Workshop Fundamentals of Data Science – Group 9

Predicting the Priority Fee Needed to Reach a Target Block Position on Solana

Antonin Ricard Boual, Killian Borgeaud, Pedro Negrao, Ali Haidar

1 Context and Definitions

Solana

Solana is a high-performance blockchain on which each transaction pays a fee. This fee consists of two components: a fixed base fee and a user-specified priority fee (which functions similarly to a voluntary "tip").

Accounts

On Solana, the term "account" doesn't just mean a user's wallet, it refers to any on-chain structure that stores information and can be used by different users. In our project we deal with a specific type of account called a "liquidity pool". This can be shortly summarized as a small automated market in which users can switch between two assets (e.g. SOL–USDT) according to an algorithm.

Blocks and Priority Fees

Blocks on Solana are short batches of transactions, produced approximately every 0.5 seconds. When multiple transactions within the same block attempt to access the same account, they cannot be executed in parallel; instead, they must be ordered. Earlier transactions may receive better execution outcomes, for example a better price. This ordering is determined by priority fees: higher priority fees lead to earlier execution within that specific account context. Importantly, this ordering is *local* to each account. Transactions that touch different accounts do not compete for ordering and may be executed in parallel within the same block without conflict.

2 Goal and Scope

Our goal is to predict the actual priority fee paid by the transaction that lands at the 0.9-quantile position of the current block's ordering for a specific account.

In other words, the output of the model will be the predicted average fee y required to achieve the target 0.9-quantile placement, given the current market context X. For this assignment, we will focus on the account *ORCA: SOL-USDC* as it is the most liquid one (ORCA being a decentralized exchange). We will not model base fees, because they are fixed and can be determined with certainty via a simple calculation (they do not contain predictive value).

With this setup in mind, it's useful to see why the problem matters in practice. On Solana, several pools can exist for the same token pair. For example: (*Raydium: SOL-USDT*, *Orca: SOL-USDT*, *Raydium: SOL-USDC*, *HUMIDIFI: SOL-USDC*). This creates short-term, cross-market arbitrage opportunities. Small price differences appear constantly and disappear very quickly.

Because expected arbitrage profits are extremely thin, the fee decision becomes critical: If you pay too little priority fee, other actors might capture the price difference first; if you pay too much, the opportunity becomes unprofitable. A predictive machine learning model could possibly capture opportunities that a fixed-fee strategy would systematically miss.

3 Data Acquisition, Cleaning

For this project, we sourced our data directly from Snowflake. We could use as many blocks as desired. We decided to extract 100,000 blocks, giving us a good trade-off between model performance and computational power; this is equivalent to around 13.9 hours. We also provide the full CSV containing the dataset used in our models.

We rely on Snowflake as our cloud data platform and obtain the required variables through a dedicated SQL query. The table below summarizes the original data sources for each feature and the transformations applied in SQL. The query collects the relevant fields from each source, builds an intermediate table for every feature, and finally merges all intermediate tables into a single dataset that is exported as a CSV or used directly for our models.

Data	**Primary Sources**
The Block ID code	*SOLANA_ONCHAIN_CORE_DATA.CORE.FACT_TRANSACTIONS* This is a Snowflake database containing Solana blockchain data. We extract the defined BLOCK IDs from it.
The total fees and priority fees	*SOLANA_ONCHAIN_CORE_DATA.CORE.FACT_TRANSACTIONS* Total fee paid for the transaction in lamports (1 SOL = 1 billion lamports). We calculate the priority fee by doing fee – base fee of 5000 lamports. The priority fees we want to predict are for the ORCA liquidity pool.
SOL/(USDC, USDT) exchange rates from 6 different liquidity pools	*SOLANA_ONCHAIN_CORE_DATA.CORE.FACT_TOKEN_BALANCES* This data is sourced from a Snowflake database. We use the pool address code to identify different liquidity pools. Exchange prices are then derived by measuring the change in SOL and USDC/USDT balances before and after each transaction associated with that pool.
Volume per Block, Average over the last 999 blocks	*SOLANA_ONCHAIN_CORE_DATA.CORE.FACT_TOKEN_BALANCES* Total USDC and USDT volume across 15 different pools, using the balance before and after, and adding everything up for each block. Or using the average over 999 blocks.
Historical features	*SOLANA_ONCHAIN_CORE_DATA.CORE.FACT_TRANSACTIONS* We calculate the average and total priority fees paid with a lag of up to 10 blocks.
Seasonality	*SOLANA_ONCHAIN_CORE_DATA.CORE.FACT_TRANSACTIONS* Similar to historical features, we calculate the priority fees for every 30 min, using the block timestamps.

4 Feature Selection

The primary focus of the model is to predict the priority fee paid for the transaction that lands at the 0.9-quantile position of the liquidity pool; this is our target value.

To construct the design matrix, we begin with the CSV file generated from the SQL extraction. The code first standardizes column names and then aggregates the data at the block level. For each block, we compute the 0.9-quantile of the priority fees using linear interpolation. This produces a single target value per block.

All other features are also reduced to block-level values. Since they do not vary within a block, we simply take the first occurrence of each feature in that block. These features are described in the table below, along with the motivation for their inclusions. The result of this process is a dataset where each row corresponds to one block.

Features used in our design matrix	**Reasoning for their inclusion and calculation**
volume, avg_volume_last_999	Higher volume generally reflects greater competition for block space, which increases priority fees. Volume over 999 blocks makes it possible to capture longer-term patterns in changes in volume.

Features used in our design matrix	**Reasoning for their inclusion and calculation**
avg_balance	This is the average USDC balance in the ORCA liquidity pool we're interested in. This feature is important because it is an indicator of liquidity depth.
tx_count_in_block	This gives us an indicator of the number of transactions in the block before. Provides a measure of short-term network congestion.
total_fee	Sum of all priority fees in the previous block; this may indicate a strong incentive to submit a higher priority fee in the current block, for example due to arbitrage opportunities or increased volume.
Units_consumed	Total computational units used by all transactions in the block. This is a similar proxy for the congestion of transactions.
avg_priority_fee	This comes from our calculation of seasonality, the average of the priority fees paid for a period of 30 min. This variable helps us determine the seasonality and movements throughout the day that our model could rely upon.
fksff_price	This is the exchange price of the ORCA liquidity pool for the previous block. Could give us an indication of potential arbitrage possibilities which would mean higher priority fees.
pct_diff between czfq3 and other pools	These are the percentage differences between the prices of the studied liquidity pool (ORCA) and other pools. This is used as an indicator for arbitrage that could potentially take place.
sum_block_minus_n	Total priority fees paid n blocks ago goes up to 10 lagged blocks; could capture a short-term trend in priority fees or a large uptake.
avg_block_minus_n	Average priority fees paid n blocks ago goes up to 10 lagged blocks; could capture a short-term trend in priority fees or a large uptake.
block_trend	How many blocks from the first block we are. If there is an underlying trend this variable will capture it.
quantile_fee_lag up to 4	These are the lagged features being the actual 0.9 quantile priority fees paid. The autocorrelation function graph below indicates that autocorrelation disappears after roughly 3–4 lags, meaning that values beyond this point provide no additional predictive signal. Therefore, we restrict the model to 4 lagged features.

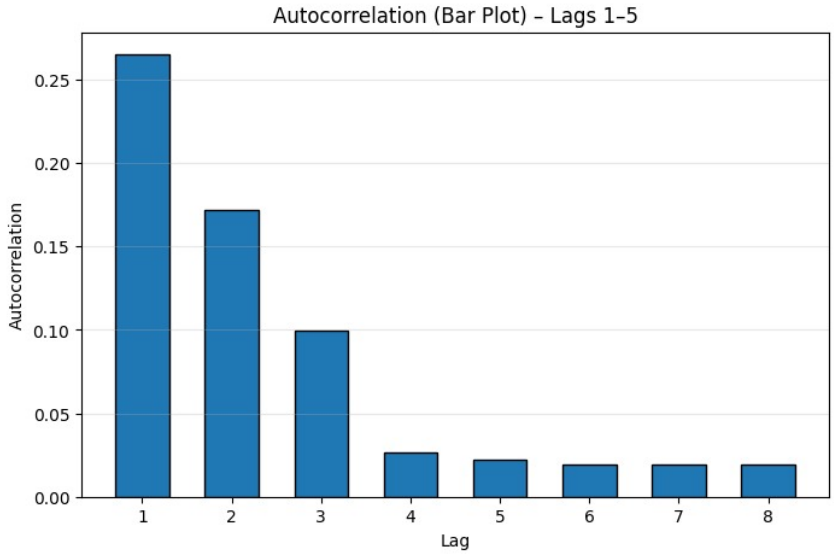


Figure 1: Autocorrelation for 0.9-Quantile priority fee up to lag of 8

5 Feature Engineering

We first clean our feature matrix by filling missing values with the value 0. After verifying the dataset, missing values are in reality values that are equal to 0. We do the same for our target column; missing values (NAs) are just priority fees of value 0.

The 0.9-quantile priority fee and the feature values occasionally take on extremely large values due to bursts of high volume, arbitrage races, or other reasons (users decide on the tip they want). These outliers represent a very small portion of the dataset but can disproportionately distort the model. To prevent this, for the target we apply minimal clipping using a relaxed Turkey upper fence ($Q_3 + 20 \cdot \text{IQR}$); any value above the threshold is brought down to the upper bound. For the features we calculate the 1st and 99th percentiles of each feature using the training data. These percentiles define lower and upper clipping bounds. Both the training and test features are then clipped to this range. Below we see the evolution of the MAE depending on the k multiplier of IQR.

We explored the distribution of every feature and the target variable. For each column, we inspected the mean, standard deviation, range, interquartile range, and especially its skewness. This allowed us to identify variables with extremely asymmetric, heavy-tailed distributions. Any feature with skewness above 4 and non-negative values was classified as a skewed feature, while all remaining variables were grouped as other features. Because these two groups behave differently, we decided to apply different transformations to each:

MAE	Linear Reg	Ridge	Random Forest	MLP Regressor	ARIMA	"Naive" Mean
K = 1,5	1169.12	1168.64	1161.81	1161.81	1222.93	2626.53
K = 3	1452.79	1452.31	1445.48	1445.48	1506.59	2910.20
K = 5	1761.40	1760.92	1754.09	1754.09	1815.20	3218.81
K = 10	2413.75	2413.28	2406.45	2406.45	2467.56	3871.17
K = 20	3482.71	3482.23	**3475.40**	3492.62	3536.51	4940.12
K = 50	5865.78	5811.50	5804.67	5821.89	5865.78	7400.93
No Fence	14990.71	14990.24	14983.41	15000.63	15044.52	16579.67

Figure 2: MAE across different models depending on k multiplier

Skewed features: We applied a two-step transformation: Log transform (log1p) applies the logarithm function +1 to avoid problems with zero values; it compresses larger values more than smaller ones. We then applied the RobustScaler, which rescales each feature by subtracting its median and dividing by its interquartile range (IQR). Because it ignores extreme minimum and maximum values, it produces a transformation that is much less sensitive to outliers than standard scaling methods.

Other features: We applied a PowerTransformer (Yeo–Johnson) to this group. This method can handle both positive and negative values. It makes distributions more symmetric and closer to Gaussian, which improves performance for many model types.

We also tested alternative preprocessing pipelines for comparison; these will be explained in the model selection section:

- PowerTransformer applied to all features at once: A uniform transformation approach to test whether treating all variables identically performs better.
- StandardScaler on all features: Standardization of the features
- PolynomialFeatures (degree 2): This expands the original feature set by generating all polynomial combinations of the input features up to degree 2. It creates new, more expressive features that allow models to capture nonlinear relationships between the predictors and the target.

Because the target (the 0.9-quantile priority fee) is also right-skewed, we created a transformed version using a log1p transform on y; we use np.expm1 afterwards to revert when evaluating the model.

Our data follows a time-series structure and includes lagged features, so the train–test split is performed chronologically at an 80% to 20% ratio based on the index. This approach ensures the model is trained only on past observations and includes enough test data to reduce the risk of overfitting.

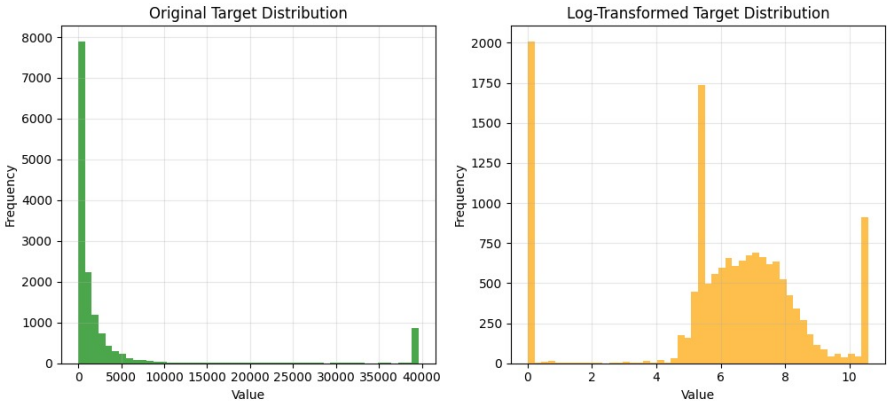


Figure 3: Distribution of target values before and after log transformation

6 Model Selection

6.1 Naive Model

A naive model is a simple, non-informative baseline predictor. Any machine learning model should outperform this minimum standard to demonstrate that it offers predictive utility. Our naive model predicts the mean of the training targets every time.

6.2 Linear Regression

We first apply a simple linear regression; this serves as a benchmark to evaluate other models later. To enhance predictive modelling, Ridge and ElasticNet regression models were also implemented. Both approaches address multicollinearity and regularize coefficients to prevent overfitting. For the models here we tried a multitude of different preprocessing techniques to compare the results; a table summarizing these can be found in the following section.

The Ridge and ElasticNet regression models were trained using cross-validation. For hyperparameter tuning, we used a function that creates 50 candidate values for the regularization parameter alpha. RidgeCV and ElasticNetCV with built-in cross-validation will automatically select the best alpha from the candidate values.

The Ridge regression model minimizes the residual sum of squares and adds a penalty on the squared magnitude of coefficients. The ElasticNet model minimizes the residual sum of squares while adding a combined penalty that blends the L1 (absolute value) and L2 (squared magnitude) of the coefficients.

6.3 Random Forest

The random forest model makes predictions by aggregating the outputs of many decision trees, each trained on different bootstrap samples and feature subsets to reduce variance and improve generalization. We defined a predefined number of estimators of 100, meaning a good tradeoff

between computation time and model performance. We use the standardized train features and the log-transformed target values for this model.

6.4 Neural Network

For the neural network model, we used scikit-learn's MLPRegressor and performed hyperparameter tuning through RandomizedSearchCV. We defined a parameter distribution covering different hidden-layer structures, multiple alpha values on a log scale, and several learning-rate settings. A base MLP with ReLU activation, the Adam optimizer, early stopping, and a fixed validation fraction served as the foundation for the search. The randomized search evaluated parameter combinations using time-series cross-validation, ensuring the temporal ordering of the data was respected. After identifying the best configuration, we created a new MLPRegressor with these optimal parameters and trained it on the preprocessed training data. Predictions were generated in log space and converted back to the original scale to compute the final error metrics on the test set.

7 Model Evaluation and Results

7.1 Evaluation

We evaluate our models with the mean average error and the root mean squared error using the functions directly provided in the sklearn.metrics package.

The target column (0.9 quantile priority fees) has following descriptive statistics after clipping the values:

Min	Max	Mean	Standard deviation	Median
0.00	39633.10	3874.37	9569.17	645.00

7.2 Naive Model

The naïve mean model provides a simple baseline on which to compare the other models. The model simply always predicts the mean of the training data. It has an MAE of 5072.09 and an RMSE of 9230.69. Both lie well above the central tendency of the target distribution. Given that the mean clipped priority fee is 3,874 and the median is 645, the naïve model's MAE of 5,072 and RMSE of 9,230 show that errors routinely exceed the scale of most real observations, reflecting how strongly outliers dominate.

7.3 Linear Regression

Different preprocessing techniques were tested using the simple linear regression to compare their effects; we list them in the table below.

Model and Preprocessing	MAE	RMSE
Linear Regression with Yeo Johnson applied to all features	4260.89	9296.57
Linear Regression with X clipping & skewed features log + Robust Scaler & other features Yeo Johnson	4132.25	9326.24
Linear Regression with X clipping	4003.43	9370.87
Linear Regression with Standard Scaler	4003.43	9370.87
Linear Regression with X clipping & skewed features log	3968.00	9375.85
Linear Regression with X clipping & skewed log + Robust Scaler	3968.00	9375.85
Linear Regression without X clipping	3775.32	9485.16

Model and Preprocessing	MAE	RMSE
Linear Regression with X clipping and y log	3524.48	9826.88
Linear Regression with X clipping & skewed features log + Robust Scaler & other features Yeo Johnson and y log	3483.45	9782.24

The RMSE of the simple linear regression is higher than that of the naïve model because RMSE penalizes large errors more heavily, and our dataset contains a substantial number of extreme priority-fee values. Even small mispredictions on these rare but very large observations significantly inflate the RMSE. In terms of the MAE, which we believe to be a more accurate depiction of performance given the nature of our data, the simple linear regression performed better than the naive model whatever the preprocessing; this shows that the model has some predictive value. The preprocessing findings are interesting: by far the biggest improvement in MAE stems from the log transformation of the target variables, showing that extreme values negatively impact the prediction and when corrected for, the predictive ability is better overall. The model performed best when the maximum number of preprocessing steps were applied. For all models from here on this was the standard preprocessing we applied: X clipping, skewed features are log transformed and we apply Robust Scaler- For the other features we apply Yeo Johnson, and for the target y a log transformation.

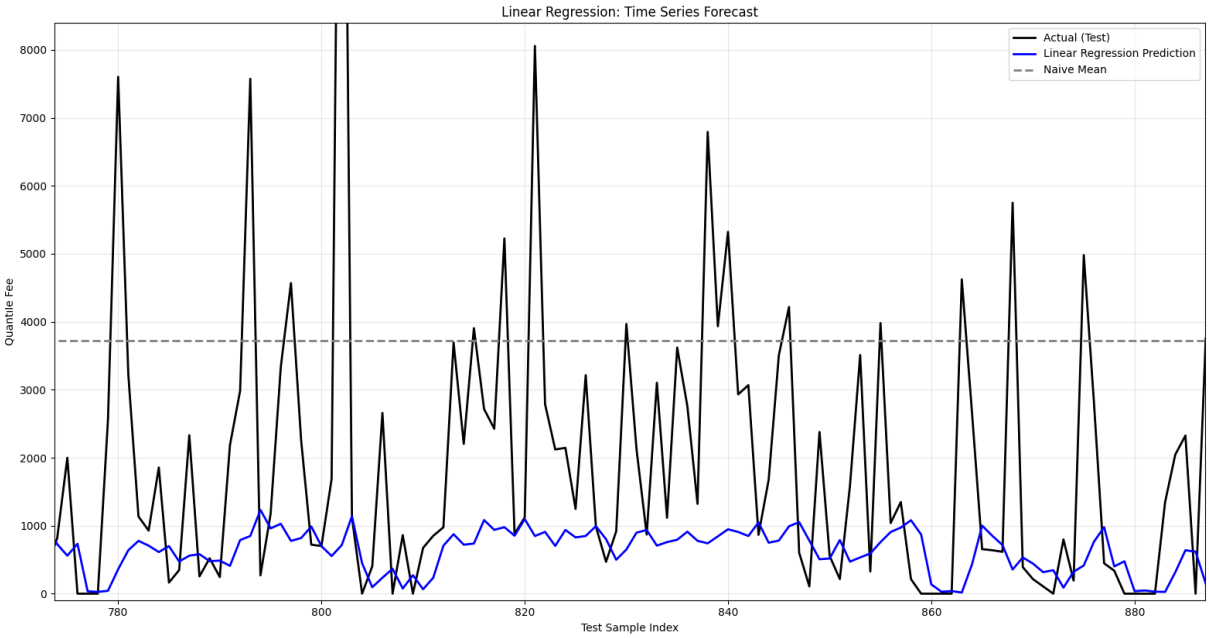


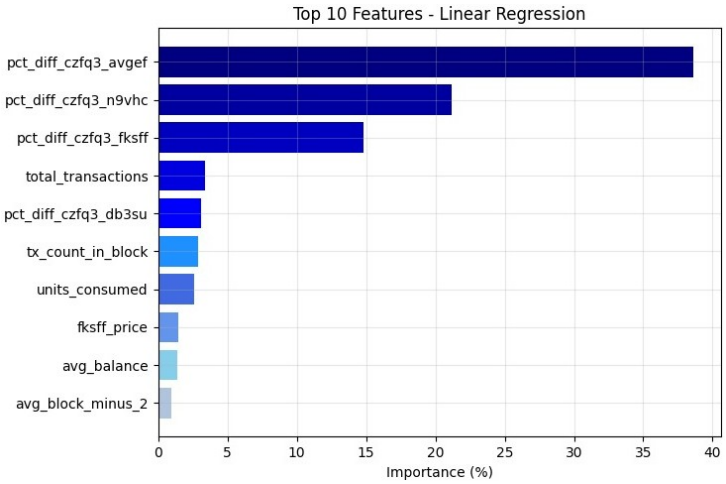
Figure 4: Plotted actual and predicted priority fees over a test sample

We also tested the Ridge and ElasticNet with hyperparameter tuning with cross-validation:

Model and Preprocessing	MAE	RMSE
Ridge Regression with standard preprocessing	3483.04	9782.81
Ridge Regression with Polynomial Features	3472.60	9732.14
Elastic Net Regressor with Polynomial Features	3472.72	9762.18

Ridge and ElasticNet regressors had no major differences in comparison to the simple linear regression model; polynomial features had a negligible impact and did not capture any meaningful non-linear relationships that could help with prediction. The MAE was lower than for the naive model.

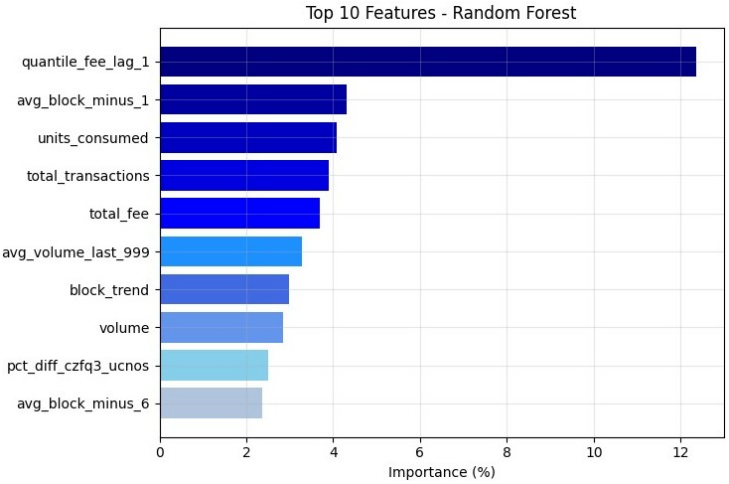
For the feature importance, the most significant ones are the price differences with the other liquidity pools; this could signify that our theory that price differences lead to arbitrage opportunities which are seized upon by putting up a higher or lower priority fee. However, we will see later on that other models do not use these features with the same importance.



7.4 Random Forest

The random forest was trained using the standard scaled features and log-transformed target y. These are the preprocessing steps that delivered the best empirical results. It has an MAE of 3520.65 and an RMSE of 9828.80. The random forest performed generally well, however slightly worse than the simple linear regression.

Being able to evaluate the importance of features is a great benefit of random forests as it allows for more transparency and comprehensibility. When looking at the feature importances, we find that the most significant feature is the quantile fee just one block before; this aligns with the findings of the autocorrelation function above. Sudden fee rises or dips often last for 2–3 blocks, so it would make sense for the model to make use of this information. Otherwise, other significant features are all directly measures or proxies of the volume, showing that the congestion can also predict higher priority fees.



7.5 Neural Network

We found the neural network performed well with an RMSE of 9772.85 and an MAE of 3474.64, comparable to the simple linear regression model. The neural network had a learning rate of 0.0003 and hidden layer sizes of (32, 64).

8 Results Table

The table below summarizes our model performance. All models outperform the naïve mean baselines. Additonally although the MAE appears large relative to the average fee level, this is mainly driven by a small number of extreme spikes in the target variable. Most blocks have moderate fees, while a few have extraordinarily high values. The MAE is also good for other reasons than the predictive performance.

The average percentage error below the true fee is dramatically smaller than the average percentage error above it. On the other hand, the actual values are above the predictive values only slightly more times than they are below. Our MAE is a desirable outcome given our setting. Overbidding, predicting a fee that is unnecessarily high is very risky as it could still be

lost to another bidder or it would represent too high of a cost and the position could have been gained with a much smaller fee.

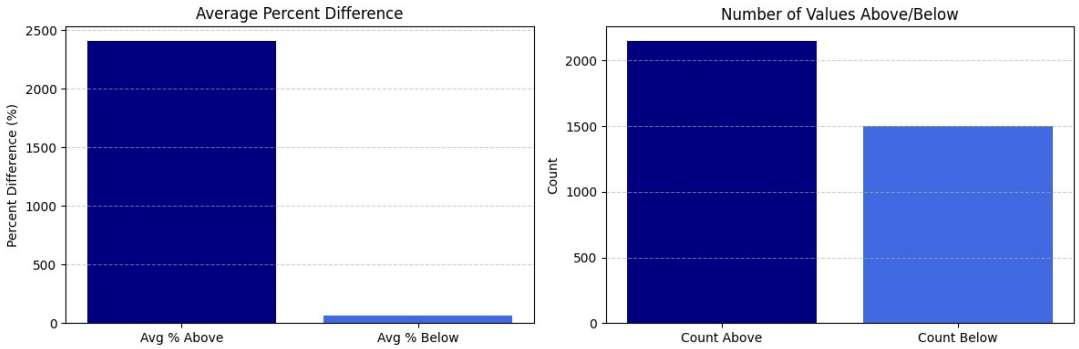


Figure 5: Percent differences between predicted and actual values and Number of values above/below predicted

In contrast, slightly underbidding is often harmless: in a large amount of cases the required fee is modest, and the percentage difference is small and the absolute value when we lose to higher bidders is small.

While not linked to the predictive ability of the model and perhaps delving into prescriptive analytics, the model uses a conservative strategy. It stays near typical values, avoids overpaying, and accepts that occasionally missing extreme, unprofitable spikes is the economically sensible trade-off. This pattern aligns with our setting where limiting losses from overpaying matters more than capturing every possible opportunity.

Model	**Neural Net**	**Random Forest**	**Linear Reg**	**Ridge Reg**	**ElasticNet**	**Naive**
MAE	3474.64	3520.65	3483.45	3472.60	3472.72	5072.09
RMSE	9772.85	9828.80	9782.24	9732.14	9762.18	9230.69