CS6140 Machine Learning Fall 2023 (Seattle)

Final Project Report

Instructor: Dr Ning Rui

Team B members: Kaijun, Gao, ShiangJin, Chin, Yishuang, Chen

Table of Contents

1. Introduction & Overview

1.1 Why did our team choose this topic?

In the dynamic and complex realm of financial markets, accurate predictions of stock prices are crucial for a range of stakeholders, from individual investors to large financial institutions. The motivation for this study stems from the ongoing quest for more reliable and precise forecasting tools in the volatile domain of stock trading. Enhanced predictive accuracy can significantly contribute to risk management, investment strategy formulation, and overall market stability. This research is particularly timely and relevant in the context of the "Optiver - Trading at the Close" Kaggle competition, which emphasizes the importance of predicting stock prices accurately at market close (Forbes et al., 2023).

In this competition, the participants are challenged to develop a machine-learning model capable of predicting the relative closing price movements in the last 10 minutes for hundreds of Nasdaq-listed stocks based on data from the order book and the closing auction (Forbes et al., 2023). The model can contribute to the research in the consolidation of signals from both sources, leading to improved market efficiency and accessibility (Forbes et al., 2023). We picked this topic as we can get firsthand experience in handling real-world data science problems faced by the traders, quantitative researchers, and engineers in the industry while getting to learn in-depth about two of the most popular machine learning models (LightGBM and XGBoost) used in Kaggle competitions.

This paper presents an in-depth investigation into the efficacy of two advanced predictive models used for regression analysis and forecast of the stock price: LightGBM and XGBoost. A benchmark run with the traditional linear regression model is conducted for comparison. By comparing the performance of these models after different preprocessing techniques, feature engineering effort, and hyperparameter tuning, this paper aims to provide insights and practical recommendations for market participants and algorithmic traders. The findings are expected to contribute valuable perspectives to the field of financial forecasting and support the development of more efficient trading strategies.

1.2 Prior Research and Literature Review

As the main goal is trying to predict the future movement of stock price, the Regression Model is our pick for this problem. One of the prior research analyzes the performance of different regression models in predicting the closing price of stocks belonging to the S&P500 index. The research found that the Decision Tree Regression outperforms the traditional simple linear regression model (S. Ravikumar, 2020). Hence for this project, we decided to try out LightGBM and XGBoost which can act as Decision Tree Regression models.

LightGBM (Light Gradient Boosting Machine) is a distributed high-performance framework that uses decision trees for ranking, classification, and regression tasks (Saha, 2023). It is a free, open-source, distributed gradient-boosting framework for machine learning, originally developed by Microsoft (Guolin, 2017). In contrast to the level-wise(horizontal) growth in

XGBoost, LightGBM carries out leaf-wise (vertical) growth that results in more loss reduction, and higher accuracy while being faster (Saha, 2023). However, this approach may lead to the overfitting of the training data (Saha, 2023).

XGBoost (Extreme Gradient Boosting) is an optimized distributed gradient boosting library designed to be highly efficient, flexible, and portable, and provides a parallel tree boosting (also known as GBDT, GBM) that solves many data science problems in a fast and accurate way (Chen, 2016).

1.3 Information About the Dataset Used

1.3.1 Description of data

The dataset is available for download from the Kaggle website (https://www.kaggle.com/competitions/optiver-trading-at-the-close/data). The available features in the data set are as follows:

- stock_id: Identifier for the stock.
- date_id: Identifier for the date.
- seconds_in_bucket: Time in seconds, starting from the last 10 minutes towards the closing. 0 seconds means 10 minutes left until the closing.
- imbalance_size Size of imbalance.
- imbalance_buy_sell_flag: Flag indicating buy or sell imbalance.
- reference_price: Reference price.
- matched_size: Size of matched orders.
- far_price: Far price.
- near_price: Near price.
- bid_price: Bid price.
- bid_size: Size of the bid.
- ask_price: Ask price.
- ask_size: Size of the ask.
- wap: The weighted average price in the non-auction book. Equal to bid_price * ask_size + ask_price * bid_size / (bid_size + ask_size)
- time_id: Identifier for the time.
- row_id: Unique row identifier.
- currently_scored: Boolean flag indicating if currently scored. This feature is only available in the testing set during submission to the competition.
- target: relative movement of the wap price of a particular stock to the index in the next 60 seconds. defined using the following formula (Forbes et al., 2023):

$$Target = \left(\frac{StockWAP_{t+60}}{StockWAP_t} - \frac{IndexWAP_{t+60}}{IndexWAP_t}\right) * 10000$$

The total number of unique stock_id counts is 200. The total number of unique date_id counts is 481. There are 54 seconds_in_bucket available for each unique stock_id and date_id combination. Thus the total number of rows (samples) available is the number of unique stock * number of unique dates * number of unique seconds in bucket ≈ 200 * 480 * 54 ≈ 5, 237, 980 (5.2 million).

1.3.2 Correlation Analysis of the Data

Pearson correlation analysis and Spearman correlation analysis were performed between the dependent variable (target) and other variables. Their results are sorted by absolute value and presented below. In summary, all the variables seemed to have **weak correlations** with the target.

Sorted Pearson Correlation (sorted by absolute value):

- wap: -0.056194051943220885
- ask_price: -0.04838886498913048
- bid_price: -0.04775542189749235
- reference_price: -0.04473762032423138
- bid_size: -0.017739631092372504
- ask_size: 0.012752139801386852
- imbalance_size: 0.0010597508079042894
- matched_size: 0.0006216808516379594
- far_price: -0.0018037281170692225
- near_price: -0.002115035448593967

Sorted Spearman Correlation (sorted by absolute value):

- bid_size: -0.05963073825519416
- wap: - 0.04282104755286715
- ask_price: -0.03501851042029415
- bid_price: -0.03414298133949177
- reference_price: -0.03203920862358736
- near_price: -0.007609943577612957
- far_price: -0.005374061289319017
- ask_size: 0.057489574839903836
- matched_size: 0.0017232624948999111
- imbalance_size: 0.0010833034541324852

A regression analysis using OLS was carried out against some of the features above. Feature stock_id was omitted as treating it as a category variable will cause the OLS to take too long to run. Feature near_price and far_price were omitted as these two columns contain more than half of the values as NaN.

Based on the results as shown in **Figure 1**, it seems all variables have statistical significance with the target (with P>|t| value of 0), but the portion of variance that can be explained by these variables is only about 0.024 (2.4%). This indicates that while the regression model can still be used to predict the target, it might have a larger error rate inherently.

```
Regression Result for Target
                            OLS Regression Results
==============================================================================
Dep. Variable:                 target   R-squared:                       0.024
Model:                            OLS   Adj. R-squared:                  0.024
Method:                 Least Squares   F-statistic:                 1.156e+04
Date:                Sat, 16 Dec 2023   Prob (F-statistic):               0.00
Time:                        18:58:42   Log-Likelihood:            -1.9135e+07
No. Observations:             5237760   AIC:                         3.827e+07
Df Residuals:                 5237748   BIC:                         3.827e+07
Df Model:                          11
Covariance Type:            nonrobust
=================================================================================================
                                    coef    std err          t      P>|t|      [0.025      0.975]
-------------------------------------------------------------------------------------------------
Intercept                        99.1965      1.655     59.935      0.000      95.953     102.440
C(imbalance_buy_sell_flag)[T.0]   0.0102      0.011      0.898      0.369      -0.012       0.032
C(imbalance_buy_sell_flag)[T.1]   0.0806      0.010      7.976      0.000       0.061       0.100
seconds_in_bucket                -0.0002   2.74e-05     -6.081      0.000      -0.000      -0.000
imbalance_size                -1.281e-09   2.37e-10     -5.405      0.000   -1.74e-09   -8.16e-10
reference_price                1305.8639     13.203     98.903      0.000    1279.986    1331.742
matched_size                  1.295e-10   3.46e-11      3.738      0.000    6.16e-11    1.97e-10
bid_price                      2492.8616     13.116    190.067      0.000    2467.155    2518.568
bid_size                      7.409e-07   3.96e-08     18.734      0.000    6.63e-07    8.18e-07
ask_price                      2404.0694     13.137    182.994      0.000    2378.320    2429.818
ask_size                      -6.225e-07   3.38e-08    -18.423      0.000   -6.89e-07   -5.56e-07
wap                           -6302.0292     18.699   -337.029      0.000   -6338.678   -6265.380
==============================================================================
Omnibus:                  1392897.247   Durbin-Watson:                   1.969
Prob(Omnibus):                  0.000   Jarque-Bera (JB):        119839488.451
Skew:                           0.217   Prob(JB):                         0.00
Kurtosis:                      26.429   Cond. No.                     8.40e+11
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 8.4e+11. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Figure 1: Regression Analysis Result for Target

1.3.3 EDA showing Distributions of Data

The following figures will show the Violin plot for the data points for some of the numeric attributes. Noticed that for attributes related to size, most of the data concentrated on a narrow band near the left (smaller value) with few of the data points located at the right. This is expected as a few companies in Nasdaq (like Google, Facebook, Apple, Amazon, etc.) have large trading volumes compared to the other smaller companies. On the other hand, for attributes related to price, most of the data concentrated on a narrow band centered around 1 as the prices have been converted to a value relative to the base price of the stock.

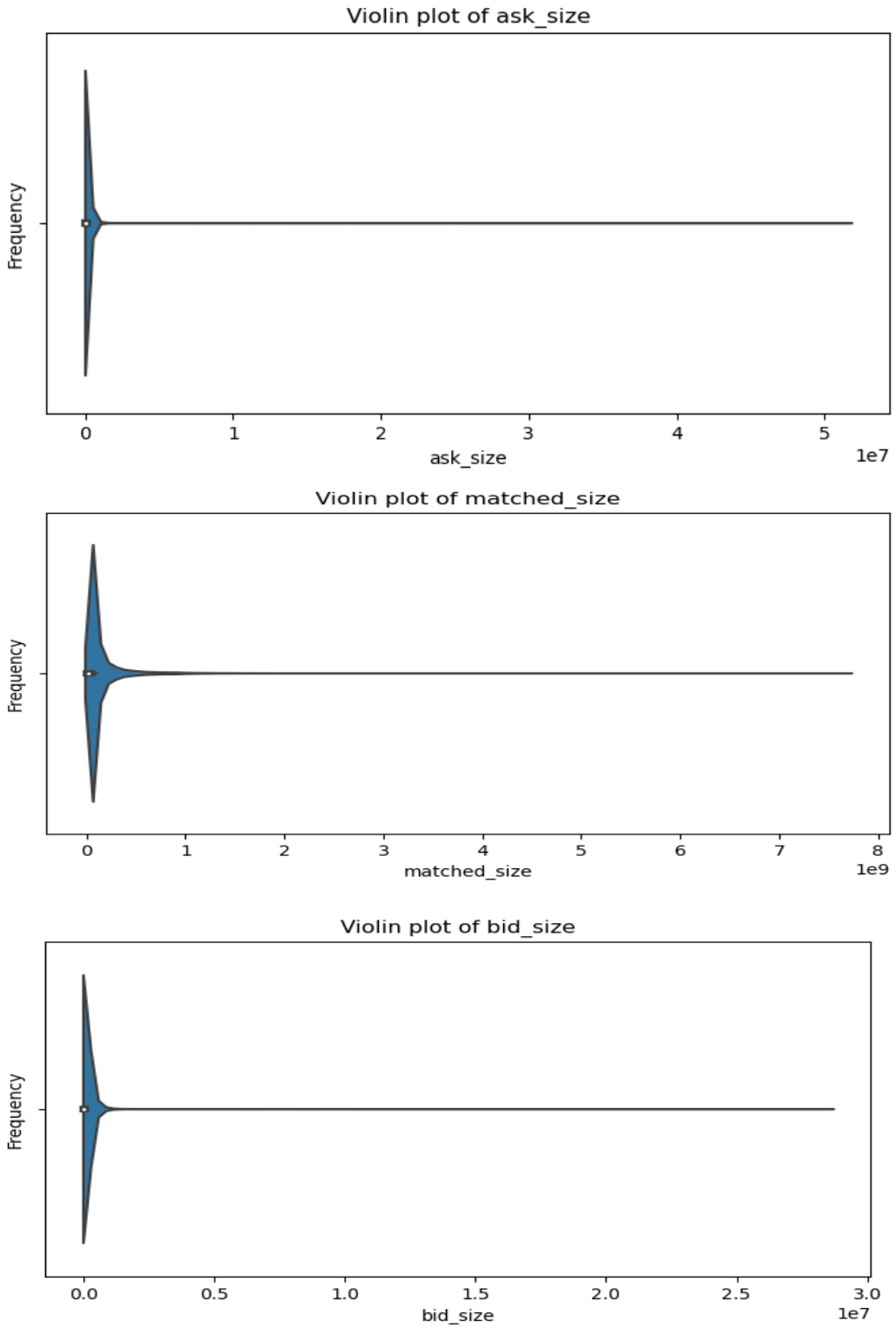

Figure 2: Violin plots for ask size, matched_size and bid_size

Figure 3: Violin plots for imbalance_size

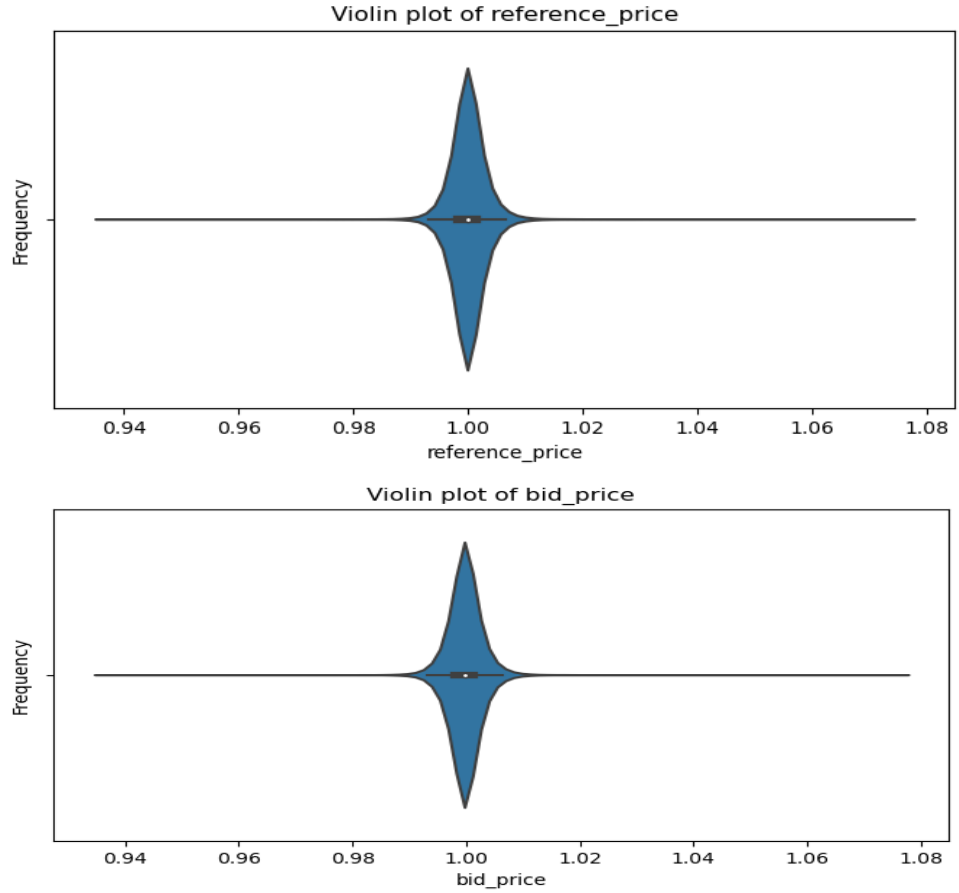

Figure 4: Violin plots for reference_price and bid_price

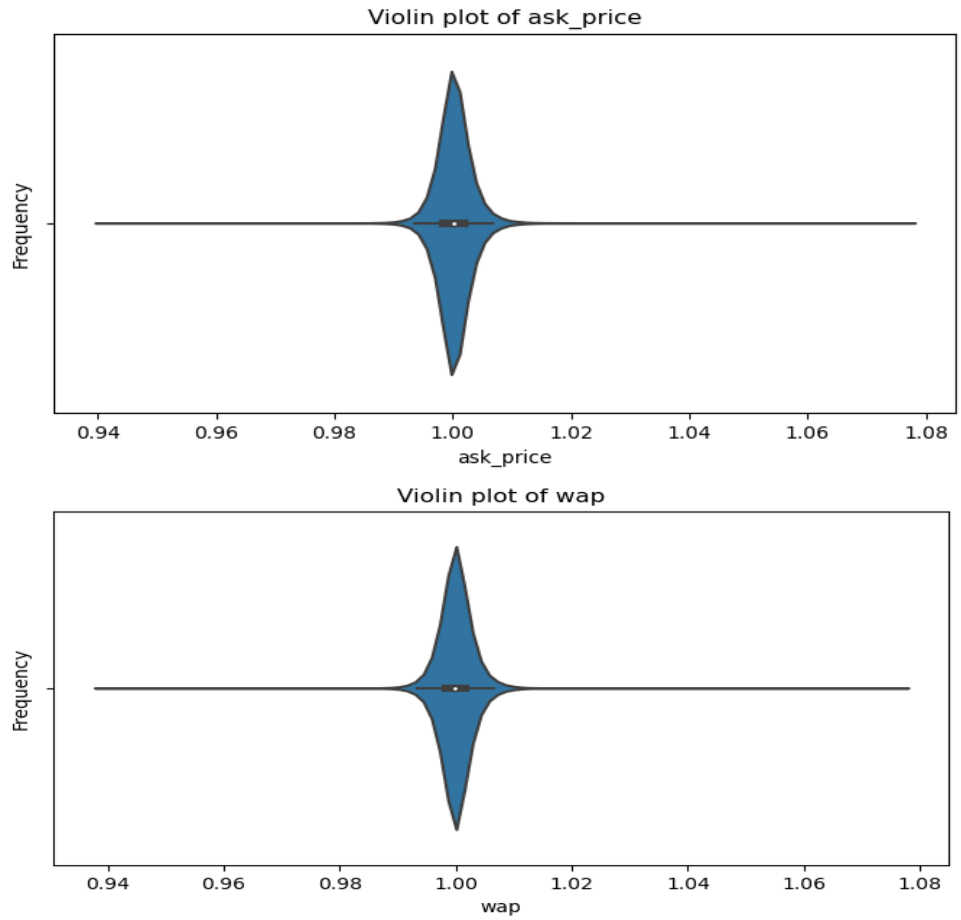

Figure 5: Violin plots for ask_price and wap

2. Our SMART Questions

The main goal is to achieve a Mean Absolute Error (MAE) score between the predicted return and the observed target based on the test set of less than 10. The formula to compute the MAE is given by

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - x_i|$$

Where n is the total number of data points, y_i is the predicted value for data point I, x_i is the observed value for data point i.

The main and ambitious goal is set based on the best MAE score achieved by the top team in the competition, which is around 5.4 when drafting our proposal. As our team only tried this kind of problem for the first time, we think less than 10 will be achievable and less than 6 will be ambitious.

3. Establishing the Benchmark Using Linear Regression Model

A quick run was conducted using the traditional Linear Regression Model to establish the benchmark for all remaining experiments. The dataset is split into the training set and validation set with a 0.8: 0.2 ratio. Out of the features available in section 1.3.1, the features used as the independent variables are stock_id, seconds_in_bucket, imbalance_size, imbalance_buy_sell_flag, reference_price, matched_size, bid_price, bid_size, ask_price, ask_size, wap.

Note that near price and far price were not used as more than half of the rows contain NaN values in these two columns, and the traditional LR model cannot handle NaN values. Dropping all rows with NaN values is not a feasible approach as we still need to predict test data containing NaN values in these two columns. The alternative is to fill the NaN with mean or median values; however, this will pose a challenge as well when we want to handle unseen test data. Hence the final approach we decided is to drop both columns, given the relatively less important of these two columns in the later models.

Also, note time_id and row_id columns were dropped as these two columns carry no significant meaning to the prediction of the target.

As a result, the MAE achieved on the validation data is 6.315.

The next two sections will describe the experiments, results, and learning outcomes of the XGBoost and LightGBM models separately.

4. XGBoost

The experiments and improvement attempt with XGBoost is summarized below:

- Preprocessing and Cross-Validation options:
  - Initial run using all features in the training dataset with Scikit-learn's TimeSeriesSplit for Cross-Validation, with the default setting for XGBRegressor.
  - Testing the effect of applying StandardScaler() on data preprocessing.
  - Evaluating the impact of using standard k-fold cross-validation.
  - Evaluating the impact of applying Principal Component Analysis (PCA)
- Exploring Feature Importances:
  - Feature engineering was not performed extensively on the XGBoost model compared to the LightGBM model as the XGBoost model is unable to handle NaN data. This put a limitation on the type of feature engineering we can perform with XGBoost
- Hyperparameter Tunings of the XGBoost Model:
  - Tuning hyperparameters like eta, gamma, max_depth, min_child_weight, min_delta_step, subsample, colsample_bytre, lambda, and alpha.
  - Using methods like grid search or random search for systematic hyperparameter optimization.

The outcomes of the different preprocessing techniques, feature engineering, and hyperparameter tuning are presented with the accuracy metrics, feature importance, and other relevant insights.

4.1 Preprocessing and Cross-Validation option

4.1.1 XGBoost Model Initial Run Feedback

Model Performance Overview:

The initial trial of the XGBoost model has been conducted using a TimeSeriesSplit cross-validation with 10 folds. The model was set up with a reg: absolute error objective, utilizing a histogram-based tree method optimized for GPU usage. Random states were fixed across the environment to ensure the reproducibility of results. No data scaling or feature selection preprocessing steps were applied before model training:

Model Training and Validation:

During the cross-validation process, the model was trained on each fold, and predictions were made for both training and validation sets. The mean absolute error (MAE) was used to evaluate the model's performance, and the results for each fold were recorded.

```
#evaluate model for a fold
train_score = mean_absolute_error(y_train, train_preds)
val_score = mean_absolute_error(y_val, val_preds)
```

```
  warnings.warn(smsg, UserWarning)
Val Score: 6.61353 ± 0.59647 | Train Score: 6.04233 ± 0.45635 | XGBoost
fold:0, Val Score: 6.650475526798687, Train Score: 4.972179806224664
fold:1, Val Score: 7.422689645499801, Train Score: 5.3727768601913946
fold:2, Val Score: 7.663932010415142, Train Score: 5.977863323193469
fold:3, Val Score: 7.27150484912544, Train Score: 6.193702867103443
fold:4, Val Score: 6.223767826337636, Train Score: 6.378619824889995
fold:5, Val Score: 6.090286371739084, Train Score: 6.33527456692885
fold:6, Val Score: 6.5637552277036715, Train Score: 6.287565154316903
fold:7, Val Score: 6.31732102893549, Train Score: 6.318108853350397
fold:8, Val Score: 6.037901709376864, Train Score: 6.3098912006188295
fold:9, Val Score: 5.8936869639033205, Train Score: 6.2773465893420966
Best validation score: 5.8936869639033205, associated train score: 6.2773465893420966
```

Figure 6: time series split scores

Validation Scores: The validation scores across the ten folds show some variance, with the lowest being approximately 6.05 and the highest around 7.66. This suggests that the model's performance isn't consistent across different segments of the time series data.

Training Scores: Training scores are relatively close to the validation scores within each fold, which is a good indication that the model isn't severely overfitting to the training data. However, the training scores are consistently lower than the validation scores, which is expected and indicates some level of overfitting.

Best Performance: The best validation score is around 5.89, which occurred in fold 9, with an associated training score of approximately 6.27.

Fluctuations: The noticeable fluctuations in scores from fold to fold could be due to several factors

- Changes in market behavior over time, which the model is sensitive to.
- Possible overfitting to particular patterns present in certain folds but not others.
- The default model parameters may not be optimal across all segments of the time series.

4.1.2 XGBoost Model Second Run Feedback with StandardScaler

Experiment Setup:

In the second run of the XGBoost model, the training features were scaled using StandardScaler. This aims to standardize the features by removing the mean and scaling to unit variance. Standardizing can often lead to better model performance, especially for algorithms that are sensitive to the scale of the data.

Model Training and Validation:

The model training followed the same TimeSeriesSplit cross-validation with 10 folds as in the initial run. The only change in the process was the introduction of feature scaling.

```
Val Score: 6.67774 ± 0.71738 | Train Score: 6.04191 ± 0.45789 | XGBoost
fold:0, Val Score: 6.651442431729478, Train Score: 4.9678951647068
fold:1, Val Score: 7.463162911172949, Train Score: 5.370602119043408
fold:2, Val Score: 8.253435960585419, Train Score: 5.977173397138673
fold:3, Val Score: 7.25387176014858, Train Score: 6.193839269671612
fold:4, Val Score: 6.296115778234773, Train Score: 6.3795233442720605
fold:5, Val Score: 6.07792033942205, Train Score: 6.338010684557273
fold:6, Val Score: 6.565528257620425, Train Score: 6.287720338852062
fold:7, Val Score: 6.3102524292045405, Train Score: 6.317997149035993
fold:8, Val Score: 6.023147421178138, Train Score: 6.310524511169559
fold:9, Val Score: 5.882476722828915, Train Score: 6.275806372850901
Best validation score: 5.882476722828915, associated train score: 6.275806372850901
```

Figure 7: Training and validation scores for standard scaler

Model Performance Insights:

Validation and Training Scores: Like the initial run, the validation and training scores across the ten folds varied. The scores ranged from approximately 6.1 to 7.4 on the validation set and from around 4.9 to 6.2 on the training set.

Impact of Scaling: Applying StandardScaler seems to have impacted the validation scores positively slightly, as indicated by a lower mean validation score compared to the initial run without scaling.

Best Fold: The fold with the best validation score was fold 9, with a validation score of approximately 5.88 and an associated training score of around 6.27. This suggests that the model has managed to generalize well on this segment of the data.

Score Variability: The standard deviation of the validation scores is lower than in the initial run, indicating that the model's performance is more consistent across different folds after scaling the features.

4.1.3 XGBoost Model Analysis with K-Fold Cross-Validation

Experiment Summary:
In this iteration, the XGBoost model was subjected to a K-Fold cross-validation scheme with 10 splits. This strategy deviates from the previous time series split, potentially adjusting for temporal dependencies differently. The model parameters remained consistent with the initial setup to maintain comparability.

Model Performance Overview:

```
  warnings.warn(smsg, UserWarning)
Val Score: 6.29997 ± 0.54734 | Train Score: 6.23408 ± 0.06038 | XGBoost
fold:0, Val Score: 5.214812064532243, Train Score: 6.351955683614929
fold:1, Val Score: 6.2720395977725705, Train Score: 6.239644802178207
fold:2, Val Score: 7.207334967348881, Train Score: 6.133554564743615
fold:3, Val Score: 6.95519325684386, Train Score: 6.160805100623553
fold:4, Val Score: 6.766504592288016, Train Score: 6.182261818859731
fold:5, Val Score: 5.9371821922697166, Train Score: 6.273262671941194
fold:6, Val Score: 6.368913956178462, Train Score: 6.226118771546864
fold:7, Val Score: 6.313149945169311, Train Score: 6.232847648386358
fold:8, Val Score: 6.0484279049740515, Train Score: 6.261248201198719
fold:9, Val Score: 5.916099201611698, Train Score: 6.279123133650754
Best validation score: 5.214812064532243, associated train score: 6.351955683614929
```

Figure 8: k-folder split scores

Validation Scores: The model exhibited a range of validation scores, with the lowest being approximately 5.21 and the highest around 7.21. The variance in validation scores across folds suggests different levels of model performance for various segments of the dataset.

Training Scores: Training scores were relatively stable, fluctuating between approximately 6.23 and 6.35. These scores are slightly more consistent than the validation scores, which is an expected outcome as the model is directly learning from the training data.

Consistency and Variability:
The standard deviation of the validation scores in the K-Fold cross-validation is approximately 0.543, which is lower than the standard deviation observed in the previous time series split runs. This reduced standard deviation indicates a more consistent model performance across the different folds when using a K-Fold strategy.

Best Fold Performance:
The best validation score was achieved in fold 0, with a score of approximately 5.21, and the associated training score for that fold was around 6.35. This suggests that for this particular data partition, the model's predictive accuracy was the highest.

4.1.4 XGBoost Model Feedback with PCA

Experiment Context:
The experiment incorporated Principal Component Analysis (PCA) into the feature engineering process. PCA is used for dimensionality reduction, transforming the data into a smaller set of uncorrelated variables while retaining most of the information. In this case, PCA was applied after standardizing the features and imputing missing values with the mean. It is hoped that PCA can help to create clearer boundaries for the decision trees used by the XGBoost model.

PCA Implementation:
The PCA analysis determined that three principal components were sufficient to capture 95% of the variance in the data. This substantial reduction in feature space aims to focus the model on the most informative aspects of the data.

Model Training and Evaluation:
The transformed dataset with three principal components was then used to train the XGBoost model using K-Fold cross-validation with 10 splits.

The model's performance was evaluated using the mean absolute error (MAE) metric for both the training and validation sets.

```
Val Score: 5.60285 ± 0.38991 | Train Score: 5.21884 ± 0.04300 | XGBoost
fold:0, Val Score: 4.829147891132235, Train Score: 5.229792845414395
fold:1, Val Score: 5.62854728041726, Train Score: 5.17726180984362
fold:2, Val Score: 6.1976428214652595, Train Score: 5.151080414493467
fold:3, Val Score: 6.104356862585992, Train Score: 5.254707525496789
fold:4, Val Score: 5.951381992278112, Train Score: 5.244198625280036
fold:5, Val Score: 5.285984083335795, Train Score: 5.302952432636409
fold:6, Val Score: 5.68384174814063, Train Score: 5.247276401513742
fold:7, Val Score: 5.513362193718579, Train Score: 5.181644506112343
fold:8, Val Score: 5.345304374305956, Train Score: 5.191858247008685
fold:9, Val Score: 5.488891270263276, Train Score: 5.207577964497738
Best validation score: 4.829147891132235, associated train score: 5.229792845414395
```

Figure 9: k-fold split with PCA scores

Model Performance Overview:
Validation Scores: The model achieved a range of validation scores across the 10 folds, from approximately 4.82 to 6.19. The variation in scores indicates how the model performed across different subsets of the transformed feature space.

Training Scores: The training scores were consistent, with values closely clustered around the mean training score, which suggests the model trained effectively on the reduced feature set.
Consistency and Variability:

The standard deviation of the validation scores in this PCA-based experiment was lower compared to the initial run without PCA, indicating improved consistency in the model's performance across folds.

Best Model Selection:
The fold with the best validation performance (fold 0) achieved a score of approximately 4.82, and the associated training score was around 5.23, suggesting that the model was able to make accurate predictions on that particular subset of the data.

Discussion for PCA Result

Although the training and validation score is exceptionally better than other approaches, we are unable to replicate the success when predicting the test set for Kaggle submission. One of the possible causes leading to the model's performance is overestimated due to **data leakage**. As in this experiment, PCA was performed before the training and validation split, thus the training set may already contain information from the testing set.

The other potential cause for overperformance is due to the NaN values being imputed by using the mean of the columns. For near_price and far_price columns, This means more than half of

the rows are being filled with the mean. This could potentially cause data leakage or unbeneficial manipulation of the data.

Due to the reasons highlighted, PCA() was not used moving forward.

4.2 Feature Importance Analysis Report

The feature importances derived from the XGBoost model provide valuable insights into which features are most influential in predicting the target variable. Here is a summary of the findings from the time series split and K-folder split run:

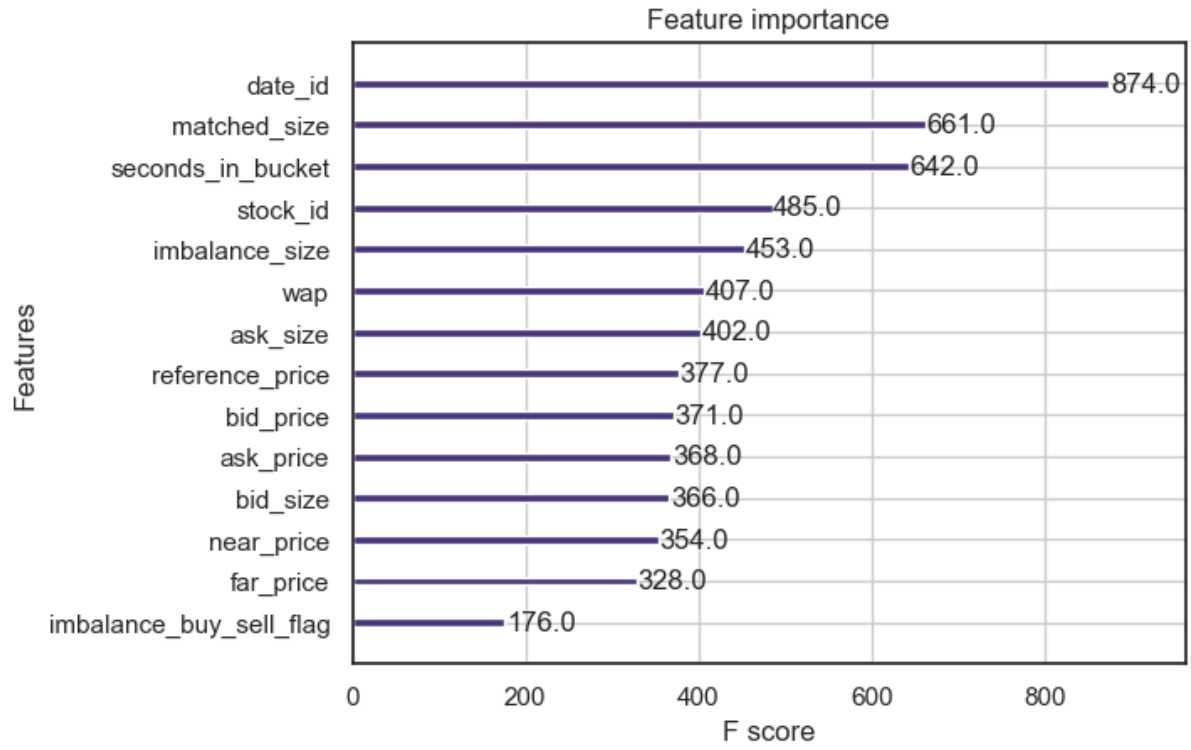

Figure 10: Feature importance from time series split

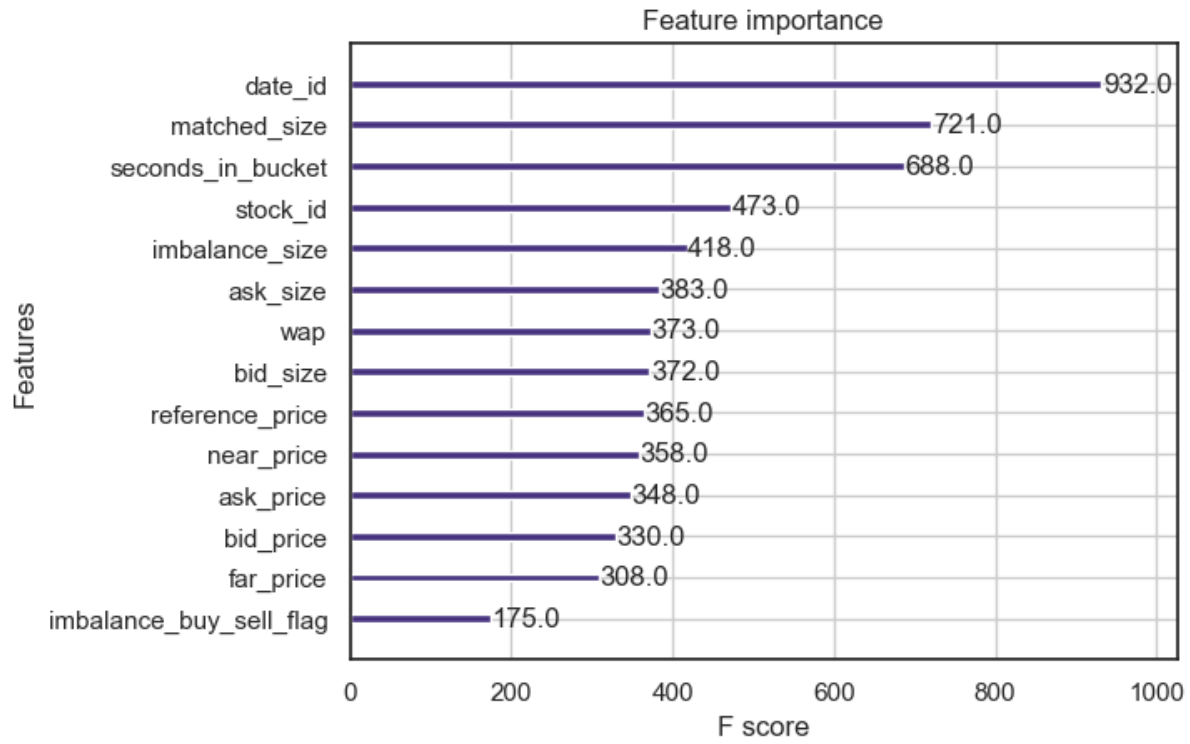

Figure 11: Feature importance from k-folder split

Comparing these results may reveal whether the model's reliance on features changes when the temporal dimension is handled differently:

Temporal Features:

- date_id: This feature consistently ranks as the most important across both TSS and K-Fold models, with even higher importance in the K-Fold validation (932 vs. 874). This underlines the strong effect of date-related factors on the target variable.
- seconds_in_bucket: There is a noticeable increase in importance in the K-Fold validation (688) compared to TSS (642), suggesting that this feature's predictive power is not solely dependent on the temporal order of data.

Transaction Size Features:

- matched_size: Exhibits increased importance in the K-Fold cross-validation (721) versus TSS (661), maintaining its status as a highly predictive feature across both models.

Price Features:

- reference_price, bid_price, ask_price: These features showed a slight decrease in importance in the K-Fold results, indicating a potential dependency on the temporal sequence of data for their predictive strength in the TSS model.
- wap (Weighted Average Price): The importance of this composite price feature is stable across both models, demonstrating its consistent value in predicting the target variable.

Order Size Features:

- bid_size and ask_size: Their importance remains relatively consistent across both models, with slight variations. This suggests that order sizes have a stable influence regardless of the validation strategy.

4.3 Hyperparameter Tuning Results

4.3.1 Tuning Sessions Results

The First Tuning session description and *the hyperparameter results* are as follows:

Model Score: 5.414
Key Hyperparameters:

- Moderate n_estimators and max_depth suggest a balance between complexity and generalization.
- A lower learning rate combined with a moderate gamma indicates a careful approach to learning with an eye on regularization.
- High subsample and colsample_bytree values close to 1 suggest using most of the data and features for learning, which can be suitable for smaller datasets or those with less noisy features.
- The low reg_lambda value suggests less weight on L2 regularization.

The second Tuning session description and results are as follows:

Model Score: 5.4539, slightly worse than the first tuning.
Key Hyperparameters:

- Decreased n_estimators and max_depth, suggesting a simpler model that might underfit or generalize better.
- A higher learning_rate indicates faster learning but can lead to missing optimal solutions.
- A very high gamma value suggests strong regularization against complex models.
- The colsample_bytree is nearly 1, indicating using all features for building trees.
- Extremely low reg_lambda which might indicate that L2 regularization isn't critical for this dataset.

The third Tuning session description and results are as follows:
Model Score: 5.4262
Key Hyperparameters:

- Increased n_estimators (315): Suggests a more complex model with a higher number of trees, potentially capturing more intricate patterns but with an increased risk of overfitting.
- Moderate learning_rate (0.060666355678753454): Indicates a balanced approach to learning, neither too slow (which might be inefficient) nor too fast (which might skip optimal solutions).

- Higher max_depth (9): Allows trees to grow deeper, enabling the model to learn more complex structures in the data, but it also increases the risk of overfitting.
- Moderate min_child_weight (6): A higher value than the default, suggesting a more conservative approach in creating leaf nodes, which can help in preventing overfitting.
- High gamma (0.49091230014294834): Implies a strong regularization effect, making the model more conservative by cutting down on further splits unless there's a significant reduction in the loss.
- High subsample (0.9695398997536578): Indicates that a large fraction of the data is used for growing trees, leading to higher variability and potentially better model performance.
- Moderate colsample_bytree (0.6814598096875906): Shows a balanced approach in selecting a subset of features for each tree, which can help in managing overfitting while still capturing essential feature interactions.
- Low reg_alpha (0.002392596935934544) and reg_lambda (0.00022946212402305305): Suggest minimal emphasis on L1 and L2 regularization, indicating that the model might not be prone to overfitting or that these forms of regularization are not crucial for the dataset at hand.

4.3.2 Hyperparameter Importance

The next figures show the hyperparameter importance ranking derived from three times hyperparameter tunings.

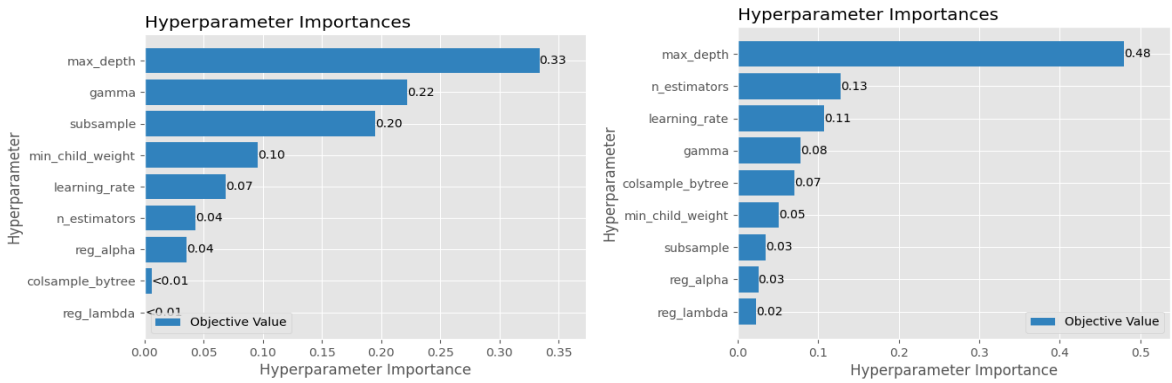

Figure 12： Hyperparameter Importance ranking from various runs

Explanation:

- max_depth tends to have a substantial impact on model complexity, with deeper trees potentially capturing more nuanced patterns but also risking overfitting.
- n_estimators and learning_rate often trade-off with each other; a higher number of estimators can be balanced with a lower learning rate to prevent overfitting while ensuring sufficient model complexity.
- gamma, min_child_weight, and subsample are regularization hyperparameters that help control overfitting.
- colsample_bytree, reg_alpha, and reg_lambda offer additional ways to regularize the model and can significantly impact performance depending on the dataset.

4.3.3 Optimization History

The optimization history plots typically show the objective value (MAE) over a series of trials. The "best value" line helps to quickly identify the most promising hyperparameter sets. The following figures show the optimization history plot for the second and third runs.

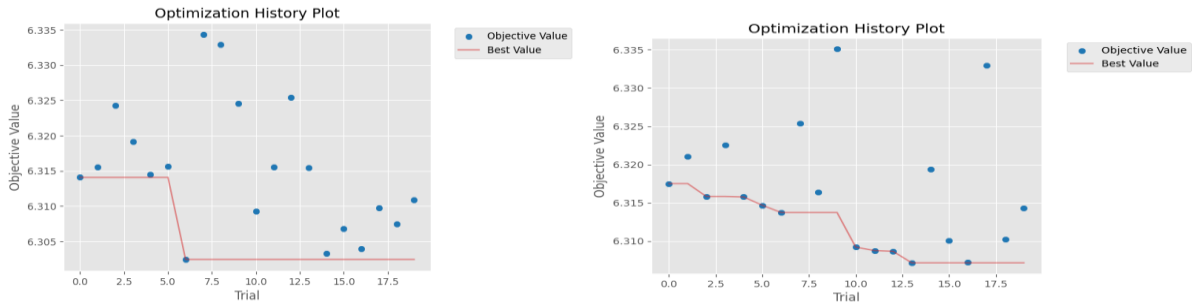

Figure 13: Optimization History Plot

4.4 XGBoost Results Summary

Table 1 shows the summary of results in all the previous sections for the experiment with the XGBoost model. The best test score achieved for submission to Kaggle is 5.4142. It can concluded that

- Cross-validation split using k-fold is better than the tss
- There is no significant improvement using StandardScaler
- Although PCA seemed to be able to achieve better results with certain processing steps. It is unlikely to bring improvements to actual test submissions.

Test Title	Training Score		Validation Score		Test Score (Submission)
	Average	Best Validate	Average	Best	
XGBoost with tss	6.04233	6.27734	6.61353	5.89368	N/A
XGBoost with k-fold	6.23408	6.35195	6.2999	5.21481	N/A
Scaled XGBoost with tss	6.04191	6.27580	6.67774	5.88247	N/A
Scaled XGBoost with k-fold	6.23368	6.35322	6.29938	5.22012	N/A
XGBoost with PCA	5.21884	5.22979	5.60285	4.82915	N/A
First Tuning	6.31042	6.30532	N/A	N/A	5.4142
Second Tuning	6.31143	6.30353	N/A	N/A	5.4539
Third Tuning	6.31497	6.30682	N/A	N/A	5.4262

Table 1: Summary of test conducted with XGBoost

5. LightGBM Model

The experiments with LightGBM are organized as below:

- Exploring preprocessing techniques:
  - An initial run was done with all the features available in the training data set, using Scikit-learn's TimeSeriesSplit for Cross-Validation, without scaling the data using the StandardScaler(), and using all default hyperparameter values of LGBMRegressor.
  - Test the effect of applying StandardScaler() with the rest of the processing and setting equal to the initial run.
  - Test the effect of using normal k-fold cross-validation (non-time series data) with the rest of the processing and setting equal to the initial run.
  - Test the effect of applying PCA (principal component analysis) on the training data set
- Exploring Feature Engineering
  - The first test will check if removing seemingly unrelated features will affect or improve the training performance.
  - The second test will check if extra engineered features help.
    - First extra sets– incorporate the movement of the reference price, this includes the first derivative, which is the price of the current minus the price of the previous 10 seconds, and the second derivative, which is the change of the first derivative in the last 10 seconds.
    - Second extra sets – add the first and second derivatives for all 8 prices.
    - Third extra sets – Add the imbalance factors
  - The third test set tries to predict the two dependent variables which define the final target separately and combine the result.
- Exploring hyperparameter tunings of the LightGBM model
  - The hyperparameters considered include n_estimators, learning_rate, num_leavers, max_depth, min_data_in_leaf, max_bin, min_gain_to_split and subsample.

5.1 Exploring Preprocessing Techniques

5.1.1 Initial Trial Run

Figure 14 shows the training and validation scores for the initial run using all features available in the training data set and Scikit-learn's TimeSeriesSplit of 10 for cross-validation. The main observations are as follows:

Observation 1: The average validation scores are about 5% higher than the average training scores, this indicates potential overfitting of the model to the training data.

Observation 2: The validation score seems to be improving when more data is used for training. This indicates that the series might be stationary without a trend or seasonal component. Note when using the Scikit-learn's TimeSeriesSplit function, the testing data of the first fold is added to the training data of the second fold (see **Figure 15** for illustration), thus the training data set will continue to expand.

Since the number of days in the training data set is 480, one concern is the original split (TimeSeriesSplit of 10 will split the training data set into 11 parts) may contain fractional data of the day which affect the performance of the model. A quick test was performed using the TimeSeriesSplit setting of 9 (which split the data into 10 parts) with the result shown in table 1. As the resulting models didn't perform significantly better than the previous models, it is concluded that the number of splits has no significant effect on the performance.

```
Val Score: 6.47656 ± 0.54012 | Train Score: 6.14128 ± 0.42522 | LightGBM
fold:0, Val Score: 5.946031802134712, Train Score: 5.1532272783947946
fold:1, Val Score: 7.323238700736963, Train Score: 5.503533047046515
fold:2, Val Score: 7.199806914615412, Train Score: 6.089348053341895
fold:3, Val Score: 7.234725565869836, Train Score: 6.2920516346192334
fold:4, Val Score: 6.1653873848025, Train Score: 6.475964154419878
fold:5, Val Score: 6.045562384448515, Train Score: 6.421133250051477
fold:6, Val Score: 6.562517120671359, Train Score: 6.3667592016489625
fold:7, Val Score: 6.324485563729338, Train Score: 6.3893219047254695
fold:8, Val Score: 6.078645750424249, Train Score: 6.3793287398361755
fold:9, Val Score: 5.885231645998333, Train Score: 6.342175608974108
Best validation score: 5.885231645998333, associated train score: 6.342175608974108
```

Figure 14: Training scores and Validation scores for the initial run

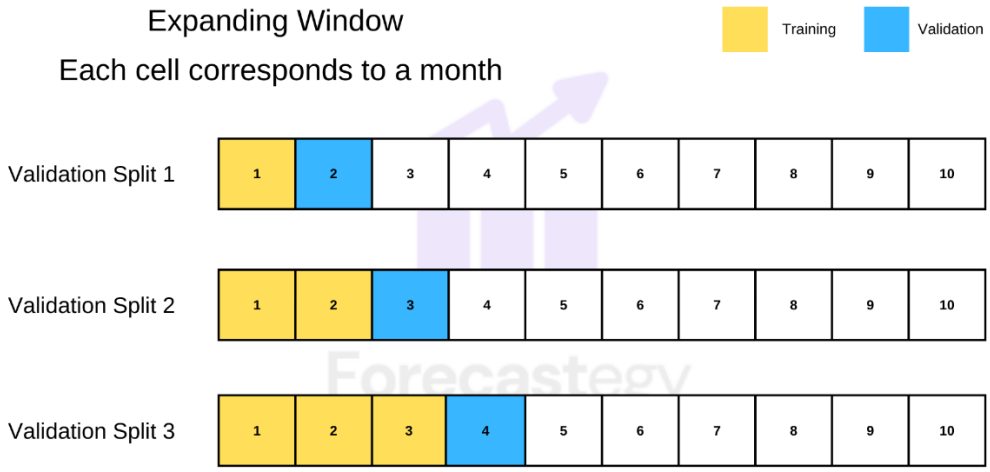

Figure 15: Expanding Window Time Series Split Validation (Source: (Filho, 2023))

5.1.2 Initial Trial with Standard Scaler Applied

This test applied StandardScaler() to the following numerical categories: imbalance_size, reference_price, matched_size, far_price, near_price, bid_price, bid_size, ask_price and ask_size. The results are shown in Table 2. It doesn't seem to have significant improvement compared to training data without StandardScaler() applied.

5.1.3 Initial Trial with k-fold cross-validation

This test uses the k-fold instead of TimeSeriesSplit for cross-validation. The results are shown in **Figure 16**. On average the model performs better compared to the model trained using the TimeSeriesSplit. Besides, there are two observations:

Observation 1: The is one part of the data (the first part) where the model predicts well.

Observation 2: The model performs better in the validation set when it is underfitting the training data. This happens when the validation score is lower than the training score achieved.

```
[LightGBM] [Info] Start training from score -0.045517
Val Score: 6.32005 ± 0.55010 | Train Score: 6.29844 ± 0.06159 | LightGBM
fold:0, Val Score: 5.234492085589952, Train Score: 6.42056048182349
fold:1, Val Score: 6.2949092659642325, Train Score: 6.3020573593216485
fold:2, Val Score: 7.2345442496949985, Train Score: 6.196144490776656
fold:3, Val Score: 6.972368693892824, Train Score: 6.224936147482282
fold:4, Val Score: 6.789950707068252, Train Score: 6.24650201468046
fold:5, Val Score: 5.958771478459255, Train Score: 6.339136552342583
fold:6, Val Score: 6.392767914749096, Train Score: 6.290235521507489
fold:7, Val Score: 6.336152908468828, Train Score: 6.296386848750162
fold:8, Val Score: 6.079116732159518, Train Score: 6.325292155098591
fold:9, Val Score: 5.907416843950074, Train Score: 6.3431194666341515
Best validation score: 5.234492085589952, associated train score: 6.42056048182349, fold:0
```

Figure 16: Training scores and Validation scores for the initial run with k-fold cross-validation

5.1.4 Initial Trial with PCA

In this trial, Principle Component Analysis (PCA) was performed on the training data set hoping to improve the decision boundaries for the LightGBM model. However as PCA does not handle NaN values, there are two choices. The first choice is to drop all rows whenever one of the columns contains NaN values. However, there are too many NaN values in the far_price and near_price (about 2.8 million each), and dropping them will reduce the training data size to about 0.4 of the original. Although this approach yields good results (see Table 2), the good results applied only to the smaller dataset with NaN values available. Furthermore, the model won't be able to deal with new testing data that contains the NaN values. Hence this approach was not selected.

The second choice is to replace all the NaN values in the far_price and near_price columns with 0 while dropping other rows with NaN values. This approach preserves more than 99% of the training data. As shown in Table 1, this approach yields a minimal improvement compared to training with data without the PCA transformation.

5.1.5 Summary of Preprocessing Techniques

Table 2 shows the summary of the comparison of the performance of the models using different preprocessing techniques. Based on the results, it can be concluded that,

- The training data exhibits stationary property with variations independent of time, hence processing the data in time series is not necessary.
- The best-performing model is likely to be the model that underfit the training data.
- Applying StandardScaler or PCA yields minimal or no improvement

Test Title	Training Score		Validation Score	
	Average	Best Validate	Average	Best
Initial Run	6.14128	6.34218	6.47656	5.88523
With TimeSeriesSplit of 9	6.15360	6.34312	6.47738	5.90742
With StandardScaler	6.14125	6.34143	6.48478	5.88302
With kfold	6.29844	6.42056	6.32005	5.23449
With PCA	6.29806	6.42048	6.31941	5.23251
With PCA(drop na)	5.59804	5.69312	5.63493	4.77072

Table 2: Summary comparison of the performance of the models for different preprocessing techniques

5.2 Exploring Feature Engineering

Figure 17 shows the feature importance (Gain) for all the features used to train the best model. It seems that the feature ask_size and bid_size play a significant role compared to the other features. **Figure 18** shows the feature importance (Split) for all the features used to train the best model. The distribution of the feature importance is more even compared to the feature importance (Gain).

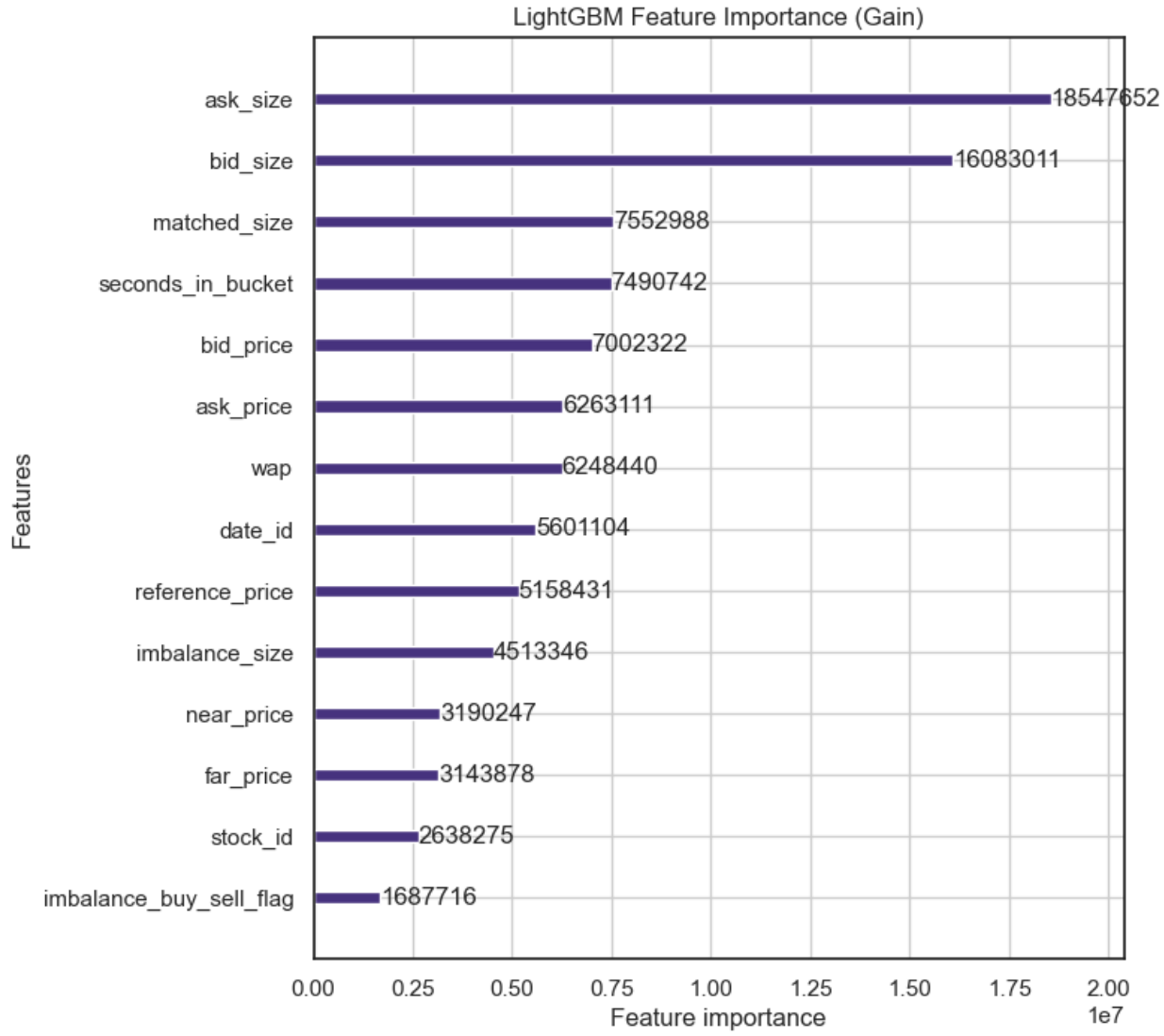

Figure 17: Feature Importance (Gain) for the base model

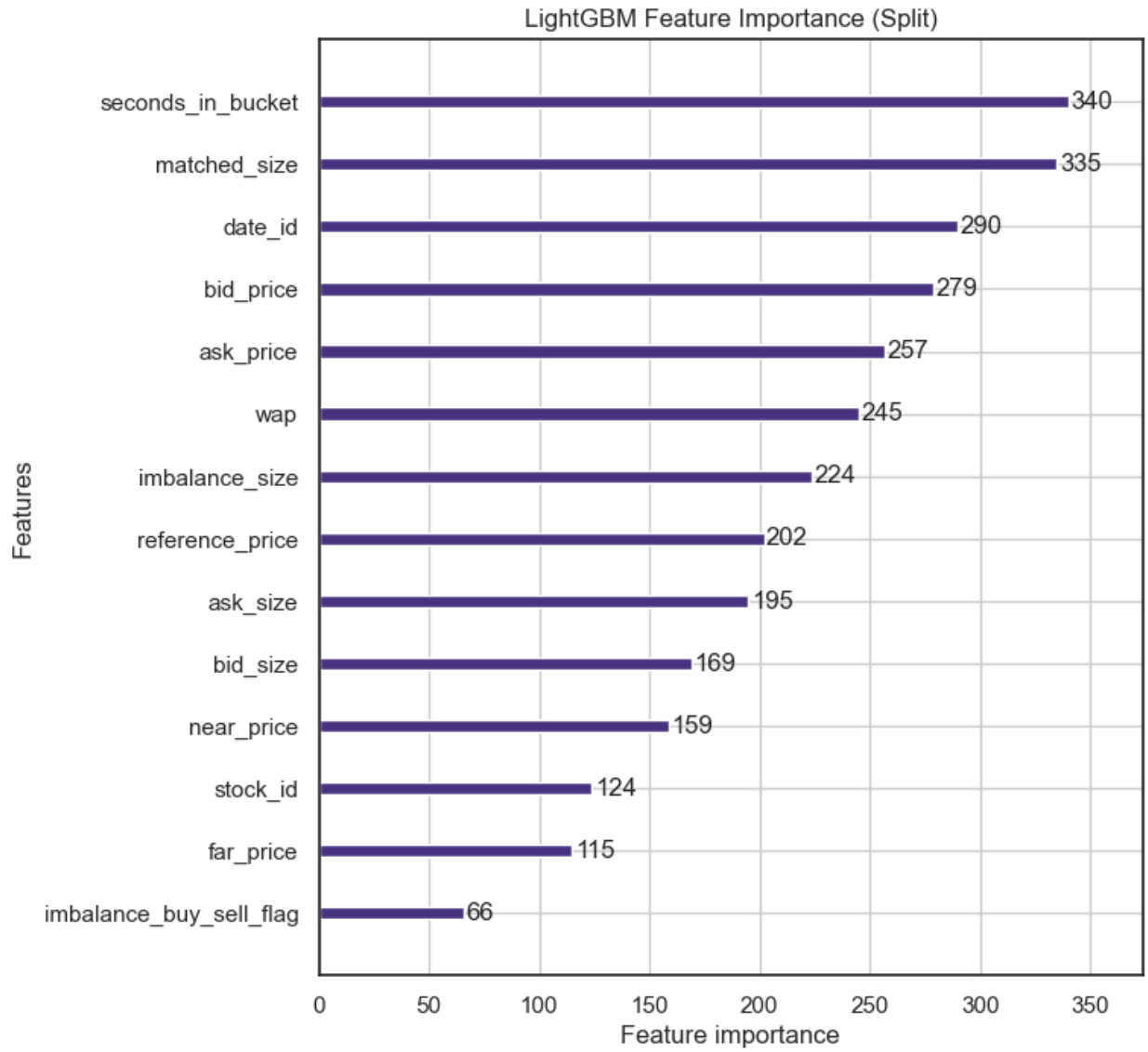

Figure 18: Feature Importance (Split) for the base model

5.2.1 Removing Less Important Features

Based on the feature importance (Gain & Split) for the best base model shown before, an experiment was carried out by removing the four least important features (near_price, stock_id, far_price, imbalance_buy_sell_flag). The results are shown in **Table 3**. Compared with the initial run, there is a slight drop in the average training score and validation score performance. As the extra features don't affect the training speed and contribute slightly to the performance of the models, it is decided to keep these four features for future training.

5.2.2 Adding First and Second Derivatives

The idea is to try to capture the movement trend (first and second derivative) for the eight potential trending features (matched_size, bid_price, ask_price, wap, ask_size, bid_size, reference_price, and imbalance_size). Taking the reference price as an example, the hope is that if the model knows the relative movement of the reference price (change in price in last 10

seconds) and the relative change in movement (change of change in price in last 10 seconds) it can learn better the direction where the target is heading.

Figures 19 & 20 show the performance and feature importance (gain & split) for the model trained with the first and second derivatives of the reference price. As shown in the figure the first and second derivatives have a minor impact on the gain and split of the model. As shown in **Table 3**, there is minimal improvement in the training results compared to the initial base model.

Figures 21 & 22 show the performance and feature importance (gain & split) for the model trained with the first and second derivatives of the eight potential trending features. the first and second derivatives have a minor impact on the gain and split of the model, with the second derivatives generally having less impact compared to the first derivatives. As shown in **Table 3**, there is a minimal improvement in the training results compared to the initial base model, and negligible improvement compared with the model trained using additional first and second derivatives of the reference price.

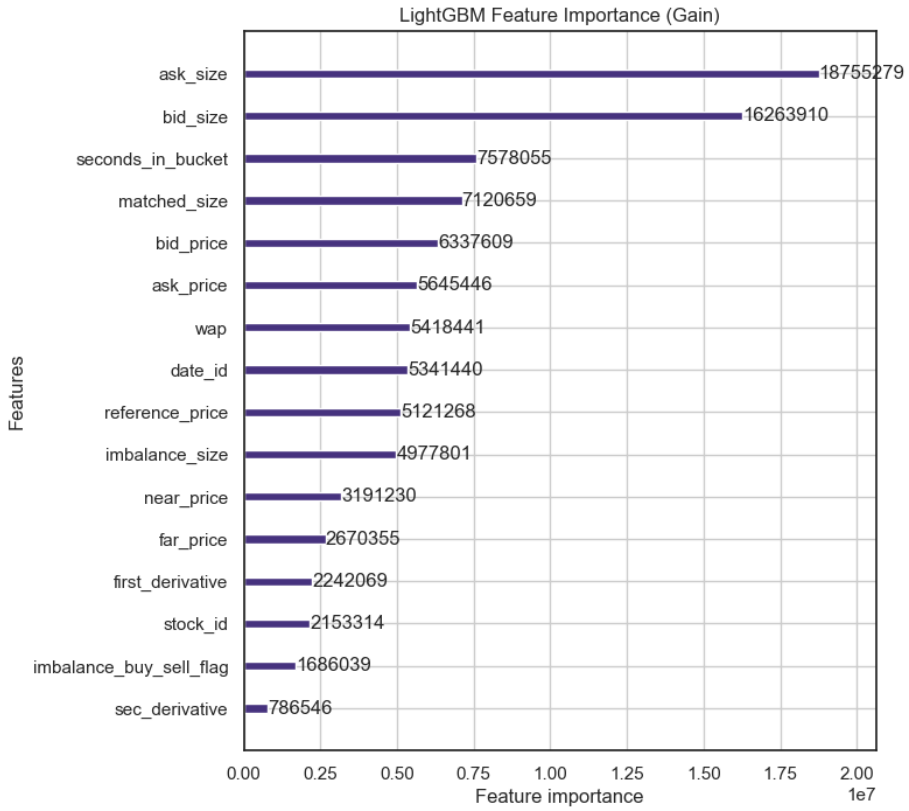

Figure 19: Feature Importance (Gain) for model trained with first and second derivatives of reference price

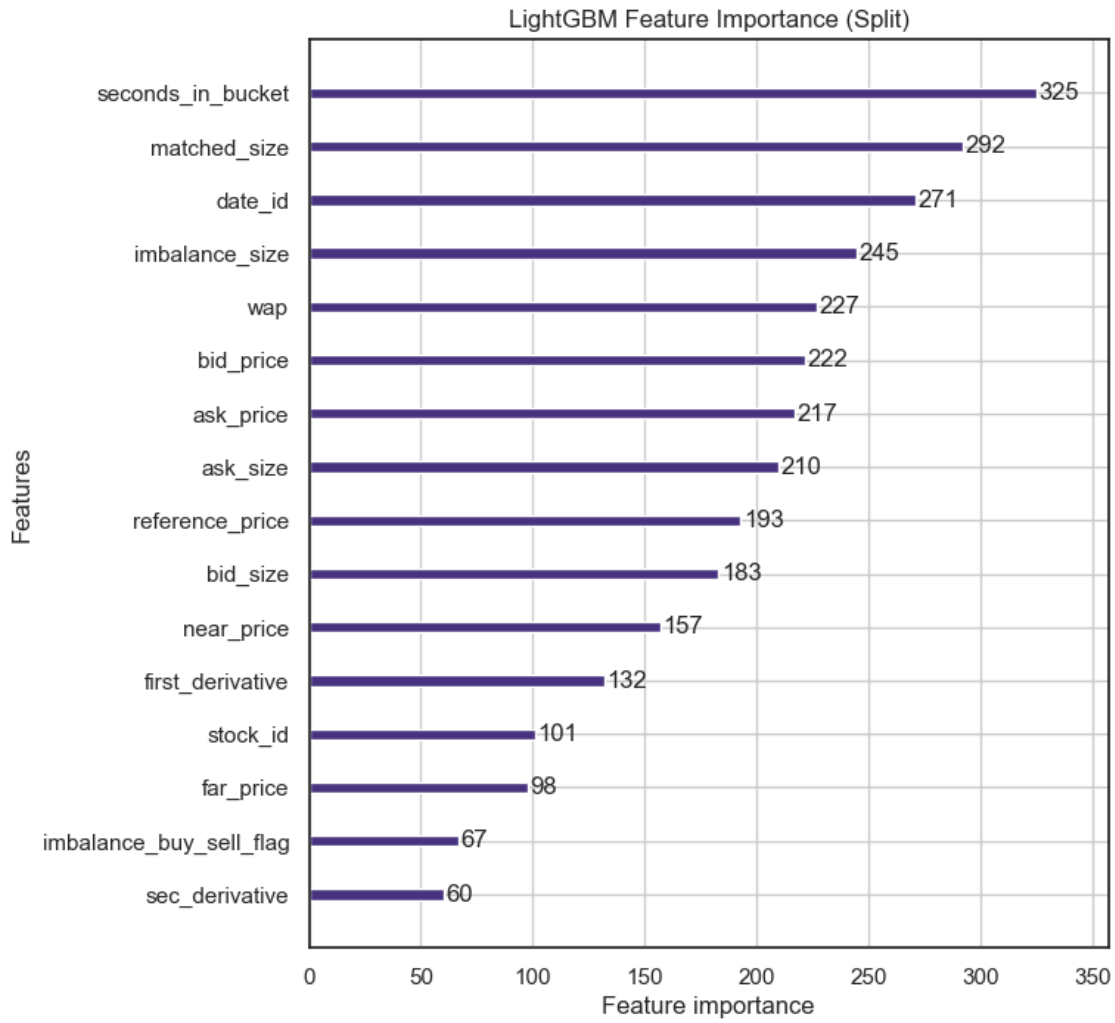

Figure 20: Feature Importance (Gain) for a model trained with first and second derivatives of reference price

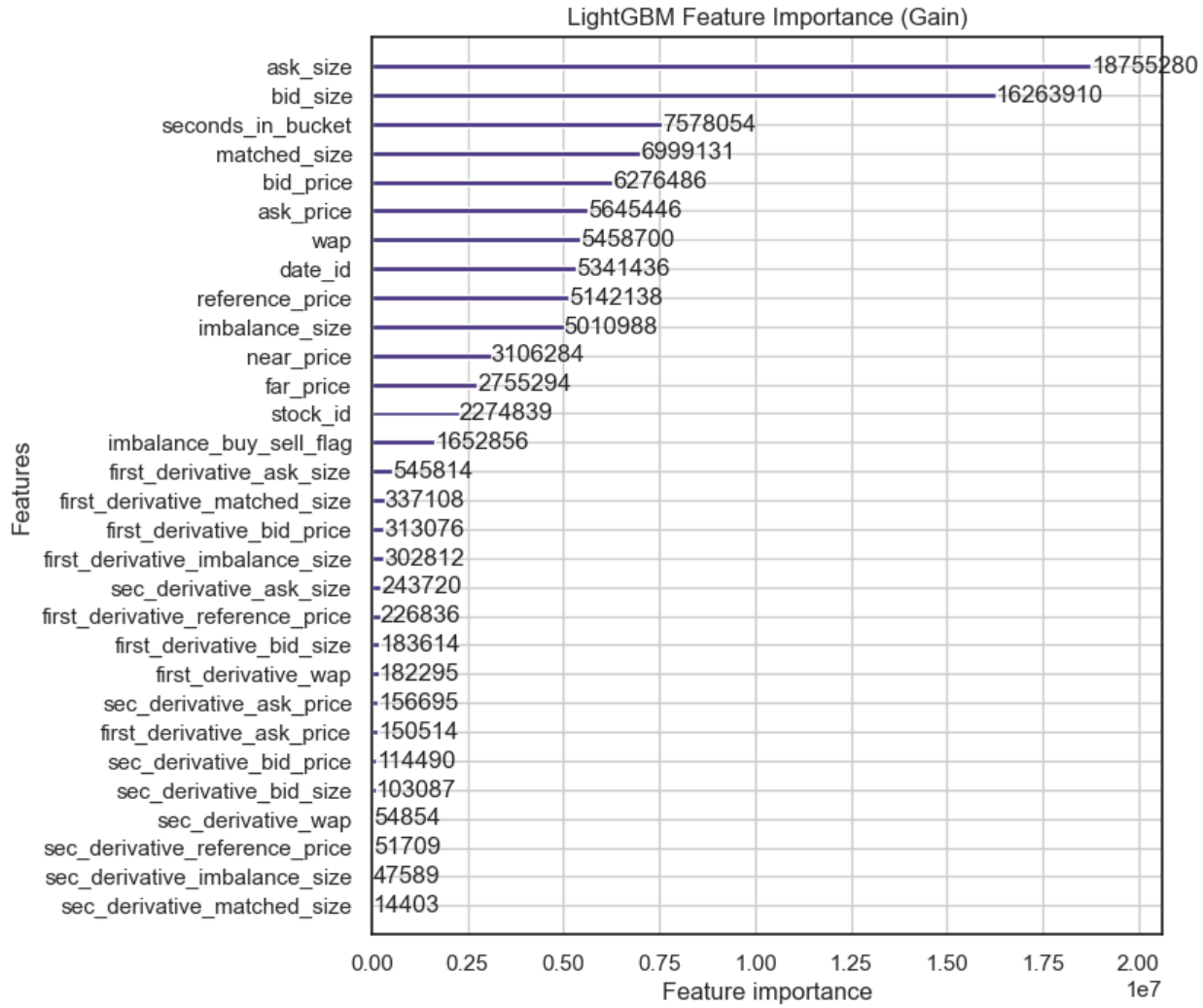

Figure 21: Feature Importance (Gain) for model trained with first and second derivatives of 8 features

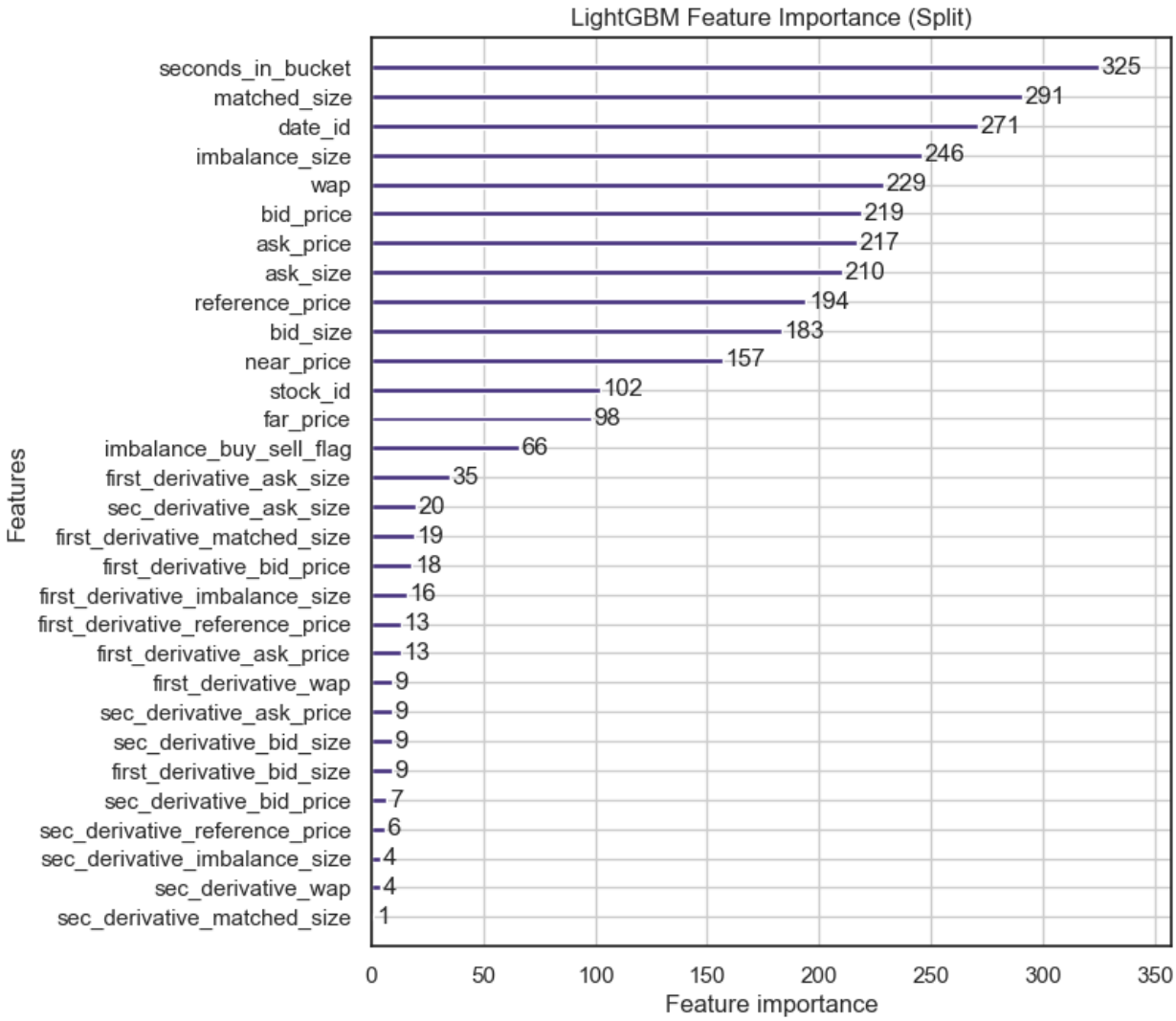

Figure 22: Feature Importance (Split) for a model trained with first and second derivatives of 8 features.

5.2.3 Adding Imbalance Features

The imbalance features were proposed by one of the Kaggle competition participants (zhezhou, 2023). The formula for two of the most important imbalance features is shown below.

Formula for imb_s1

$$imb_{s1} = \frac{bid_{size} - ask_{size}}{bid_{size} + ask_{size}}$$

Formula for imb_s2

$$imb_{s2} = \frac{imbalance_{size} - matched_{size}}{imbalance_{size} + matched_{size}}$$

Reference formula for wap

$$wap = \frac{BidPrice * AskSize + AskPrice * BidSize}{BidSize + AskSize}$$

The results of applying the imbalance features are shown in **Table 3, Figures 23 & 24**. As compared to the base model in **Table 3**, there is a minor improvement in the training results with the imbalanced features added in.

Based on **Figures 23 & 24**, one might wonder if only two imbalance features (imb_s1 and imb_s2) will be enough to improve the result. The test was carried out with only these two additional imbalance features. As shown in **Table 3**, there is a minor degradation of the overall training performance when only using two imbalanced features. Hence the suggestion is to keep all imbalanced features. The resulting test score using test data provided by the competition is 5.39 as shown in **Figure 25**.

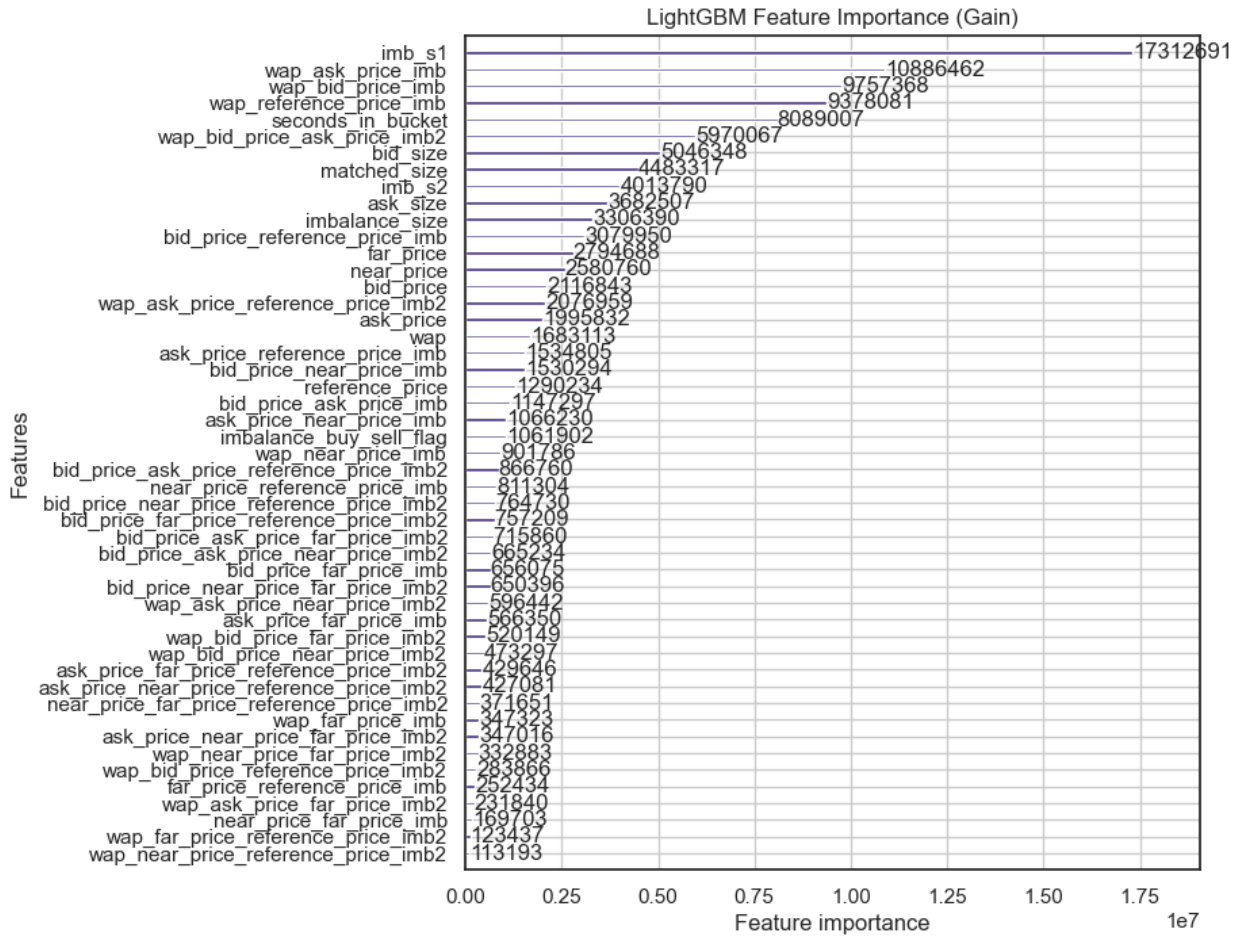

Figure 23: Feature Importance (Gain) for model trained with imbalance features

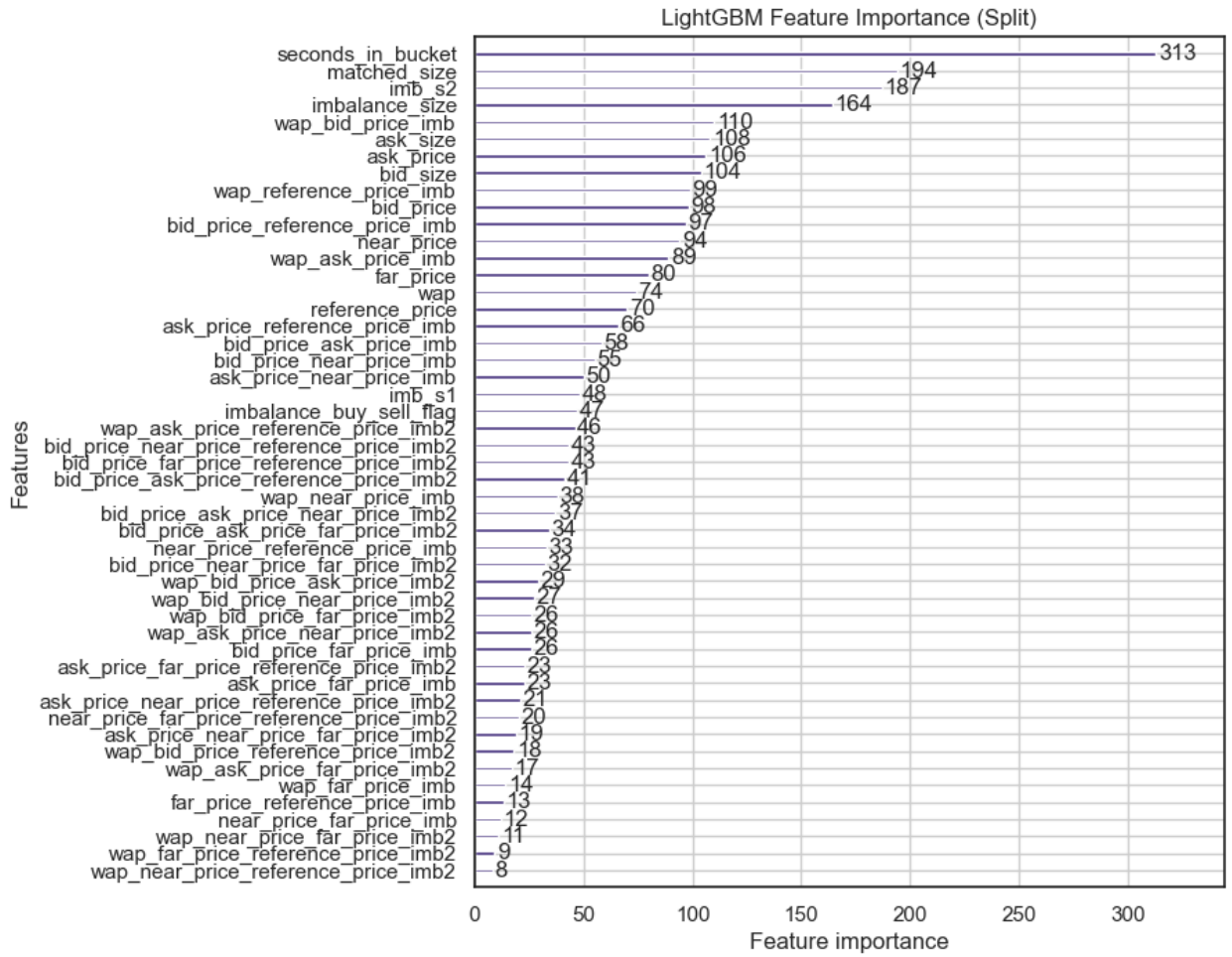

Figure 24: Feature Importance (Split) for model trained with imbalance features

Submission and Description	Public Score	Select
notebook_raw_imb - Version 6 Succeeded · 1h ago · Notebook notebook_raw_imb | Version 6	5.3823	☐
notebook_raw_imb - Version 5 Succeeded · 3h ago · Notebook notebook_raw_imb | Version 5	5.39	☑

Figure 25: Test scores for the model trained with imbalance features

5.2.4 Separate Model for the dependent variable

The prediction target is defined using the following formula (Forbes et al., 2023):

$$Target = \left(\frac{StockWAP_{t+60}}{StockWAP_t} - \frac{IndexWAP_{t+60}}{IndexWAP_t}\right) * 10000$$

From the formula, only $StockWAP_t$ is known by the model, the model needs to predict both $StockWAP_{t+60}$ and $\frac{IndexWAP_{t+60}}{IndexWAP_t}$. Thus the idea is to check if we create a model to predict these two term separately, will it perform better compared to the model predicting their result (the Target) directly?

Tests were conducted using two models with and without the imbalance factors with results shown in **Table 3**. It can be concluded that the two model approaches didn't yield any improvement, probably due to the errors being compounded between the two models.

Pearson correlation and Spearman Correlation analysis were carried out between the columns and the target wap and target change in index separately. The results are shown in **Figures 26 & 27**. Based on the results, for target wap, only the columns wap, ask_price, bid_price, and reference_price have moderate (absolute value of 0.3 – 0.5) relationship with the future wap, and this coincides well with the feature importance (gain & split) shown in **Figure 28** for the model to predict future wap.

For target change in the index, besides the target_wap & target, none of the columns have a moderate or stronger relationship. The top five columns with weak relationships are wap, reference_price, bid_price, ask_price, and near_price. This coincides well with the feature importance (gain & split) shown in **Figure 29** for the model to predict future wap.

Sorted Pearson Correlation:

	0	1
target_index		0.805769
wap		-0.369626
ask_price		-0.357905
bid_price		-0.357894
reference_price		-0.357509
near_price		-0.237978
imbalance_buy_sell_flag		-0.097481
target		0.069783
bid_size		-0.048246
ask_size		0.035485
seconds_in_bucket		-0.014676
date_id		0.007054
time_id		0.007043
far_price		-0.006218
matched_size		-0.003821
imbalance_size		0.002372
stock_id		-0.001847

Sorted Spearman Correlation:

	0	1
target_index		0.778648
wap		-0.356157
bid_price		-0.344897
ask_price		-0.344380
reference_price		-0.343817
near_price		-0.263072
far_price		-0.216873
bid_size		-0.097340
ask_size		0.093346
imbalance_buy_sell_flag		-0.092567
target		0.060433
seconds_in_bucket		-0.020809
imbalance_size		0.005369
date_id		0.002745
time_id		0.002702
stock_id		-0.001426
matched_size		0.000721

Figure 26: Correlation between the columns and the target_wap, sorted by absolute value

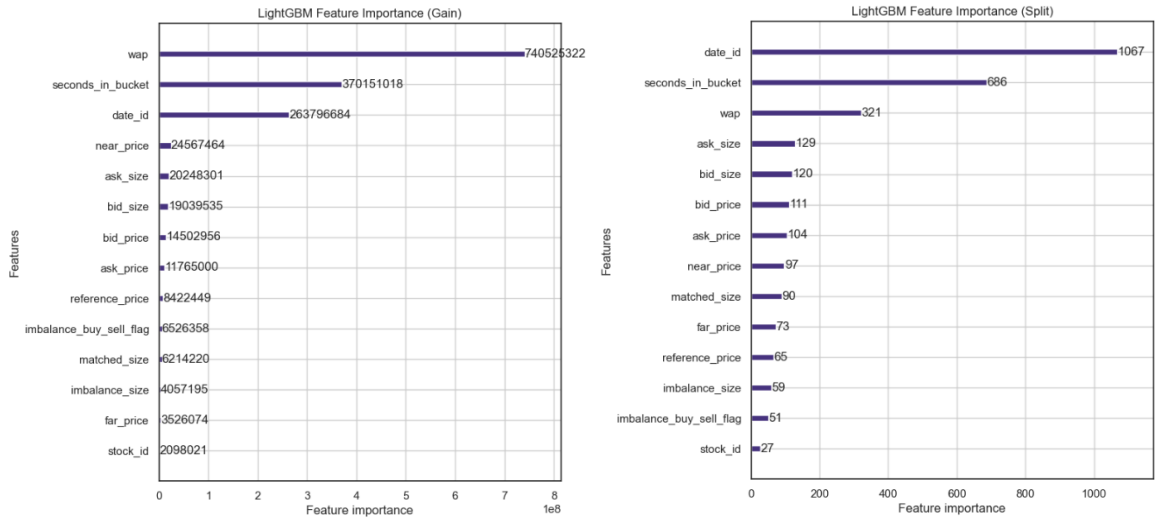

Figure 27: Feature Importance (Gain & Split) for the model to predict future wap

```
Sorted Pearson Correlation:
                          0         1
                target_wap  0.805769
                    target -0.534557
                       wap -0.276808
           reference_price -0.273288
                 bid_price -0.272771
                 ask_price -0.272228
               near_price -0.200354
imbalance_buy_sell_flag -0.083122
                bid_size -0.026391
                ask_size  0.020881
      seconds_in_bucket -0.012884
                 date_id  0.005075
                 time_id  0.005066
               far_price -0.004000
           matched_size -0.003534
                stock_id -0.001113
          imbalance_size  0.001021
```

```
Sorted Spearman Correlation:
                          0         1
                target_wap  0.778648
                    target -0.497866
                       wap -0.265644
                 bid_price -0.262189
           reference_price -0.261954
                 ask_price -0.261180
               near_price -0.212285
                far_price -0.175101
imbalance_buy_sell_flag -0.077478
                bid_size -0.041227
                ask_size  0.039375
      seconds_in_bucket -0.015937
          imbalance_size  0.003342
                 date_id  0.002178
                 time_id  0.002145
                stock_id -0.000637
           matched_size  0.000079
```

Figure 28: Correlation between the columns and the target_index, sorted by absolute value

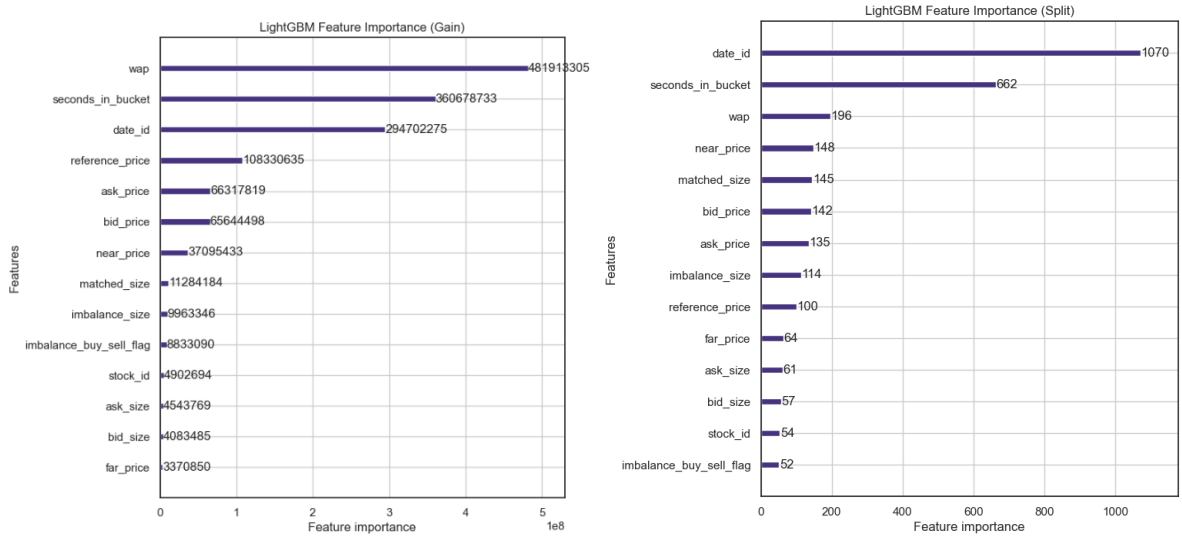

Figure 29: Feature Importance (Gain & Split) for the model to predict change in index

5.2.5 Summary of Feature Engineering

Based on the experiment done against a set of features, it can be concluded that the best feature set will include the derivatives on the reference price and all the imbalance features. A final test was run using the best feature set. As shown in **Table 3**, this indeed generates the best average training scores, best average validation scores, and best validation scores from the cross-validation.

However, as it will be difficult to obtain the first and second derivatives with the test data provided, it is decided to use the imbalance factors only as extra features. The resulting test score is 5.39. The 2 models' approach to predicting the dependent variables separately was not selected as the validation scores and test scores are no better than the single model approach.

Test Title	Training Score		Validation Score		Test Score (Submission)
	Average	Best Validate	Average	Best	
Initial Run with kfold	6.29844	6.42056	6.32005	5.23449	5.4249
4 features removed	6.30591	6.42768	6.32560	5.23688	5.4245
With derivatives on Reference price	6.29823	6.41934	6.31806	5.23228	N/A
With derivatives on all trend features	6.29823	6.41934	6.31804	5.23228	N/A
With all imbalance features	6.26616	6.38593	6.28535	5.1937	5.39
With only two imbalance features	6.29743	6.41905	6.31597	5.22403	N/A
With derivatives on reference price & all imbalance features	6.26397	6.38409	6.28376	5.19254	N/A
2 model	6.42320	6.54786	6.47963	5.28060	5.4613
2 model with imbalance features	6.39075	6.51506	6.43437	5.26175	5.4391

Table 3: Summary comparison of the performance of the models for different feature sets

5.3 Exploring Hyperparameter Tuning

An experiment with Hyperparameter tuning was carried out using the Optuna library with reference to one of the online guides (T., 2023).

Selected Hyperparameter for tuning are as follows:

- n_estimators (default=100) – set to 10,000, controls the number of decision trees, a higher number may result in overfitting and longer training time. Often tuned together with the learning rate.
- learning_rate (default=0.1) - range between 0.01 – 0.3, controls the learning speed. A smaller number leads to a slower learning rate, and the need to use early stopping rounds to terminate the training early to avoid excessive long training duration.
- num_leaves (default=31) range between 20 – 3000, controls the number of decision leaves in a single tree. Range limit also depends on the max_depth and should be 2^(max_depth) according to LGBM documentation.
- max_depth(default=-1) – range between 3 – 12, control the level of the tree. Lower numbers may lead to underfitting, higher numbers may lead to overfitting.
- min_data_in_leaf(default=20) – range between 200 – 10000, specifies the minimum number of observations that fit the decision criteria in a leaf. Lower numbers may lead to overfitting.
- max_bin(default=255) – range between 63 – 256. Smaller values increase the training speed while larger values increase the accuracy (Bahmani, 2023). For GPU training max_bin limit is 256.
- min_gain_to_split(default=0) – range between 0 – 15. Similar to XGBoosts's gamma. Can be used as extra regularization in large parameter grids.

Figure 30 shows the best hyperparameter setting obtained from the first run. **Figure 31 & Figure 32** shows the Optimization history and hyperparameter importance. As can be seen from **Table 4** the model trained with tuned parameters performs slightly better with better average & best training and validation scores, as well as test score upon submission.

```
        Best value (mse): 5.17973
        Best params:
                n_estimators: 10000
                learning_rate: 0.09276507151247129
                num_leaves: 900
                max_depth: 5
                min_data_in_leaf: 3100
                max_bin: 69
                min_gain_to_split: 4.133698816916528
<ipython-input-7-74515d8d1afc>:6: ExperimentalWarning: plot_o
  plot_optimization_history(study)
<ipython-input-7-74515d8d1afc>:7: ExperimentalWarning: plot_p
  plot_param_importances(study)
```

Figure 30: Best hyperparameters setting for first tuning (note optimization goal is minimum validation score achieved)

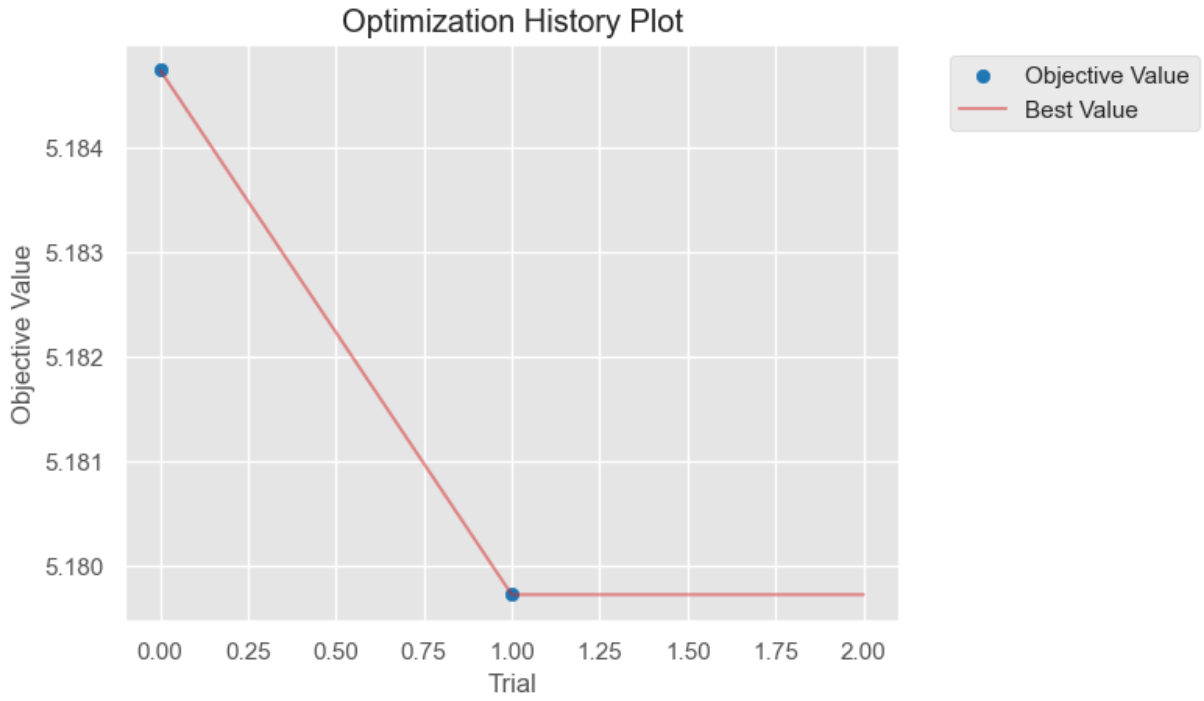

Figure 31: Optimization History Plot for first tuning

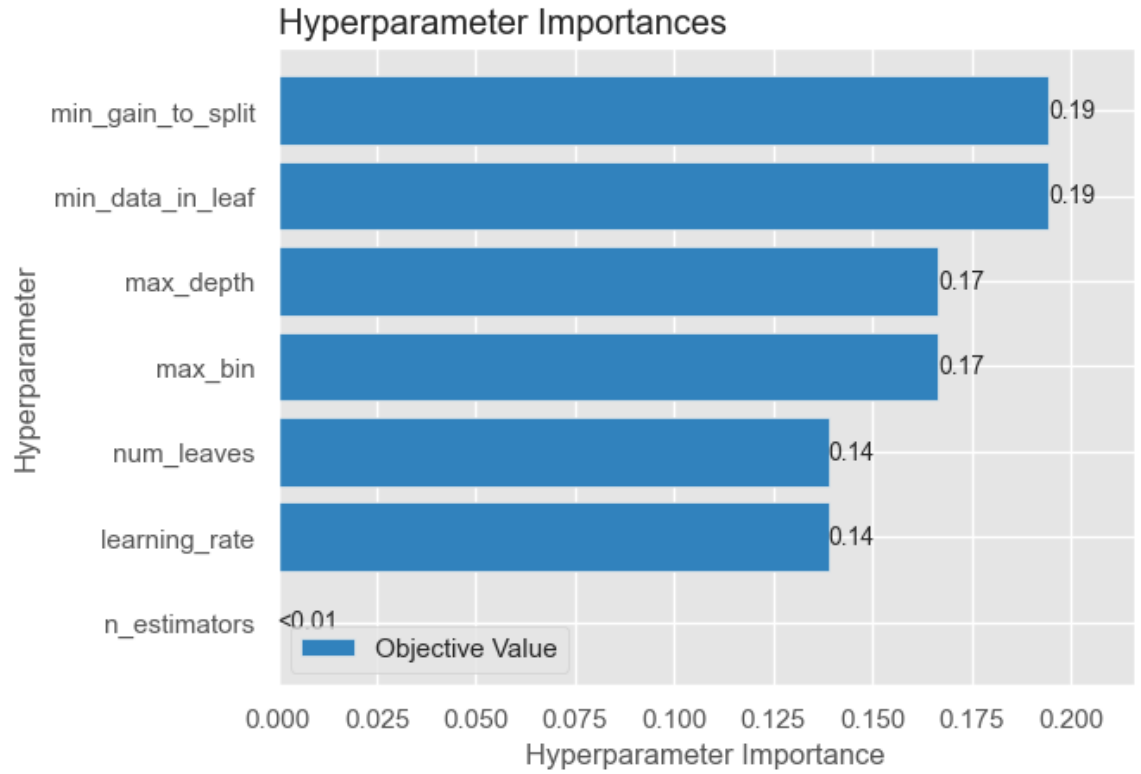

Figure 32: Hyperparameter Importance ranking for first tuning.

The second run included all parameter choices from the first run with the following additions/modifications:

- subsample (or bagging_fraction, default=1) – specify the percentage of rows used per tree building iteration. Improved generalization but also speed of training (Bahmani, 2023).
- Change the n_estimators to 1000
- Modify the range for num_leaves to be within 2^(max_depth-1) – 2^(max_depth)
- Modify the range for max_bin to between 32 and 256.

The best hyperparameter settings obtained are shown in **Figure 33**. As shown in **Table 4**, the average training scores and validation scores improved compared to the baseline, but the submission test score is slightly worse.

```
Best value (mse): 6.27053
Best params:
        max_depth: 11
        n_estimators: 1000
        learning_rate: 0.02617472280886847
        num_leaves: 1920
        min_data_in_leaf: 9300
        max_bin: 108
        min_gain_to_split: 5.473402167326732
        subsample: 0.6052347144775843
```

Figure 33: Best hyperparameters setting for second tuning (note optimization goal is minimum average validation score achieved)

The third run included all parameter choices from the second run with only one modification:

- Change the n_estimators setting from the default value (1000) to a suggestion range between 50 to 500 with a step of 50. As higher n_estimators increased the training time required significantly with a potential risk of overfitting, this attempt is to find the n_estimator that allows the model to train fast and generalize better.

The best hyperparameter settings obtained are shown in **Figure 34**. As shown in **Table 4**, the average training scores and validation scores improved compared to the baseline, and the submission test score is the best achieved thus far.

```
Best value (mse): 6.27026
Best params:
        max_depth: 9
        n_estimators: 150
        learning_rate: 0.06772311913826479
        num_leaves: 480
        min_data_in_leaf: 4100
        max_bin: 78
        min_gain_to_split: 10.128856386646307
        subsample: 0.9138045192239712
```

Figure 34: Best hyperparameters setting for third tuning (note optimization goal is minimum average validation score achieved)

Test Title	Training Score		Validation Score		Test Score (Submission)
	Average	Best Validate	Average	Best	
Baseline (with imbalance factors)	6.26616	6.38593	6.28535	5.1937	5.39
First Tuning	6.19929	6.31821	6.27048	5.1798	5.3823
Second Tuning	6.22611	6.34055	6.27049	5.1807	5.3906
Third Tuning	6.23874	6.35838	6.27021	5.1807	5.3785

Table 4: Summary comparison of the performance of the models for different hyperparameter tuning trial

6. Discussion and Conclusion

Based on the experiments we conducted with the XGBoost and LightGBM models, we can conclude that

- Applying StandardScaler does not bring significant improvements to the results.
- Applying the PCA() may yield improved results within certain processes, but is unlikely to be repeatable with actual test submission.
- Removing less important features may lead to worse performance of the models
- Adding first and second derivatives features resulted in minor improvements in the performance, due to difficulty in implementing it with the test data, it was not applied in the later experiments.
- Adding the imbalanced features helped improve the performance of the model.
- Hyperparameter tuning can yield better performance compared to an untuned base model.

Table 5 shows the comparison of the best results between the XGBoost and LightGBM models. It can be concluded that the LightGBM model performs better compared to the XGBoost model. We also achieved the ambitious goal set in our SMART target which is to have the MAE of less than 6.

Test Title	Training Score		Validation Score		Test Score (Submission)
	Average	Best Validate	Average	Best	
Best XGBoost	6.31042	6.30532	N/A	N/A	5.4142
Best LightGBM	6.23874	6.35838	6.27021	5.1807	5.3785

Table 5: Comparison of best performance from each model

However, notice that 99% of the target value is distributed around the range of -30 to 30 with a larger portion concentrated around the range of -10 to 10, an MAE of about 5.38 is not very good in predicting the target value. This reflects the nature that all the features available have weak correlations with the target value as discovered during the correlation analysis.

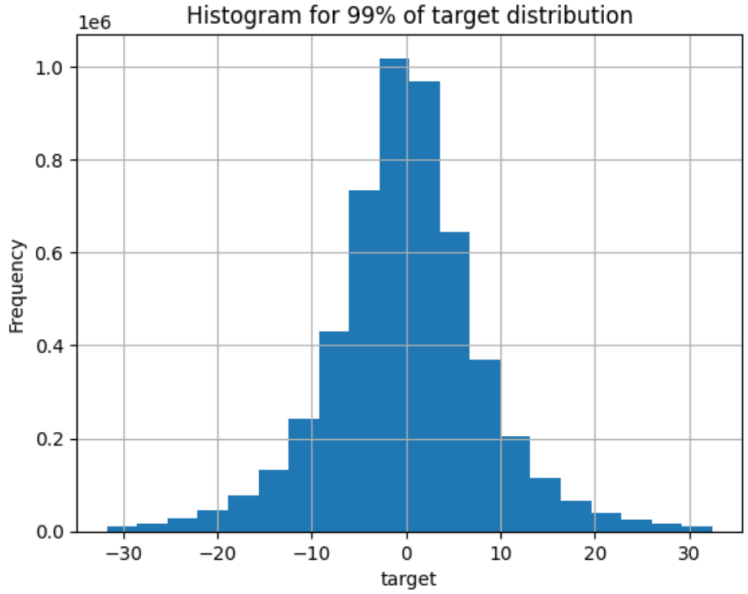

Figure 35: Histogram distribution for 99% of the target value

We believe we have tried most of the optimization techniques and this score is likely the best score that we can achieve at this moment. For reference the best score currently achieved on Kaggle is about 5.32.

For further improvement idea, there is one more technique to try on. This is to separate data of extreme values (extreme sizes for big companies) and the rest and use separate models to predict the target. Unfortunately, as the due date is approaching, this can only be tried out after the report submission.

7. Reference

1. Forbes, T., Macgillivray, J., Pietrobon, M., Dane, S., & Demkin, M. (2023). Optiver - Trading at the Close. https://kaggle.com/competitions/optiver-trading-at-the-close
2. S. Ravikumar and P. Saraf, "Prediction of Stock Prices using Machine Learning (Regression, Classification) Algorithms," 2020 International Conference for Emerging Technology (INCET), Belgaum, India, 2020, pp. 1-5, doi: 10.1109/INCET49848.2020.9154061.
3. Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, Tie-Yan Liu (2017). "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". Advances in Neural Information Processing Systems 30 (NIPS 2017), pp. 3149-3157.
4. Saha, S. (2023, August 8). XGBoost vs lightgbm: How are they different. neptune.ai. https://neptune.ai/blog/xgboost-vs-lightgbm
5. Chen, Tianqi; Guestrin, Carlos (2016). XGBoost. Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. https://doi.org/10.1145/2939672.2939785
6. Filho, M. (2023, July 12). How to do time series cross-validation in Python. Forecastegy (Alt + H). https://forecastegy.com/posts/time-series-cross-validation-python/
7. zhezhou, Y. (2023, September 23). 🥇🥇baseline LGB, xgb and catboost🥇🥇. Kaggle. https://www.kaggle.com/code/yuanzhezhou/baseline-lgb-xgb-and-catboost/notebook
8. T., B. (2023, April 8). Kaggler's Guide to lightgbm hyperparameter tuning with optuna in 2021. Medium. https://towardsdatascience.com/kagglers-guide-to-lightgbm-hyperparameter-tuning-with-optuna-in-2021-ed048d9838b5
9. Bahmani, M. (2023, September 4). Understanding LIGHTGBM parameters (and how to tune them). neptune.ai. https://neptune.ai/blog/lightgbm-parameters-guide