. Moreover, we did not have access to trading data, which has empirical and theoretical support for its links to liquidity dynamics and would have significantly augmented our feature space. Other obvious limitations are the short time range of our data, which makes the appearance of dataset shift more likely as unusual days stand out more; and our need to forward-fill

missing snapshot data, which inaccurately presents the evolution of prices and market depth.

The most immediate extensions address the above limitations. With complete snapshot data, we could repeat the analysis and 6 of our 46 features would have much richer content. These features were scarcely identified as important for our estimators' performance, but in their true form could improve predictive performance. Accessing blockchain data over a longer period would improve the robustness of our performance estimates, and could combat dataset shift. Without probing into a longer period of observations, it remains unclear whether dataset shift is a consequence of our limited sample size or whether it is unavoidable. In the latter case, a proper solution may require departing from our comprehensive supervised learning framework in favor of a dedicated time series approach. This would likely restrict our feature space and the complexity of prediction rules, but could improve generalization performance by projecting the forecasting problem to a stationary setting. An added benefit is forecast intervals for regression targets would be more easily obtainable.

After regarding current limitations, a wide range of extensions is available. A natural starting point is investigating the consistency between our regression and classification predictions by regarding which event type is predicted a lower arrival time. This would make the seemingly different tasks comparable and might indicate a separate classification task is not necessary. Furthermore, expanding the feature space with information about trading activity in the pools under study and other related pools (as in [19]) and information about common Uniswap v3 liquidity trends (inspired by [41]) is a promising avenue for improving forecast performance. More practically, it is important to extend liquidity dynamics forecasts to the size and width of incoming liquidity operations. These are required inputs for forecasting the price slippage of trades, which is a downstream extension we envision for this work. Overall, we believe this paper presents a principled approach to forecasting liquidity dynamics in Uniswap v3 pools which shows promising results despite data constraints and which can achieve better performance upon addressing existing limitations.

# Bibliography


[1] D. Yaga, P. Mell, N. Roby, and K. Scarfone, "Blockchain technology overview," Tech. Rep., Oct. 2018. DOI: 10.6028/nist.ir.8202. [Online]. Available: https://doi.org/10.6028%2Fnist.ir.8202.

[2] Y. Chen and C. Bellavitis, "Blockchain disruption and decentralized finance: The rise of decentralized business models," *Journal of Business Venturing Insights*, vol. 13, e00151, 2020, ISSN: 2352-6734. DOI: https://doi.org/10.1016/j.jbvi.2019.e00151. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352673419300824.

[3] "Decentralized finance (defi)." (), [Online]. Available: https://ethereum.org/en/defi/.

[4] B. Endres. "Defi statistics 2023." (Apr. 2023), [Online]. Available: https://www.finder.com/defi-statistics.

[5] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, 2008.

[6] F. Corva. "Blockchain statistics 2023." (May 2023), [Online]. Available: https://www.finder.com/blockchain-statistics.

[7] V. Buterin *et al.*, "Ethereum: A next-generation smart contract and decentralized application platform," *Whitepaper*, vol. 3, no. 37, pp. 2–1, 2014.

[8] S. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. Knottenbelt, "Sok: Decentralized finance (defi)," New York, NY, USA: Association for Computing Machinery, 2023, ISBN: 9781450398619. DOI: 10.1145/3558535.3559780. [Online]. Available: https://doi.org/10.1145/3558535.3559780.

[9] L. X. Lin, E. Budish, L. W. Cong, *et al.*, "Deconstructing decentralized exchanges," *Stanford Journal of Blockchain Law & Policy*, vol. 2, no. 1, pp. 58–77, 2019.

[10] Á. Cartea, F. Drissi, and M. Monga, "Decentralised finance and automated market making: Execution and speculation," *arXiv:2307.03499*, 2023.

[11] I. Makarov and A. Schoar, "Trading and arbitrage in cryptocurrency markets," *Journal of Financial Economics*, vol. 135, no. 2, pp. 293–319, 2020.

[12] G. Angeris and T. Chitra, "Improved price oracles," in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, ACM, Oct. 2020. DOI: 10.1145/3419614.3423251. [Online]. Available: https://doi.org/10.1145%2F3419614.3423251.

[13] H. Adams, N. Zinsmeister, and D. Robinson, "Uniswap v2 core, 2020," 2020. [Online]. Available: https://uniswap.org/whitepaper.pdf.

[14] H. Adams, "Uniswap whitepaper," 2018. [Online]. Available: https://hackmd.io/@HaydenAdams/HJ9jLsfTz.

[15] G. Liao and D. Robinson, "The dominance of uniswap v3 liquidity," 2022. [Online]. Available: https://uniswap.org/TheDominanceofUniswapv3Liquidity.pdf.

[16] G. Liao and D. Robinson. "The dominance of uniswap v3 liquidity." (May 2022), [Online]. Available: https://blog.uniswap.org/uniswap-v3-dominance.

[17] H. Adams, N. Zinsmeister, M. Salem, R. Keefer, and D. Robinson, "Uniswap v3 core," 2021. [Online]. Available: https://uniswap.org/whitepaper-v3.pdf.

[18] "Introducing uniswap v3." (Mar. 2021), [Online]. Available: https://blog.uniswap.org/uniswap-v3.

[19] D. Miori and M. Cucuringu, "Defi: Modeling and forecasting trading volume on uniswap v3 liquidity pools," *Available at SSRN 4445351*, 2023.

[20] E. Budish, P. Cramton, and J. Shim, "Tthe high-frequency trading arms race: Frequent batch auctions as a market design response," *The Quarterly Journal of Economics*, vol. 130, no. 4, pp. 1547–1621, 2015.

[21] G. Angeris, A. Agrawal, A. Evans, T. Chitra, and S. Boyd, "Constant function market makers: Multi-asset trades via convex optimization," in *Handbook on Blockchain*, Springer, 2022, pp. 415–444.

[22] J. Xu, K. Paruch, S. Cousaert, and Y. Feng, "Sok: Decentralized exchanges (dex) with automated market maker (amm) protocols," *ACM Computing Surveys*, vol. 55, no. 11, pp. 1–50, 2023.

[23] F. Martinelli and N. Mushegian, "A non-custodial portfolio manager, liquidity provider, and price sensor," 2019. [Online]. Available: https://balancer.fi/whitepaper.pdf.

[24] M. Egorov, "Stableswap - efficient mechanism for stablecoin liquidity," 2019. [Online]. Available: https://berkeley-defi.github.io/assets/material/StableSwap.pdf.

[25] Y. Zhang, X. Chen, and D. Park, "Formal specification of constant product (xy=k) market maker model and implementation," 2018. [Online]. Available: https://github.com/runtimeverification/verified-smart-contracts/blob/master/uniswap/x-y-k.pdf.

[26] Y. Bar-On and Y. Mansour, "Uniswap liquidity provision: An online learning approach," *arXiv:2302.00610*, 2023.

[27] M. Neuder, R. Rao, D. J. Moroz, and D. C. Parkes, "Strategic liquidity provision in uniswap v3," *arXiv:2106.12033*, 2021.

[28] L. Zhou, K. Qin, C. F. Torres, D. V. Le, and A. Gervais, "High-frequency trading on decentralized on-chain exchanges," in *2021 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2021, pp. 428–445.

[29] Á. Cartea, F. Drissi, and M. Monga, "Predictable losses of liquidity provision in constant function markets and concentrated liquidity markets," *Available at SSRN 4541034*, 2023.

[30] R. Durand Saradjian and A. Melachrinos. "Liquidity distribution on uniswap v3." (Dec. 2022), [Online]. Available: https://marketing.kaiko.com/hubfs/Factsheets/Uniswap_V3_snapshot_Dec22.pdf.

[31] A. Lehar, C. Parlour, and M. Zoican, "Liquidity fragmentation on decentralized exchanges," *arXiv:2307.13772*, 2023.

[32] L. Heimbach, E. Schertenleib, and R. Wattenhofer, "Risks and returns of uniswap v3 liquidity providers," *arXiv:2205.08904*, 2022.
[33] Z. Fan, F. J. Marmolejo-Cossío, B. Altschuler, H. Sun, X. Wang, and D. Parkes, "Differential liquidity provision in uniswap v3 and implications for contract design," in *Proceedings of the Third ACM International Conference on AI in Finance*, 2022, pp. 9–17.
[34] R. Almgren, "Optimal trading with stochastic liquidity and volatility," *SIAM Journal on Financial Mathematics*, vol. 3, no. 1, pp. 163–181, 2012.
[35] C. Conrad, A. Custovic, and E. Ghysels, "Long- and short-term cryptocurrency volatility components: A garch-midas analysis," *Journal of Risk and Financial Management*, vol. 11, no. 2, p. 23, 2018.
[36] A. F. Perold, "The implementation shortfall: Paper versus reality," *Journal of Portfolio Management*, vol. 14, no. 3, p. 4, 1988.
[37] I. Aldridge, "Slippage in amm markets," *Available at SSRN 4133897*, 2022.
[38] B. Krishnamachari, Q. Feng, and E. Grippo, "Dynamic curves for decentralized autonomous cryptocurrency exchanges," *arXiv:2101.02778*, 2021.
[39] D. Miori and M. Cucuringu, "Defi: Data-driven characterisation of uniswap v3 ecosystem & an ideal crypto law for liquidity pools," *arXiv:2301.13009*, 2022.
[40] R. Fritsch, "Concentrated liquidity in automated market makers," in *Proceedings of the 2021 ACM CCS Workshop on Decentralized Finance and Security*, 2021, pp. 15–20.
[41] E. Djanga, C. Zhang, and M. Cucuringu, *Cryptocurrency volatility forecasting using commonality in intraday volatility*, Submitted to International Conference on Artificial Intelligence in Finance (ICAIF)., 2023.
[42] C. Zhang, Y. Zhang, M. Cucuringu, and Z. Qian, "Volatility forecasting with machine learning and intraday commonality," *arXiv:2202.08962*, 2022.
[43] "Pancakeswap intro." (), [Online]. Available: https://docs.pancakeswap.finance/.
[44] "Bnb smart chain." (), [Online]. Available: https://github.com/bnb-chain/whitepaper/blob/master/WHITEPAPER.md.
[45] "Tether: Fiat currencies on the bitcoin blockchain." (), [Online]. Available: https://whitepaper.io/document/6/tether-whitepaper.
[46] "Busd: All you need to know about the stablecoin." (Nov. 2021), [Online]. Available: https://www.binance.com/en/blog/futures/busd-all-you-need-to-know-about-the-stablecoin-421499824684903051.
[47] "Introducing drip." (), [Online]. Available: https://drip.community/docs/DRIP_LIGHTPAPER_v0.8_Lit_Version.pdf.
[48] C. Smith. "Proof-of-stake (pos)." (Jul. 2023), [Online]. Available: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/.
[49] "Usd coin whitepaper." (May 2018), [Online]. Available: https://whitepaper.io/document/716/usd-coin-whitepaper.
[50] "Eth vs weth: What's the difference?" (Nov. 2021), [Online]. Available: https://coinmarketcap.com/alexandria/article/eth-vs-weth-what-s-the-difference.

[51] G. Vardy. "Automated market makers: It's curves all the way down." (Jun. 2022), [Online]. Available: https://medium.com/nethermind-eth/automated-market-makers-its-curves-all-the-way-down-1b7804cae8c7.
[52] X. Wan and A. Adams, "Just-in-time liquidity on the uniswap protocol," *Available at SSRN 4382303*, 2022.
[53] I.-K. Yeo and R. A. Johnson, "A new family of power transformations to improve normality or symmetry," *Biometrika*, vol. 87, no. 4, pp. 954–959, 2000.
[54] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, no. 2, 2012.
[55] S. Saxena. "What's the difference between rmse and rmsle?" (Feb. 2020), [Online]. Available: https://medium.com/analytics-vidhya/root-mean-square-log-error-rmse-vs-rmlse-935c6cc1802a.
[56] M. Kapronczay. "Mean squared error (mse) vs. mean squared logarithmic error (msle): A guide." (Apr. 2023), [Online]. Available: https://builtin.com/data-science/msle-vs-mse.
[57] F. Caron. "Statistical machine learning lecture notes: Supervised learning." (2023).
[58] L. Breiman, "Random forests," *Machine learning*, vol. 45, pp. 5–32, 2001.
[59] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, "Do we need hundreds of classifiers to solve real world classification problems?" *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3133–3181, 2014.
[60] F. Caron. "Statistical machine learning: Random forests." (2023).
[61] F. Caron. "Statistical machine learning: Boosting." (2023).
[62] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Annals of Statistics*, pp. 1189–1232, 2001.
[63] A. Natekin and A. Knoll, "Gradient boosting machines, a tutorial," *Frontiers in Neurorobotics*, vol. 7, p. 21, 2013.
[64] "Introduction to boosted trees." (2022), [Online]. Available: https://xgboost.readthedocs.io/en/stable/tutorials/model.html.
[65] L. Grinsztajn, E. Oyallon, and G. Varoquaux, "Why do tree-based models still outperform deep learning on typical tabular data?" *Advances in Neural Information Processing Systems*, vol. 35, pp. 507–520, 2022.
[66] V. Borisov, T. Leemann, K. Seßler, J. Haug, M. Pawelczyk, and G. Kasneci, "Deep neural networks and tabular data: A survey," *IEEE Transactions on Neural Networks and Learning Systems*, 2022.
[67] "State of machine learning and data science 2021." (Oct. 2021), [Online]. Available: https://www.kaggle.com/kaggle-survey-2021.
[68] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 785–794.
[69] G. Ke, Q. Meng, T. Finley, *et al.*, "Lightgbm: A highly efficient gradient boosting decision tree," *Advances in Neural Information Processing Systems*, vol. 30, 2017.
[70] L. Prokhorenkova, G. Gusev, A. Vorobev, A. V. Dorogush, and A. Gulin, "Catboost: Unbiased boosting with categorical features," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[71] E. Al Daoud, "Comparison between xgboost, lightgbm and catboost using a home credit dataset," *International Journal of Computer and Information Engineering*, vol. 13, no. 1, pp. 6–10, 2019.

[72] C. Bentéjac, A. Csörgő, and G. Martínez-Muñoz, "A comparative analysis of gradient boosting algorithms," *Artificial Intelligence Review*, vol. 54, pp. 1937–1967, 2021.

[73] J. H. Friedman, "Stochastic gradient boosting," *Computational Statistics & Data Analysis*, vol. 38, no. 4, pp. 367–378, 2002.

[74] H. Shi, "Best-first decision tree learning," Ph.D. dissertation, The University of Waikato, 2007.

[75] "Lightgbm features." (2023), [Online]. Available: https://lightgbm.readthedocs.io/en/stable/Features.html.

[76] Y. Lou and M. Obukhov, "Bdt: Gradient boosted decision tables for high accuracy and scoring efficiency," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 1893–1901.

[77] T. Rainforth. "Advanced topics in statistical machine learning: Kernel methods." (2023).

[78] T. Rainforth. "Advanced topics in statistical machine learning: Deep learning." (2023).

[79] F. Caron. "Statistical machine learning: Neural networks." (2023).

[80] J. Brownlee. "How to make baseline predictions for time series forecasting with python." (Aug. 2019), [Online]. Available: https://machinelearningmastery.com/persistence-time-series-forecasting-with-python/.

[81] J. G. Moreno-Torres, T. Raeder, R. Alaiz-Rodríguez, N. V. Chawla, and F. Herrera, "A unifying view on dataset shift in classification," *Pattern Recognition*, vol. 45, no. 1, pp. 521–530, 2012.

# Appendix A

# Full List of Features

Table A.1: List of all 46 features and their corresponding labels. We build these for both datasets; a pool is the main pool if the observations are its liquidity operations.

| Label | Feature |
|---|---|
| **Contemporaneous Liquidity Features** | |
| *type* | Labelled "mint" if the current operation is a mint and "burn" otherwise. |
| *s0* | The current operation's size in USD. |
| *w0* | The current operation's width, measured by tick range. |
| *liquidity_type* | Labelled "active" if the current operation's tick range includes the current tick and "inactive" otherwise. |
| **Lagged Main-Pool Features** | |
| *b{n}_mint_main* | The number of blocks between the current operation and the $n$-th last mint operation in the main pool for $n \in \{1, 2, 3\}$. |
| *b{n}_burn_main* | The number of blocks between the current operation and the $n$-th last burn operation in the main pool for $n \in \{1, 2, 3\}$. |
| *s{n}_mint_main* | The size in USD of the $n$-th last mint operation in the main pool for $n \in \{1, 2, 3\}$. |
| *s{n}_burn_main* | The size in USD of the $n$-th last burn operation in the main pool for $n \in \{1, 2, 3\}$. |
| *w{n}_mint_main* | The width in tick range of the $n$-th last mint operation in the main pool for $n \in \{1, 2, 3\}$. |
| *w{n}_burn_main* | The width in tick range of the $n$-th last burn operation in the main pool for $n \in \{1, 2, 3\}$. |
| **Lagged Other-Pool Features** | |
| *b{n}_mint_other* | The number of blocks between the current operation and the $n$-th last mint operation in the other pool for $n \in \{1, 2, 3\}$. |
| *b{n}_burn_other* | The number of blocks between the current operation and the $n$-th last burn operation in the other pool for $n \in \{1, 2, 3\}$. |
| *s{n}_mint_other* | The size in USD of the $n$-th last mint operation in the other pool for $n \in \{1, 2, 3\}$. |
| *s{n}_burn_other* | The size in USD of the $n$-th last burn operation in the other pool for $n \in \{1, 2, 3\}$. |
| *w{n}_mint_other* | The width in tick range of the $n$-th last mint operation in the other pool for $n \in \{1, 2, 3\}$. |
| *w{n}_burn_other* | The width in tick range of the $n$-th last burn operation in the other pool for $n \in \{1, 2, 3\}$. |
| **Price & Market Depth Features** | |
| *depth_diff* | The current market depth in pool USDC-WETH-0.003 minus the current market depth in pool USDC-WETH-0.0005. |
| *depth_ratio* | The current market depth in pool USDC-WETH-0.003 divided by the current market depth in pool USDC-WETH-0.0005. |
| *price_diff* | The current price in the main pool minus the current price in the other pool. |
| *vol_{n}* | Price volatility between the current operation and the $n$-th last operation in the main pool as measured by standard deviation for $n \in \{1, 2, 3\}$. |

# Appendix B

# Classification Performance Tables

Table B.1: Supervised learning pipeline outcomes for classifying *next_type_main* where the main pool is USDC-WETH-0.0005. 95% confidence intervals are Wald intervals and the lower limit is colored in comparison to the baseline test accuracy of 0.4618. $p = 46$ is the number of features.

| Classifier | Acc. | 95% C.I. | T/V Acc. | Search (s) | Prediction (s) | Transformation | Parameters |
|---|---|---|---|---|---|---|---|
| LogisticRegression | 0.5184 | [0.4816, 0.5553] | 0.5535 | 116.1966 | 0.0574 | StandardScaler | C: 0.0007, max_iter: 100 |
| RidgeClassifier | **0.5354** | [0.4986, 0.5722] | 0.5738 | 86.3244 | 0.0644 | StandardScaler | alpha: 89.7569 |
| SVC | 0.5184 | [0.4816, 0.5553] | 0.5688 | 158.5933 | 0.1327 | StandardScaler | C: 0.2428, gamma: 0.0132, kernel: 'poly' |
| KNeighborsClassifier | 0.5099 | [0.473, 0.5468] | 1 | 105.5718 | 0.1142 | StandardScaler | n_neighbors: 89, p: 1, weights: 'distance' |
| MLPClassifier | 0.5255 | [0.4887, 0.5623] | 0.5355 | 560.3905 | 0.0504 | StandardScaler | alpha: 27.3994, hidden_layer_sizes: (200, ) |
| DecisionTreeClassifier | 0.5255 | [0.4887, 0.5623] | 0.5355 | 5.1174 | 0.0000 | None | max_depth: 1, min_samples_leaf: 8 |
| RandomForestClassifier | 0.5255 | [0.4887, 0.5623] | 1 | 341.626 | 0.0302 | None | max_features: 'log2', n_estimators: 172 |
| AdaBoostClassifier | 0.4788 | [0.4419, 0.5156] | 0.861 | 344.8497 | 0.0090 | None | estimator: DecisionTreeClassifier(max_depth = 3), n_estimators: 44 |
| XGBClassifier | 0.5255 | [0.4887, 0.5623] | 0.5355 | 76.2601 | 0.0080 | None | learning_rate: 0.0086, max_depth: 1, min_split_loss: 0.9016, n_estimators: 112, reg_alpha: 8.5067, reg_lambda: 28.4645 |
| LGBMClassifier | 0.5255 | [0.4887, 0.5623] | 0.5355 | 417.0068 | 0.0020 | None | learning_rate: 0.001, num_leaves: 17, max_bin: 510, feature_fraction_bynode: $\log_2(p)$, n_estimators: 153, reg_alpha: 48.6345, reg_lambda: 43.7649 |
| CatBoostClassifier | 0.5255 | [0.4887, 0.5623] | 0.5355 | 323.0668 | 0.0000 | None | learning_rate: 0.0009, depth: 2, l2_leaf_reg: 0.1453, leaf_estimation_iterations: 5, n_estimators: 191, rsm: $\log_2(p)$ |

Table B.2: Supervised learning pipeline outcomes for classifying *next_type_other* where the main pool is USDC-WETH-0.0005. 95% confidence intervals are Wald intervals and the lower limit is colored in comparison to the baseline test accuracy of 0.4802. $p = 46$ is the number of features.

| Classifier | Acc. | 95% C.I. | T/V Acc. | Search (s) | Prediction (s) | Transformation | Parameters |
|---|---|---|---|---|---|---|---|
| LogisticRegression | 0.5354 | [0.4986, 0.5722] | 0.5993 | 53.7937 | 0.0582 | StandardScaler | C: 0.0002, max_iter: 425 |
| RidgeClassifier | 0.5227 | [0.4858, 0.5595] | 0.6805 | 52.7984 | 0.0625 | StandardScaler | alpha: 94.0762 |
| SVC | 0.5 | [0.4631, 0.5369] | 0.6809 | 51.5589 | 0.2197 | StandardScaler | C: 18.7856, gamma: 0.0008, kernel: 'rbf' |
| KNeighborsClassifier | 0.5057 | [0.4986, 0.5722] | 1 | 42.1315 | 0.0835 | PowerTransformer | n_neighbors: 71, p: 2, weights: 'distance' |
| MLPClassifier | 0.5354 | [0.4887, 0.5623] | 0.5993 | 210.6569 | 0.0514 | StandardScaler | alpha: 20.6439, hidden_layer_sizes: (100, 100) |
| DecisionTreeClassifier | **0.5567** | [0.52, 0.5933] | 0.6422 | 2.9945 | 0.0000 | None | max_depth: 2, min_samples_leaf: 7 |
| RandomForestClassifier | 0.5071 | [0.4702, 0.544] | 1 | 127.387 | 0.0101 | None | max_features: 'log2', n_estimators: 86 |
| AdaBoostClassifier | 0.4943 | [0.4575, 0.5312] | 0.8809 | 90.7729 | 0.0120 | None | estimator: None, n_estimators: 85 |
| XGBClassifier | 0.5354 | [0.4986, 0.5722] | 0.6206 | 20.9194 | 0.0000 | None | learning_rate: 0.0024, max_depth: 5, min_split_loss: 1.2331, n_estimators: 184, reg_alpha: 0.4096, reg_lambda: 774.32 |
| LGBMClassifier | 0.5354 | [0.4986, 0.5722] | 0.5993 | 28.3328 | 0.0080 | None | learning_rate: 0.001, num_leaves: 17, max_bin: 510, feature_fraction_bynode: $\log_2(p)$, n_estimators: 153, reg_alpha: 48.6345, reg_lambda: 43.7649 |
| CatBoostClassifier | 0.5354 | [0.4986, 0.5722] | 0.5993 | 186.0724 | 0.0100 | None | learning_rate: 0.0009, depth: 1, l2_leaf_reg: 12.5184, leaf_estimation_iterations: 1, n_estimators: 153, rsm: $\log_2(p)$ |

Table B.3: Supervised learning pipeline outcomes for classifying *next_type_main* where the main pool is USDC-WETH-0.003. 95% confidence intervals are Wald intervals and the lower limit is colored in comparison to the baseline test accuracy of 0.5407. $p = 46$ is the number of features.

| Classifier | Acc. | 95% C.I. | T/V Acc. | Search (s) | Prediction (s) | Transformation | Parameters |
|---|---|---|---|---|---|---|---|
| LogisticRegression | 0.5407 | [0.4567, 0.6248] | 0.6350 | 67.7370 | 0.0621 | PowerTransformer | C: 0.0841, max_iter: 152 |
| RidgeClassifier | 0.5407 | [0.4567, 0.6248] | 0.6369 | 86.9518 | 0.0634 | PowerTransformer | alpha: 58.2664 |
| SVC | 0.5556 | [0.4717, 0.6394] | 0.7374 | 72.2549 | 0.0524 | StandardScaler | C: 0.1079, gamma: 0.0758, kernel: 'poly' |
| KNeighborsClassifier | 0.4963 | [0.412, 0.5806] | 0.5885 | 68.6188 | 0.0601 | StandardScaler | n_neighbors: 23, p: 2, weights: 'uniform' |
| MLPClassifier | 0.5704 | [0.4869, 0.6539] | 0.5419 | 476.0037 | 0.0602 | PowerTransformer | alpha: 32.9887, hidden_layer_sizes: (100, ) |
| DecisionTreeClassifier | 0.4667 | [0.3825, 0.5508] | 0.7598 | 2.0931 | 0.0080 | None | max_depth: 8, min_samples_leaf: 17 |
| RandomForestClassifier | 0.5333 | [0.4492, 0.6175] | 1.0000 | 76.0256 | 0.0000 | None | max_features: 'log2', n_estimators: 51 |
| AdaBoostClassifier | 0.4444 | [0.3606, 0.5283] | 1.0000 | 71.7168 | 0.0080 | None | estimator: DecisionTreeClassifier(max_depth = 3), n_estimators: 35 |
| EncodedXGBClassifier | 0.5556 | [0.4717, 0.6394] | 0.7784 | 186.5733 | 0.0080 | None | learning_rate: 0.0009, max_depth: 4, min_split_loss: 1.6466, n_estimators: 137, reg_alpha: 0.2845, reg_lambda: 1.7714 |
| LGBMClassifier | **0.5926** | [0.5097, 0.6755] | 0.6182 | 720.1609 | 0.0000 | None | learning_rate: 0.0369, num_leaves: 10, max_bin: 287, feature_fraction_bynode: 1, n_estimators: 158, reg_alpha: 13.3515, reg_lambda: 1.1837 |
| CatBoostClassifier | 0.5333 | [0.4492, 0.6175] | 0.8603 | 372.8456 | 0.0000 | None | learning_rate: 0.0009, depth: 3, l2_leaf_reg: 7.071, leaf_estimation_iterations: 5, n_estimators: 119, rsm: 1 |

Table B.4: Supervised learning pipeline outcomes for classifying *next_type_other* where the main pool is USDC-WETH-0.003. 95% confidence intervals are Wald intervals and the lower limit is colored in comparison to the baseline test accuracy of 0.5259. $p = 46$ is the number of features.

| Classifier | Acc. | 95% C.I. | T/V Acc. | Search (s) | Prediction (s) | Transformation | Parameters |
|---|---|---|---|---|---|---|---|
| LogisticRegression | 0.4444 | [0.3606, 0.5283] | 0.6201 | 67.7873 | 0.0588 | StandardScaler | C: 0.0115, max_iter: 167 |
| RidgeClassifier | 0.4741 | [0.3898, 0.5583] | 0.6331 | 85.0461 | 0.0502 | StandardScaler | alpha: 26.6858 |
| SVC | 0.4296 | [0.3461, 0.5131] | 1.0000 | 70.3983 | 0.0601 | StandardScaler | C: 89.6792, gamma: 0.3669, kernel: 'rbf' |
| KNeighborsClassifier | **0.4889** | [0.4046, 0.5732] | 1.0000 | 68.5167 | 0.0522 | StandardScaler | n_neighbors: 24, p: 2, weights: 'distance' |
| MLPClassifier | 0.3852 | [0.3031, 0.4673] | 0.5363 | 540.9638 | 0.0801 | StandardScaler | alpha: 56.9982, hidden_layer_sizes: (200, ) |
| DecisionTreeClassifier | 0.4148 | [0.3317, 0.4979] | 0.6946 | 3.1952 | 0.0000 | None | max_depth: 5, min_samples_leaf: 11 |
| RandomForestClassifier | 0.4741 | [0.3898, 0.5583] | 1.0000 | 86.4985 | 0.0301 | None | max_features: 'sqrt', n_estimators: 248 |
| AdaBoostClassifier | 0.4519 | [0.3679, 0.5358] | 0.8510 | 61.4095 | 0.0100 | None | estimator: None, n_estimators: 54 |
| EncodedXGBClassifier | 0.4000 | [0.3174, 0.4826] | 0.8231 | 481.7843 | 0.0100 | None | learning_rate: 0.0031, max_depth: 5, min_split_loss: 1.0789, n_estimators: 167, reg_alpha: 4.033, reg_lambda: 5.5997 |
| LGBMClassifier | 0.4296 | [0.3461, 0.5131] | 0.8585 | 373.0607 | 0.0080 | None | learning_rate: 0.0761, num_leaves: 18, max_bin: 510, feature_fraction_bynode: 0.25, n_estimators: 109, reg_alpha: 8.5367, reg_lambda: 0.194 |
| CatBoostClassifier | 0.3926 | [0.3102, 0.475] | 0.5717 | 369.0498 | 0.0000 | None | learning_rate: 0.0021, depth: 2, l2_leaf_reg: 0.3209, leaf_estimation_iterations: 1, n_estimators: 192, rsm: $\sqrt{p}$ |

# Appendix C

# Regression Performance Tables

Table C.1: Regression pipeline outcomes for predicting *next_mint_time_main* where the main pool is USDC-WETH-0.0005. 95% confidence intervals are Wald intervals and the upper limit is colored in comparison to the baseline test MSLE of 3.4051. $p = 46$ is the number of features.

| Regressor | MSLE | 95% C.I. | T/V MSLE | Search (s) | Prediction (s) | Transformation | Parameters |
|---|---|---|---|---|---|---|---|
| Ridge | 1.9413 | [1.769, 2.1136] | 2.1957 | 241.4051 | 0.0613 | PowerTransformer | alpha: 1092.7314 |
| Lasso | 2.0126 | [1.8349, 2.1904] | 2.2967 | 215.6088 | 0.0637 | PowerTransformer | alpha: 0.1094 |
| ElasticNet | 1.9404 | [1.7691, 2.1118] | 2.2392 | 188.4546 | 0.0603 | PowerTransformer | alpha: 0.1198, l1_ratio: 0.4746 |
| KernelRidge | **1.9379** | [1.7657, 2.1101] | 2.1740 | 463.7403 | 0.0826 | PowerTransformer | alpha: 0.6462, gamma: 0.0005, kernel: 'rbf' |
| SVR | 1.9414 | [1.768, 2.1148] | 2.2170 | 672.0135 | 0.1407 | PowerTransformer | C: 128.679, epsilon: 1.1132, gamma: 5.4389e-06, kernel: 'rbf' |
| KNeighborsRegressor | 2.0498 | [1.8697, 2.2299] | 0.0000 | 190.0265 | 0.1325 | PowerTransformer | n_neighbors: 88, p: 1, weights: 'distance' |
| MLPRegressor | 2.0275 | [1.8486, 2.2065] | 2.2901 | 5576.4208 | 0.0592 | PowerTransformer | alpha: 96.9377, hidden_layer_sizes: (200, ) |
| DecisionTreeRegressor | 2.1153 | [1.9254, 2.3053] | 2.2844 | 10.6840 | 0.0020 | None | max_depth: 2, min_samples_leaf: 8 |
| RandomForestRegressor | 2.0240 | [1.8482, 2.1998] | 0.4567 | 484.9959 | 0.0201 | None | max_features: 'sqrt', n_estimators: 152 |
| AdaBoostRegressor | 2.0673 | [1.8878, 2.2467] | 2.1316 | 145.9648 | 0.0000 | None | estimator: None, n_estimators: 30 |
| XGBRegressor | 1.9382 | [1.7704, 2.1061] | 2.0118 | 38.5479 | 0.0000 | None | learning_rate: 0.0345, max_depth: 2, min_split_loss: 1.4572, n_estimators: 165, reg_alpha: 2.6649, reg_lambda: 0.1489 |
| LGBMRegressor | 1.9765 | [1.804, 2.1491] | 1.7115 | 47.2701 | 0.0000 | None | learning_rate: 0.0138, num_leaves: 16, max_bin: 393, feature_fraction_bynode: $\sqrt{p}$, n_estimators: 198, reg_alpha: 0.3171, reg_lambda: 0.1481 |
| CatBoostRegressor | 1.9483 | [1.7794, 2.1173] | 2.0150 | 276.0148 | 0.0000 | None | learning_rate: 0.0209, depth: 5, l2_leaf_reg: 102.6957, leaf_estimation_iterations: 5, n_estimators: 196, rsm: $\log_2(p)$ |

Table C.2: Regression pipeline outcomes for predicting *next_burn_time_main* where the main pool is USDC-WETH-0.0005. 95% confidence intervals are Wald intervals and the upper limit is colored in comparison to the baseline test MSLE of 3.9844. $p = 46$ is the number of features.

| Regressor | MSLE | 95% C.I. | T/V MSLE | Search (s) | Prediction (s) | Transformation | Parameters |
|---|---|---|---|---|---|---|---|
| Ridge | 2.1974 | [1.9587, 2.4362] | 2.1061 | 252.7033 | 0.0612 | PowerTransformer | alpha: 2328.8328 |
| Lasso | 2.2261 | [1.9853, 2.4669] | 2.1608 | 209.2040 | 0.0805 | PowerTransformer | alpha: 0.1094 |
| ElasticNet | 2.2021 | [1.9632, 2.441] | 2.1441 | 194.8909 | 0.0604 | PowerTransformer | alpha: 0.1164, l1_ratio: 0.6943 |
| KernelRidge | 2.1973 | [1.9583, 2.4363] | 2.0869 | 451.1353 | 0.0887 | PowerTransformer | alpha: 1,0312, gamma: 0.0004, kernel: 'rbf' |
| SVR | 2.2275 | [1.9751, 2.48] | 2.0114 | 642.5432 | 0.2781 | PowerTransformer | C: 0.0866, epsilon: 0.2283, gamma: 0.019, kernel: 'rbf' |
| KNeighborsRegressor | **2.1615** | [1.9301, 2.3929] | 0.0000 | 194.9255 | 0.1243 | PowerTransformer | n_neighbors: 88, p: 1, weights: 'distance' |
| MLPRegressor | 2.2722 | [2.023, 2.5215] | 2.1619 | 5733.7705 | 0.0606 | PowerTransformer | alpha: 96.9377, hidden_layer_sizes: (200, ) |
| DecisionTreeRegressor | 2.5241 | [2.2364, 2.8118] | 1.7172 | 10.7461 | 0.0000 | None | max_depth: 6, min_samples_leaf: 17 |
| RandomForestRegressor | 2.2364 | [1.9924, 2.4804] | 0.3898 | 425.1898 | 0.0401 | None | max_features: 'log2', n_estimators: 248 |
| AdaBoostRegressor | 2.4847 | [2.2534, 2.7161] | 2.2195 | 208.2133 | 0.0100 | None | estimator: None, n_estimators: 54 |
| XGBRegressor | 2.1814 | [1.9457, 2.4171] | 2.0923 | 197.8203 | 0.0000 | None | learning_rate: 0.0267, max_depth: 1, min_split_loss: 1.5659, n_estimators: 184, reg_alpha: 3.3386, reg_lambda: 4.623 |
| LGBMRegressor | 2.2422 | [1.9985, 2.486] | 2.0046 | 52.6738 | 0.0000 | None | learning_rate: 0.0085, num_leaves: 9, max_bin: 293, feature_fraction_bynode: $\log_2(p)$, n_estimators: 161, reg_alpha: 7.3554, reg_lambda: 0.275 |
| CatBoostRegressor | 2.1711 | [1.9325, 2.4097] | 1.9909 | 248.3100 | 0.0020 | None | learning_rate: 0.0538, depth: 2, l2_leaf_reg: 344.5738, leaf_estimation_iterations: 5, n_estimators: 197, rsm: $\log_2(p)$ |

Table C.3: Regression pipeline outcomes for predicting *next_mint_time_other* where the main pool is USDC-WETH-0.0005. 95% confidence intervals are Wald intervals and the upper limit is colored in comparison to the baseline test MSLE of 3.3366. $p = 46$ is the number of features.

| Regressor | MSLE | 95% C.I. | T/V MSLE | Search (s) | Prediction (s) | Transformation | Parameters |
|---|---|---|---|---|---|---|---|
| Ridge | 2.0206 | [1.7562, 2.285] | 2.0146 | 283.0247 | 0.0610 | PowerTransformer | alpha: 7268.037 |
| Lasso | 2.0713 | [1.798, 2.3447] | 2.1513 | 210.8840 | 0.0468 | PowerTransformer | alpha: 0.2284 |
| ElasticNet | 2.1239 | [1.846, 2.4017] | 2.2159 | 187.5735 | 0.0604 | PowerTransformer | alpha: 4.0733, l1_ratio: 0.8514 |
| KernelRidge | 2.0191 | [1.7547, 2.2835] | 2.0300 | 540.1419 | 0.1308 | PowerTransformer | alpha: 3.4519, gamma: 0.0001, kernel: 'poly' |
| SVR | 2.0407 | [1.7705, 2.311] | 2.0056 | 454.2616 | 0.1370 | PowerTransformer | C: 0.0285, epsilon: 1.3181, gamma: 0.017, kernel: 'rbf' |
| KNeighborsRegressor | **1.9772** | [1.721, 2.2333] | 1.8259 | 204.9135 | 0.1153 | StandardScaler | n_neighbors: 98, p: 1, weights: 'uniform' |
| MLPRegressor | 2.0712 | [1.8031, 2.3392] | 2.1651 | 6070.2205 | 0.0611 | PowerTransformer | alpha: 451.5155, hidden_layer_sizes: (100, ) |
| DecisionTreeRegressor | 2.2289 | [1.9729, 2.4848] | 1.8723 | 7.0614 | 0.0000 | None | max_depth: 3, min_samples_leaf: 6 |
| RandomForestRegressor | 2.2831 | [2.0099, 2.5564] | 0.1657 | 401.7919 | 0.0100 | None | max_features: 'sqrt', n_estimators: 64 |
| AdaBoostRegressor | 2.6773 | [2.4471, 2.9074] | 2.1143 | 288.0771 | 0.0000 | None | estimator: None, n_estimators: 30 |
| XGBRegressor | 2.0453 | [1.7841, 2.3065] | 1.8497 | 120.1245 | 0.0080 | None | learning_rate: 0.0267 max_depth: 1, min_split_loss: 1.5659, n_estimators: 184, reg_alpha: 3.3386, reg_lambda: 4.6228 |
| LGBMRegressor | 2.0943 | [1.8209, 2.3678] | 2.1333 | 120.4355 | 0.0000 | None | learning_rate: 0.0044, num_leaves: 6, max_bin: 321, feature_fraction_bynode: 1, n_estimators: 136, reg_alpha: 257.2405, reg_lambda: 30.2975 |
| CatBoostRegressor | 2.0692 | [1.798, 2.3405] | 2.0811 | 341.2981 | 0.0000 | None | learning_rate: 0.0081, depth: 3, l2_leaf_reg: 835.105, leaf_estimation_iterations: 5, n_estimators: 101, rsm: $\log_2(p)$ |

Table C.4: Regression pipeline outcomes for predicting *next_burn_time_other* where the main pool is USDC-WETH-0.0005. 95% confidence intervals are Wald intervals and the upper limit is colored in comparison to the baseline test MSLE of 3.8002. $p = 46$ is the number of features.

| Regressor | MSLE | 95% C.I. | T/V MSLE | Search (s) | Prediction (s) | Transformation | Parameters |
|---|---|---|---|---|---|---|---|
| Ridge | 1.9287 | [1.6923, 2.165] | 2.1494 | 254.2306 | 0.0628 | PowerTransformer | alpha: 7268.037 |
| Lasso | 1.9685 | [1.7231, 2.2139] | 2.3957 | 209.7233 | 0.0601 | PowerTransformer | alpha: 4.0733 |
| ElasticNet | 1.8961 | [1.6601, 2.1321] | 2.1997 | 196.2121 | 0.0705 | PowerTransformer | alpha: 0.3506, l1_ratio: 0.3716 |
| KernelRidge | 1.9257 | [1.6892, 2.1622] | 2.1658 | 439.6568 | 0.1713 | PowerTransformer | alpha: 3.4519, gamma: 0.0001, kernel: 'poly' |
| SVR | 1.9298 | [1.6751, 2.1845] | 2.1238 | 418.2277 | 0.1916 | StandardScaler | C: 0.3481, epsilon: 0.4163, gamma: 0.0006, kernel: 'rbf' |
| KNeighborsRegressor | 1.9439 | [1.7064, 2.1814] | 0.0000 | 252.4963 | 0.1211 | StandardScaler | n_neighbors: 98, p: 1, weights: 'distance' |
| MLPRegressor | 1.9335 | [1.6869, 2.18] | 2.3067 | 6174.1867 | 0.0618 | PowerTransformer | alpha: 118.8002, hidden_layer_sizes: (200, 100) |
| DecisionTreeRegressor | 2.1509 | [1.9031, 2.3987] | 1.9343 | 7.0496 | 0.0000 | None | max_depth: 3, min_samples_leaf: 18 |
| RandomForestRegressor | 2.2714 | [2.0232, 2.5195] | 0.1740 | 385.5491 | 0.0197 | None | max_features: 'sqrt', n_estimators: 91 |
| AdaBoostRegressor | 3.0859 | [2.8188, 3.353] | 2.2127 | 284.6043 | 0.0000 | None | estimator: None, n_estimators: 30 |
| XGBRegressor | **1.8832** | [1.6529, 2.1134] | 1.9499 | 113.3031 | 0.0020 | None | learning_rate: 0.0267, max_depth: 1, min_split_loss: 1.5659, n_estimators: 184, reg_alpha: 3.3386, reg_lambda: 4.6228 |
| LGBMRegressor | 1.9105 | [1.6719, 2.149] | 2.1673 | 144.1566 | 0.0080 | None | learning_rate: 0.0038, num_leaves: 5, max_bin: 338, feature_fraction_bynode: $\sqrt{p}$, n_estimators: 158, reg_alpha: 66.4233, reg_lambda: 0.2637 |
| CatBoostRegressor | 1.9149 | [1.6754, 2.1544] | 2.2479 | 335.0961 | 0.0100 | None | learning_rate: 0.0058, depth: 1, l2_leaf_reg: 103.7334, leaf_estimation_iterations: 1, n_estimators: 179, rsm: $\sqrt{p}$ |

Table C.5: Regression pipeline outcomes for predicting *next_mint_time_main* where the main pool is USDC-WETH-0.003. 95% confidence intervals are Wald intervals and the upper limit is colored in comparison to the baseline test MSLE of 4.9481. $p = 46$ is the number of features.

| Regressor | MSLE | 95% C.I. | T/V MSLE | Search (s) | Prediction (s) | Transformation | Parameters |
|---|---|---|---|---|---|---|---|
| Ridge | 2.3651 | [1.779, 2.9512] | 2.7536 | 116.8932 | 0.0603 | PowerTransformer | alpha: 2982.1988 |
| Lasso | 2.4390 | [1.8344, 3.0436] | 2.8911 | 99.9401 | 0.0468 | StandardScaler | alpha: 4.0733 |
| ElasticNet | 2.4390 | [1.8344, 3.0436] | 2.8911 | 105.5235 | 0.0604 | PowerTransformer | alpha: 4.0733, l1_ratio: 0.8514 |
| KernelRidge | 2.3509 | [1.7725, 2.9292] | 2.7043 | 305.0730 | 0.0805 | PowerTransformer | alpha: 0.8133, gamma: 0.0002, kernel: 'poly' |
| SVR | 2.4578 | [1.8405, 3.0751] | 2.8933 | 136.5908 | 0.0604 | PowerTransformer | C: 16.4601, epsilon: 1.422, gamma: 3.6509e-06, kernel: 'poly' |
| KNeighborsRegressor | 2.7002 | [1.9879, 3.4126] | 2.4805 | 162.1793 | 0.0566 | StandardScaler | n_neighbors: 21, p: 1, weights: 'uniform' |
| MLPRegressor | 2.3441 | [1.8068, 2.8815] | 2.8418 | 3391.3350 | 0.0550 | PowerTransformer | alpha: 360.8553, hidden_layer_sizes: (100, 50) |
| DecisionTreeRegressor | 2.4084 | [1.8379, 2.9788] | 2.7791 | 5.6434 | 0.0000 | None | max_depth: 1, min_samples_leaf: 17 |
| RandomForestRegressor | 2.3544 | [1.7598, 2.9491] | 0.5926 | 230.6841 | 0.0180 | None | max_features: 'sqrt', n_estimators: 183 |
| AdaBoostRegressor | 2.4938 | [1.9619, 3.0256] | 1.8617 | 89.3428 | 0.0020 | None | estimator: None, n_estimators: 35 |
| XGBRegressor | **2.2297** | [1.6884, 2.7709] | 2.4995 | 191.9634 | 0.0019 | None | learning_rate: 0.0267, max_depth: 1, min_split_loss: 1.5659, n_estimators: 184, reg_alpha: 3.3386, reg_lambda: 4.6228 |
| LGBMRegressor | 2.3575 | [1.7673, 2.9477] | 2.3629 | 857.6551 | 0.0000 | None | learning_rate: 0.001, num_leaves: 19, max_bin: 303, feature_fraction_bynode: $\sqrt{p}$, n_estimators: 172, reg_alpha: 21.7085, reg_lambda: 17.2078 |
| CatBoostRegressor | 2.3585 | [1.7697, 2.9472] | 2.7743 | 357.5263 | 0.0029 | None | learning_rate: 0.0076, depth: 1, l2_leaf_reg: 70.6495, leaf_estimation_iterations: 5, n_estimators: 195, rsm: $\sqrt{p}$ |

Table C.6: Regression pipeline outcomes for predicting *next_burn_time_main* where the main pool is USDC-WETH-0.003. 95% confidence intervals are Wald intervals and the upper limit is colored in comparison to the baseline test MSLE of 5.1637. $p = 46$ is the number of features.

| Regressor | MSLE | 95% C.I. | T/V MSLE | Search (s) | Prediction (s) | Transformation | Parameters |
|---|---|---|---|---|---|---|---|
| Ridge | 2.5564 | [1.8704, 3.2425] | 2.4955 | 110.5174 | 0.0590 | PowerTransformer | alpha: 1092.7314 |
| Lasso | 2.6018 | [1.98, 3.2235] | 2.6285 | 102.5400 | 0.0538 | StandardScaler | alpha: 0.1961 |
| ElasticNet | 2.5883 | [1.9631, 3.2135] | 2.6057 | 97.2448 | 0.0504 | StandardScaler | alpha: 0.3309, l1_ratio: 0.4594 |
| KernelRidge | 2.5633 | [1.8867, 3.2398] | 2.5334 | 294.9410 | 0.0624 | PowerTransformer | alpha: 0.8133, gamma: 0.0002, kernel: 'poly' |
| SVR | 2.6846 | [1.941, 3.4282] | 2.6602 | 152.4937 | 0.0572 | StandardScaler | C: 0.8996, epsilon: 0.3848, gamma: 0.0004, kernel: 'rbf' |
| KNeighborsRegressor | 2.6298 | [2.0291, 3.2306] | 0.0000 | 153.4352 | 0.0522 | StandardScaler | n_neighbors: 24, p: 1, weights: 'distance' |
| MLPRegressor | 2.5290 | [1.8802, 3.1778] | 2.5037 | 3249.2643 | 0.0531 | PowerTransformer | alpha: 118.8002, hidden_layer_sizes: (200, 100) |
| DecisionTreeRegressor | 2.8998 | [2.1948, 3.6048] | 2.4563 | 4.5614 | 0.0000 | None | max_depth: 2, min_samples_leaf: 4 |
| RandomForestRegressor | **2.4751** | [1.8603, 3.0898] | 0.5354 | 225.9393 | 0.0204 | None | max_features: 'log2', n_estimators: 248 |
| AdaBoostRegressor | 3.0361 | [2.2952, 3.7769] | 1.8095 | 144.9161 | 0.0020 | None | estimator: None, n_estimators: 30 |
| XGBRegressor | 2.5330 | [1.9044, 3.1615] | 2.3480 | 244.5415 | 0.0020 | None | learning_rate: 0.0267, max_depth: 1, min_split_loss: 1.5659, n_estimators: 184, reg_alpha: 3.3386, reg_lambda: 4.623 |
| LGBMRegressor | 2.7178 | [2.0661, 3.3695] | 2.1149 | 376.7599 | 0.0080 | None | learning_rate: 0.0054, num_leaves: 19, max_bin: 329, feature_fraction_bynode: 1, n_estimators: 193, reg_alpha: 0.5451, reg_lambda: 26.4519 |
| CatBoostRegressor | 2.5775 | [1.9362, 3.2187] | 2.4093 | 607.2818 | 0.0000 | None | learning_rate: 0.0138, depth: 2, l2_leaf_reg: 0.3623, leaf_estimation_iterations: 1, n_estimators: 152, rsm: $\sqrt{p}$ |

Table C.7: Regression pipeline outcomes for predicting *next_mint_time_other* where the main pool is USDC-WETH-0.003. 95% confidence intervals are Wald intervals and the upper limit is colored in comparison to the baseline test MSLE of 3.5054. $p = 46$ is the number of features.

| Regressor | MSLE | 95% C.I. | T/V MSLE | Search (s) | Prediction (s) | Transformation | Parameters |
|---|---|---|---|---|---|---|---|
| Ridge | **1.8563** | [1.3858, 2.3268] | 1.9080 | 116.6527 | 0.0601 | PowerTransformer | alpha: 1092.7314 |
| Lasso | 1.8566 | [1.3693, 2.3439] | 2.0632 | 98.2541 | 0.0690 | PowerTransformer | alpha: 0.2284 |
| ElasticNet | 1.8774 | [1.3876, 2.3672] | 2.0948 | 107.0948 | 0.0524 | StandardScaler | alpha: 1.6268, l1_ratio: 0.1784 |
| KernelRidge | 1.8573 | [1.3863, 2.3282] | 1.9091 | 286.3389 | 0.0705 | PowerTransformer | alpha: 1.0312, gamma: 0.0004, kernel: 'rbf' |
| SVR | 1.8642 | [1.378, 2.3504] | 1.8459 | 136.4203 | 0.0604 | PowerTransformer | C: 0.402, epsilon: 1.3529, gamma: 0.0141, kernel: 'poly' |
| KNeighborsRegressor | 1.9182 | [1.4633, 2.3732] | 1.8731 | 151.0082 | 0.0605 | PowerTransformer | n_neighbors: 24, p: 2, weights: 'uniform' |
| MLPRegressor | 1.8629 | [1.4051, 2.3206] | 1.9897 | 3217.7920 | 0.0602 | PowerTransformer | alpha: 118.8002, hidden_layer_sizes: (200, 100) |
| DecisionTreeRegressor | 1.9400 | [1.4453, 2.4348] | 1.8821 | 3.1265 | 0.0000 | None | max_depth: 2, min_samples_leaf: 18 |
| RandomForestRegressor | 1.9191 | [1.4441, 2.3940] | 0.4015 | 203.4383 | 0.0101 | None | max_features: 'log2', n_estimators: 127 |
| AdaBoostRegressor | 2.1857 | [1.7637, 2.6078] | 1.2488 | 166.9085 | 0.0000 | None | estimator: None, n_estimators: 45 |
| XGBRegressor | 2.0184 | [1.5325, 2.5042] | 0.9110 | 273.5760 | 0.0000 | None | learning_rate: 0.0369, max_depth: 3, min_split_loss: 1.1536, n_estimators: 160, reg_alpha: 1.1837, reg_lambda: 0.6507 |
| LGBMRegressor | 1.9206 | [1.4697, 2.3715] | 1.2444 | 292.9824 | 0.0000 | None | learning_rate: 0.0065, num_leaves: 30, max_bin: 361, feature_fraction_bynode: 1, n_estimators: 133, reg_alpha: 0.3209, reg_lambda: 0.3676 |
| CatBoostRegressor | 1.9023 | [1.4077, 2.3968] | 1.4699 | 614.7619 | 0.0100 | None | learning_rate: 0.0255, depth: 5, l2_leaf_reg: 146.7184, leaf_estimation_iterations: 5, n_estimators: 177, rsm: $\sqrt{p}$ |

Table C.8: Regression pipeline outcomes for predicting *next_burn_time_other* where the main pool is USDC-WETH-0.003. 95% confidence intervals are Wald intervals and the upper limit is colored in comparison to the baseline test MSLE of 3.3967. $p = 46$ is the number of features.

| Regressor | MSLE | 95% C.I. | T/V MSLE | Search (s) | Prediction (s) | Transformation | Parameters |
|---|---|---|---|---|---|---|---|
| Ridge | 1.9573 | [1.4918, 2.4229] | 1.9966 | 115.3840 | 0.0616 | PowerTransformer | alpha: 2328.8328 |
| Lasso | 1.9913 | [1.5209, 2.4616] | 2.0598 | 100.7038 | 0.0705 | StandardScaler | alpha: 0.1961 |
| ElasticNet | 2.0011 | [1.5359, 2.4664] | 2.0854 | 102.8288 | 0.0503 | StandardScaler | alpha: 0.3887, l1_ratio: 0.5806 |
| KernelRidge | 2.0081 | [1.5435, 2.4728] | 2.0760 | 308.5453 | 0.0581 | StandardScaler | alpha: 0.4295, gamma: 1.3512e-05, kernel: 'poly' |
| SVR | 1.9311 | [1.4394, 2.4228] | 2.0029 | 155.9111 | 0.0601 | StandardScaler | C: 0.8996, epsilon: 0.3848, gamma: 0.0004, kernel: 'rbf' |
| KNeighborsRegressor | **1.8429** | [1.3480, 2.3377] | 0.0000 | 141.1449 | 0.0615 | StandardScaler | n_neighbors: 18, p: 1, weights: 'distance' |
| MLPRegressor | 2.0058 | [1.5473, 2.4644] | 2.0245 | 3211.5439 | 0.0609 | PowerTransformer | alpha: 118.8002, hidden_layer_sizes: (200, 100) |
| DecisionTreeRegressor | 2.3753 | [1.8133, 2.9373] | 1.7230 | 3.1071 | 0.0020 | None | max_depth: 3, min_samples_leaf: 3 |
| RandomForestRegressor | 2.0148 | [1.5439, 2.4856] | 0.4264 | 203.9196 | 0.0223 | None | max_features: 'log2', n_estimators: 216 |
| AdaBoostRegressor | 2.2850 | [1.6666, 2.9033] | 1.2435 | 167.6585 | 0.0067 | None | estimator: None, n_estimators: 52 |
| XGBRegressor | 1.9944 | [1.5299, 2.4589] | 1.8416 | 271.6293 | 0.0029 | None | learning_rate: 0.0267, max_depth: 1, min_split_loss: 1.5659, n_estimators: 184, reg_alpha: 3.3386, reg_lambda: 4.6228 |
| LGBMRegressor | 1.9893 | [1.5051, 2.4735] | 1.7217 | 293.2550 | 0.0016 | None | learning_rate: 0.0259, num_leaves: 20, max_bin: 269, feature_fraction_bynode: $\log_2(p)$, n_estimators: 165, reg_alpha: 27.0342, reg_lambda: 9.2604 |
| CatBoostRegressor | 1.9736 | [1.4906, 2.4565] | 1.8425 | 613.6212 | 0.0040 | None | learning_rate: 0.0735, depth: 2, l2_leaf_reg: 456.8755, leaf_estimation_iterations: 1, n_estimators: 173, rsm: $\sqrt{p}$ |