DATA3888: Predicting Stock Volatility

Ayush Singh, Tobit Louis, Kylie Haryono, Christy Lee, Zichun Han

2025-05-24

Table of contents

1 Executive Summary

1.1 Research Question

Short-term volatility forecasting is critical for high-frequency trading, risk management, and market-making. This study investigates whether advanced deep learning architectures can enhance predictive accuracy using ultra-high-frequency Level-2 Limit Order Book (LOB) data. Specifically, we evaluate whether a Transformer-based model can outperform traditional Long Short-Term Memory (LSTM) networks when both models are trained on identically preprocessed datasets with equivalent engineered features. The goal is to determine if the Transformer's attention mechanism offers a material advantage in capturing complex microstructural dynamics inherent in LOB data.

1.1.1 Hypothesis

1.1.1.1 Null Hypothesis (H)

Given equivalent data quality, feature engineering, and preprocessing conditions, a Transformer-based model does not outperform LSTM networks in short-term volatility forecasting on Level-2 Limit Order Book (LOB) data.

1.1.1.2 Alternative Hypothesis (H)

Given equivalent data quality, feature engineering, and preprocessing conditions, a Transformer-based model outperforms LSTM networks in short-term volatility forecasting on Level-2 Limit Order Book (LOB) data.

1.2 Key Findings

Our analysis shows that the Transformer model achieves superior performance in short-term volatility forecasting, consistently outperforming over 15 candidate models. In general, it attains an **out-of-sample R^2 of 0.62** and **QLIKE loss of 0.14**, capturing 62% of realized volatility variation—substantially higher than the WLS and Random Forest baselines, and measurably better than the LSTM. In favorable market regimes, performance improves to **out-of-sample R^2 of 0.78** and **QLIKE loss of 0.05**. The Transformer's attention mechanism effectively identifies temporally localized predictive structures in high-frequency order flow, yielding robust and highly accurate forecasts.

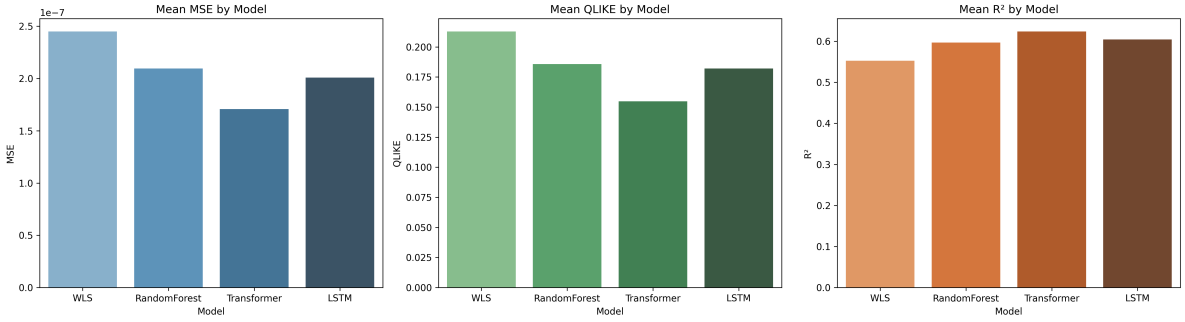

Figure 1: Bar plots of Mean MSE, QLIKE, and R^2 by Model

1.3 Relevance

The demonstrated predictive accuracy of the Transformer model has direct implications for high-frequency trading, risk management, and option pricing. High-quality real-time volatility estimates support tighter bid-ask spreads, more efficient hedging, and improved capital deployment under microsecond constraints. By capturing core properties of stochastic volatility—namely clustering, persistence, and asymmetry—the Transformer enables materially better real-time decision-making in trading and portfolio optimization contexts.

2 EDA

The Exploratory Data Analysis (EDA) stage involves data quality assessment and feature engineering, which were key to processing the dataset for time-series modelling. High frequency financial data may present challenges such as irregular sampling, missing observations, and noise that must be first addressed.

Considering the consecutive nature of the time series data, it is essential to keep missing data to a minimum and dealing with it systematically. Time IDs with at least 70% missing seconds are taken out of the used dataset. Bar charts were also used to visualise the distribution of missing percentages across time buckets, missing seconds in bucket per time ID, and missing seconds across time buckets.

Little's Missing Completely At Random (MCAR) Test was applied to check for patterns in missing data, and consequently whether imputation was appropriate. The result showed a high p-value of 1.00, indicating that the missing data is indeed MCAR and not dependent on other variables. We followed up by imputing low missing time intervals along with removing high missing time intervals to maintain the integrity of the dataset and utilising all available data that is reliable.

2.1 Feature Engineering

During the pre-cleaning stage, corrupt or out-of-range LOB records are removed. This includes (1) non-positive prices and sizes, (2) crossed quotes (ask $<$ bid) at both levels, (3) seconds_in_bucket outside of the expected range, and (4) duplicate or non-monotonic timestamps within each time_id. For the train-test split, 70-30 split was used, non-randomised to keep the consecutive nature of the data. After aggregating the features, the dataset is winsorised to reduce the impact of outliers. This scaling method is done by replacing the extreme values with less extreme values from the dataset.

The feature selection process uses two criteria: Pearson Correlation and Mutual Information. Pearson correlation and mutual information are calculated between each feature and the target variable, with Pearson correlation measuring linear relationships between the variables while mutual information captures the non-linear aspects. The features are then selected if it meets either of two criteria: it has absolute Pearson correlation of higher than or equal to 0.10, or mutual information score higher or equal to 75th percentile.

2.2 Testing

In establishing a baseline, the naive prediction was made using the current realised volatility. It calculates the Root Mean Squared Error (RMSE) of this baseline, which was 0.92. This

represents how well you would do by simply assuming tomorrow's volatility is the same as today's.

There were several regression models trained and evaluated in the process. Ridge Regression had a test RMSE of 0.56, while Random Forest had a test RMSE of 0.55, and LightGBM and Histogram-based Gradient Boosting both had test RMSEs of 0.50 each. The decreasing RMSE values show increasingly better performance. The gradient boosting models (LightGBM and Histogram GBM) both had the best performance, reducing the prediction error by approximately 46% from the naive approach. This suggests that the features selected contain valuable predictive information about future volatility, and that tree-based ensembles performed better at capturing complex relationships in this data than linear models like Ridge Regression.

3 Model Selection

After developing over 15 models, we strategically selected the following four models for detailed analysis including two top-performing models—Transformer and LSTM, along with a baseline model demonstrating solid foundational performance – Weighted Least Squares (WLS), and a high-performing model within its complexity class – Random Forest, which consistently outperformed other models of comparable complexity.

Our first implementation was a lightweight Transformer encoder for volatility forecasting. This involved projecting each 30 step LOB sequence into a 64-dimensional embedding, then stacking two encoder blocks—each with four-head self-attention with a key/query dimension of 64, a 4× feed-forward expansion to 256 units, and both residual and layer-norm connections. After global average pooling, a single linear output head produces the forecast, totaling an estimate of 100, 000 parameters. Inputs and log-transformed targets were MinMax-scaled. Training used the Adam optimiser with a learning rate of 1×10^{3}, using Mean Squared Error (MSE) loss on a strict 80/10/10 chronological split, a batch size of 32, and early stopping with a patience of 15 over 50 epochs. The model required approximately 500 GPU-hours to train and delivered strong out-of-sample RMSE, R^2, and QLIKE performance.

For our next model, we built a two-layer stacked LSTM. with 64 units returning all 30 steps (~25 088 parameters), followed by 20 % dropout, then a 32-unit LSTM (~12 416 parameters) with another 20 % dropout—topped by a compact feed-forward head (16-node dense + single linear output, ~545 parameters), for ~38 k trainable weights. Inputs were min-max–scaled 30-step sequences of engineered LOB features and the target was log-transformed volatility. We enforced a strict chronological split (80 % train, 10 % val, 10 % test) to eliminate look-ahead bias, trained with Adam at a 1×10 learning rate, 128-sample batches, and early stopping (patience = 5) for up to 50 epochs. Out-of-sample RMSE, R^2, and QLIKE metrics demonstrate strong sequence learning and stable volatility forecasts.

For the next model, a two-layer stacked LSTM was built. The first layer consisted of 64 units returning all 30 steps, comprising approximately 25,088 parameters, followed by 20% dropout.

This was then followed by a 32-unit LSTM layer with around 12,416 parameters and another 20% dropout. This architecture was topped by a compact feed-forward head, consisting of a 16-node dense layer and a single linear output, contributing approximately 545 parameters, for a total of approximately 38,000 trainable weights. Inputs were MinMax-scaled 30-step sequences of engineered LOB features, and the target was log-transformed volatility. A strict chronological split of 80% for training, 10% for validation, and 10% for testing was enforced to eliminate look-ahead bias. Training was conducted using the Adam optimizer with a $1\times10-4$ learning rate, 128-sample batches, and early stopping with a patience of 5 for up to 50 epochs. Out-of-sample RMSE, R^2, and QLIKE metrics demonstrated strong sequence learning and stable volatility forecasts.

Next, we used an ensemble of 500 trees grown to full depth, sampling the square root of features at each split and enforcing a minimum of three samples per leaf. Bootstrap aggregation was used to decorrelate trees and reduce variance. The volatility target was log-transformed to stabilize its heavy-tailed distribution, and we partitioned the data chronologically—80% for training, 10% for validation, 10% for testing—to eliminate look-ahead bias. Training ran in parallel across all CPU cores for efficiency, and we monitored progress in real time. Out-of-sample performance was measured with root-mean-square error and QLIKE loss (with clipping), demonstrating the model's ability to capture complex nonlinear interactions in order-book dynamics.

Finally, we implemented a heteroskedasticity-aware WLS model in statsmodels to forecast 30-tick-ahead realized volatility. Observation weights were set as the inverse of a long-window rolling variance to counter time-varying noise. Using a chronological 80/20 train-test split across all 30 stocks, the model delivers a strong baseline out-of-sample R^2 and QLIKE loss.

4 Model Evaluation

4.1 Metrics Used

To evaluate model performance comprehensively, we used three complementary metrics:

- **R-squared (R^2)**: This measures how much of the variance in the target variable is explained by the model. A higher R^2 means the model's predictions align more closely with actual volatility, giving a good sense of overall fit.
- **QLIKE (Quasi-Likelihood) Loss**: Tailored for volatility forecasting, QLIKE penalises underestimates more heavily than overestimates. This is especially important in financial risk settings, where underpredicting volatility can lead to poor hedging or mispricing.
- **Mean Squared Error (MSE)**: MSE calculates the average squared difference between predictions and actual values. It's sensitive to large errors, making it a good measure of overall prediction accuracy.

Together, these metrics offer a balanced view:

- R^2 captures general trend fit,
- QLIKE evaluates calibration during volatility spikes,
- MSE reflects how the model handles large deviations.

This trio allows us to assess both statistical performance and practical utility in real-world forecasting.

4.2 Fair Comparison

To keep model comparisons fair, we used the same feature engineering pipeline (make_features) and preprocessing (including scaling) across all models. Input features were kept consistent, and target labels were log-transformed in the same way where applicable. All models were trained and tested on the same temporal splits of the data, using time_id as a session boundary. This session-based splitting preserves time-series structure and prevents leakage, allowing us to compare models under the exact same conditions. Final metrics were computed strictly on held-out test sessions, untouched during training and validation, ensuring that evaluation results reflect true generalisation performance rather than overfitting or tuning artefacts.

5 Key Findings

5.0.1 Transformer: Top Performance, Limited Transparency

The Transformer stood out across all metrics:

- Lowest MSE and QLIKE, indicating strong accuracy and reliability under volatile conditions.
- Highest R^2, showing excellent fit to actual trends.

Its self-attention mechanism enables the model to capture intricate temporal and nonlinear relationships in the limit order book (LOB). However, the downside is reduced interpretability, which can be a drawback in regulated or real-time settings.

5.0.2 LSTM: Stable and Sequentially Aware

LSTM followed closely behind, with:

- Slightly higher MSE than the Transformer,
- Comparable QLIKE and R^2.

This indicates that LSTM, while simpler, still models sequential LOB dynamics effectively. It's a strong candidate when computational resources are limited or model explainability is a concern.

5.0.3 Random Forest: Reliable Static Baseline

Random Forest performed well for a non-sequential model:

- Outperformed WLS across all metrics,
- Achieved decent QLIKE and R^2 scores.

Its main limitation is the inability to directly model temporal dependencies. However, with engineered time-lagged features, it offers good predictive power and strong explainability via feature importance—useful for real-time systems requiring quick insights.

5.0.4 WLS: Transparent but Weak

WLS lagged behind:

- Lowest R^2 and highest QLIKE,
- Moderate MSE but poor calibration for volatility.

Still, its simplicity and transparency make it a valuable baseline for benchmarking and for interpreting how features relate to outcomes.

5.1 Summary Table

Table 1: Comparison of Strengths and Weaknesses Across Models

Model	Strengths	Weaknesses
Transformer	Best accuracy; learns complex temporal features	High computational cost, low interpretability
LSTM	Strong sequence modeling; stable performance	Less accurate than Transformer; harder to interpret
Random Forest	Moderate accuracy; interpretable with feature importances	No inherent temporal modeling
WLS	Transparent; fast; strong baseline	Poor fit and calibration in volatile conditions

6 Further Discussion

6.1 Interpretability

Transformers have revolutionized sequence modeling by using self-attention to weigh every input position against every other, but this very strength makes them hard to interpret. In our volatility model, the lack of positional encodings and the use of global average pooling further obscure how the network arrives at its forecasts.

One natural way to open the "black box" is to visualize the attention weights themselves. By plotting attention maps for each head and layer, we can see which past time buckets the model emphasizes—revealing, for instance, whether it zeroes in on sudden price jumps or sustained order-flow imbalances. In parallel, model-agnostic tools such as LIME and SHAP can approximate the transformer's behavior around a single forecast, attributing portions of the output to each input feature. For PyTorch implementations, Captum provides built-in gradient- and perturbation-based attributions, letting us drill down into any layer—even the pooling step—to understand how information flows through the network [@Lakham2024].

Looking further ahead, the emerging field of mechanistic interpretability (MI) offers a more fundamental approach. As demonstrated by Rai et al. (2024), MI aims to transform transformer-based language models from opaque black boxes into transparent, controllable systems by reverse-engineering their internal computations—neurons, attention heads, and composite circuits—into human-understandable mechanisms [@rai2024practical]. In our context, MI could pinpoint exactly which attention head signals an imminent spike in volatility, enabling targeted fine-tuning, bias mitigation, or safety audits. Together, these methods—from attention visualization and LIME/SHAP explanations to cutting-edge MI—provide a clear roadmap for making transformer-based volatility models both powerful and transparent.

LSTM networks remain opaque despite their power to learn nonlinear temporal patterns, because their high-dimensional hidden states and feedback loops cannot be reduced to clear, human-readable drivers. As Firouzjaee and Khalilian show in oil-stock forecasting, even adding highly correlated inputs like crude oil, gold, or USD prices does not improve an LSTM's core interpretability, since feature augmentation alone cannot expose its internal decision logic [@Firouzjaee2024]. As a result, although LSTMs excel at prediction, they offer little insight into which factors truly drive their forecasts. Addressing this "black-box" nature requires specialized interpretability techniques—such as state-space analysis, attention-augmented RNNs, or hybrid models—to bridge accuracy and transparency in financial applications.

Random forests also sacrifice some interpretability compared to single decision trees, but they are not entirely "black box." By averaging many trees, they deliver strong predictive power while still offering ensemble-level diagnostics. Each tree's split logic remains fully inspectable, allowing you to trace the exact path for any prediction. Global measures like mean decrease in impurity and permutation importance rank features by overall influence, and partial dependence or ICE plots reveal how changes in key predictors affect outputs on average or for

individual instances. Out-of-bag estimates provide nearly unbiased error and importance metrics without separate validation. Together, these built-in tools make random forests far more interpretable than most deep "black-box" models, even if they don't match the simplicity of a single tree [@Grigg2019].

```
# Get feature importances
importances = rf.feature_importances_

# Sort feature importances in descending order
indices = np.argsort(importances)[::-1]

# Print the feature ranking
print("Feature ranking:")

for f in range(len(feature_cols_mod)):
    print(f"{f + 1}. feature {indices[f]} ({importances[indices[f]]}) - {feature_cols_mod[ind

# Plot the feature importances of the forest
plt.figure()
plt.title("Feature importances")
plt.bar(range(len(feature_cols_mod)), importances[indices], align="center")
plt.xticks(range(len(feature_cols_mod)), [feature_cols_mod[i] for i in indices], rotation=90
plt.xlim([-1, len(feature_cols_mod)])
plt.show()
```

Specifically to our project, the random forest indentified _____ as the most importance features.

6.2 Limitations

Although the transformer model outperforms our other approaches, it has several constraints that may limit its practical use. First, we restricted the context window to 30 time steps—this may miss important longer-range dependencies if volatility patterns span beyond a half-minute interval. Second, our implementation uses only two encoder layers and a modest embedding size (d_model = 64), which may be insufficient to capture intricate, multi-scale dynamics in high-frequency data. Third, transformers are prone to overfitting, and even with early stopping this risk remains if the validation split does not fully represent market regimes. Finally, the heavy computation required by self-attention often forces compromises in data breadth: we trained on only a subset of stocks and files. If these trade-offs prove unacceptable, Optiver may need to scale up hardware, prune or distill the model, or fall back to simpler architectures.

Our LSTM suffers from related challenges. While two layers of 64 and 32 units coupled with dropout can learn nonlinear temporal patterns, the same 30-step window may not align with true autocorrelation horizons. Without additional regularization or batch normalization, dropout alone may not prevent overfitting—especially when model complexity outpaces training data. And, like transformers, LSTMs offer limited visibility into their hidden-state dynamics.

Random forests sidestep many of these issues by fixing model complexity and avoiding sequence recursion, but they treat each bucket independently, failing to exploit the temporal order inherent in time series. This requires extensive feature engineering—lags, rolling statistics, seasonal indicators—to expose trends and autocorrelation. Furthermore, random forests assume stationarity and can still overfit if trees grow too deep, which is why we capped depth at 12.

Across all models, our use of MinMax scaling presumes that the distribution of prices remains constant between training and deployment. Because MinMaxScaler is sensitive to extreme values, any outliers in stock prices can distort feature ranges and destabilize predictions. More robust scaling methods or outlier-mitigation steps may be necessary to ensure consistency on new data.

6.3 Industry Context: Interpretability-Accuracy Trade-Off

Although Transformer- and LSTM-based models offered better accuracy, there is a trade-off between it and interpretability. WLS and Random Forest offer some feature transparency. Transformer is the best-performing, but also the most complex and opaque, with potential latency trade-offs. In practice, models with the same level of accuracy as Transformer would give Optiver a strong edge, if it could run with an acceptable level of latency.

The traditional assumption of a strict trade-off (more interpretability implies a lower level of accuracy) is oversimplified. A model with lower standalone accuracy (e.g. Random Forest, WLS) can outperform a more accurate but opaque model (e.g. Transformer) in collaborative human-machine settings, because it allows better interpretation and decision-making by the human. In a context like trading at Optiver, where humans interpret model output, collaborative performance matters more than standalone accuracy, as misinterpretation can be costly. Even high-performing models like Transformers can perform poorly in practice if humans can not understand their outputs.

7 Conclusion

This research was focused on the practical needs of market makers and trading firms like Optiver, where precise volatility forecasts are essential for pricing, hedging, and real-time inventory management. The Transformer-based model proved to be more accurate than the

other models, with the lowest MSE and QLIKE, and the highest R2 on test data. This shows its ability to capture complex patterns in volatility, aligning with findings that Transformers effectively model long-range dependencies in financial time series. Both LSTM and Transformer models showed relatively consistent performance across samples, demonstrating robustness under volatile market conditions.

While the Transformer model offers superior accuracy, its complexity and opacity present challenges for real-world trading deployment. In practice, models that balance accuracy with interpretability and latency considerations may provide firms like Optiver with a stronger edge, enabling more efficient and transparent decision-making processes.

8 References

8.1 Model Architecture

8.1.1 LSTM

Layer (type)	Output Shape	Param #
lstm_8 (LSTM)	(None, 30, 64)	25,088
dropout_8 (Dropout)	(None, 30, 64)	0
lstm_9 (LSTM)	(None, 32)	12,416
dropout_9 (Dropout)	(None, 32)	0
dense_8 (Dense)	(None, 16)	528
dense_9 (Dense)	(None, 1)	17

Total params: 38,049 (148.63 KB)

Trainable params: 38,049 (148.63 KB)

Non-trainable params: 0 (0.00 B)

8.1.2 Transformer

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 30, 23)	0	-
dense (Dense)	(None, 30, 64)	1,536	input_l ayer[0][0]

Layer (type)	**Output Shape**	**Param #**	**Connected to**
multi_ head_attention (Multi HeadAttention)	(None, 30, 64)	16,640	d ense[0][0] x3
add (Add)	(None, 30, 64)	0	de nse[0][0], multi_ head_attention
layer_norm (Layer Normalization)	(None, 30, 64)	128	add[0][0]
dense_1 (Dense)	(None, 30, 256)	16,640	layer_norm
dense_2 (Dense)	(None, 30, 64)	16,448	den se_1[0][0]
dropout_1 (Dropout)	(None, 30, 64)	0	den se_2[0][0]
add_1 (Add)	(None, 30, 64)	0	layer_norm, dropout_1
layer_norm_2 (Layer Normalization)	(None, 30, 64)	128	a dd_1[0][0]
multi_he ad_attention_2 (Multi HeadAttention)	(None, 30, 64)	16,640	layer_norm_2 x3
add_2 (Add)	(None, 30, 64)	0	layer_norm_2, multi_he ad_attention_2
layer_norm_3 (Layer Normalization)	(None, 30, 64)	128	a dd_2[0][0]
dense_3 (Dense)	(None, 30, 256)	16,640	layer_norm_3
dense_4 (Dense)	(None, 30, 64)	16,448	den se_3[0][0]
dropout_3 (Dropout)	(None, 30, 64)	0	den se_4[0][0]
add_3 (Add)	(None, 30, 64)	0	layer_norm_3, dropout_3
layer_norm_4 (Layer Normalization)	(None, 30, 64)	128	a dd_3[0][0]
g lobal_avg_pool (GlobalAve ragePooling1D)	(None, 64)	0	layer_norm_4
dropout_4 (Dropout)	(None, 64)	0	g lobal_avg_pool

Layer (type)	Output Shape	Param #	Connected to
dense_5 (Dense)	(None, 1)	65	dropo ut_4[0][0]

Total params: 101,569 (396.75 KB)

Trainable params: 101,569 (396.75 KB)

Non-trainable params: 0 (0.00 B)

8.2 Code

8.2.1 Imports

```
# Core libraries
import os
import random
import warnings

# Numerical and data handling
import numpy as np
import pandas as pd
import polars as pl

# Visualization
import matplotlib.pyplot as plt

# File handling
from glob import glob

# Preprocessing and feature selection
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.feature_selection import VarianceThreshold, mutual_info_regression

# Classical models
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor

# Unsupervised learning
from sklearn.cluster import KMeans
```

```
# Evaluation metrics
from sklearn.metrics import r2_score, mean_squared_error, root_mean_squared_error
from scipy.stats import skew, pearsonr

# Statistical modeling
import statsmodels.api as sm

# Deep learning (TensorFlow/Keras)
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, models, callbacks
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
```

8.2.2 Warnings

```
warnings.filterwarnings("ignore", category=RuntimeWarning)
warnings.filterwarnings("ignore", category=DeprecationWarning)
```

8.2.3 Combining Raw Data

```
# Get sorted list of all CSV file paths in the directory
csv_files = sorted(glob("Data/individual_book_train/*.csv"))

# Define column data types
schema = {
    'time_id': pl.Int32,
    'seconds_in_bucket': pl.Int32,
    'bid_price1': pl.Float32,
    'ask_price1': pl.Float32,
    'bid_price2': pl.Float32,
    'ask_price2': pl.Float32,
    'bid_size1': pl.Int32,
    'ask_size1': pl.Int32,
    'bid_size2': pl.Int32,
    'ask_size2': pl.Int32,
    'stock_id': pl.Int32,
```

```
}

# Lazily scan CSVs with predefined schema (no type inference)
ldf = pl.scan_csv(
    csv_files,
    schema_overrides=schema,
    infer_schema_length=0
)

# Load into memory
df = ldf.collect()

# Write to Parquet with Snappy compression
df.write_parquet("Data/112Stocks.parquet", compression="snappy")
```

8.2.4 Global Parameters

```
# reproducibility
RANDOM_STATE = 42
# outlier filtering
VOLATILITY_IQR_MULTIPLIER = 1.5
# clustering
N_CLUSTERS = 5
# modeling threshold
MIN_PERIODS_FOR_MODEL = 10
# metric weights
R2_WEIGHT = 0.5
QLIKE_WEIGHT = 0.5
# numerical stability
EPSILON = 1e-12
# model input length
SEQ_LEN = 30
```

8.2.5 Helper Functions

```
def calculate_basic_features_snapshot(df_slice):
    features = pd.DataFrame(index=df_slice.index)
    # micro price (weighted mid-price)
```

```
    features['micro_price'] = (df_slice['bid_price1'] * df_slice['ask_size1'] + \
                               df_slice['ask_price1'] * df_slice['bid_size1']) / \
                              (df_slice['bid_size1'] + df_slice['ask_size1'] + EPSILON)
    # fallback to mid-price if NaN
    features['micro_price'] = features['micro_price'].fillna((df_slice['bid_price1'] + df_sli
    # top-of-book spreads
    features['spread1'] = df_slice['ask_price1'] - df_slice['bid_price1']
    features['spread2'] = df_slice['ask_price2'] - df_slice['bid_price2']
    # size imbalance at level 1
    features['imbalance_size1'] = (df_slice['bid_size1'] - df_slice['ask_size1']) / \
                                  (df_slice['bid_size1'] + df_slice['ask_size1'] + EPSILON)
    # aggregated book pressure (level 1 + 2)
    sum_bid_sizes = df_slice['bid_size1'] + df_slice['bid_size2']
    sum_ask_sizes = df_slice['ask_size1'] + df_slice['ask_size2']
    features['book_pressure'] = sum_bid_sizes / (sum_bid_sizes + sum_ask_sizes + EPSILON)
    return features
```

```
def calculate_time_id_features(df_group):
    df_group = df_group.sort_values('seconds_in_bucket').copy()
    snapshot_features = calculate_basic_features_snapshot(df_group)

    # log returns of micro price
    log_returns = np.log(snapshot_features['micro_price'] / snapshot_features['micro_price']
    log_returns = log_returns.replace([np.inf, -np.inf], np.nan).dropna()

    results = {}
    # volatility and skewness
    results['realized_volatility'] = np.std(log_returns) if len(log_returns) > 1 else 0
    results['realized_skewness'] = skew(log_returns) if len(log_returns) > 1 else 0
    # autocorrelation of log returns
    if len(log_returns) > 2:
        ac, _ = pearsonr(log_returns.iloc[1:], log_returns.iloc[:-1])
        results['autocorrelation_log_returns'] = ac if not np.isnan(ac) else 0
    else:
        results['autocorrelation_log_returns'] = 0

    # basic aggregated features
    results['tick_frequency'] = len(df_group)
    results['mean_micro_price'] = snapshot_features['micro_price'].mean()
    results['mean_spread1'] = snapshot_features['spread1'].mean()
    results['mean_spread2'] = snapshot_features['spread2'].mean()
    results['mean_imbalance_size1'] = snapshot_features['imbalance_size1'].mean()
```

```
    results['mean_book_pressure'] = snapshot_features['book_pressure'].mean()
    results['mean_bid_size1'] = df_group['bid_size1'].mean()
    results['mean_ask_size1'] = df_group['ask_size1'].mean()
    results['mean_bid_size2'] = df_group['bid_size2'].mean()
    results['mean_ask_size2'] = df_group['ask_size2'].mean()

    return pd.Series(results)
```

```
def qlike_loss(y_true, y_pred):
    # avoid division/log(0)
    y_pred = np.maximum(y_pred, EPSILON)
    y_true = np.maximum(y_true, 0)
    valid_indices = (y_true > EPSILON)
    if not np.any(valid_indices):
        return np.nan
    # compute QLIKE loss on valid data
    y_true_f = y_true[valid_indices]
    y_pred_f = y_pred[valid_indices]
    y_pred_f = np.maximum(y_pred_f, EPSILON)
    loss = np.mean(y_true_f / y_pred_f - np.log(y_true_f / y_pred_f) - 1)
    return loss
```

8.2.6 Feature Generation

```
print("Calculating features per stock_id and time_id.")
stock_time_id_features = df.groupby(['stock_id', 'time_id']).apply(calculate_time_id_features
print(f"Calculated detailed features for {stock_time_id_features.shape[0]} stock/time_id pai
print(stock_time_id_features.head())
```

8.2.7 Filtering

```
# compute mean realized volatility per stock
overall_stock_mean_rv = stock_time_id_features.groupby('stock_id')['realized_volatility'].mea
overall_stock_mean_rv = overall_stock_mean_rv.rename(columns={'realized_volatility': 'mean_re

# define IQR bounds
q1 = overall_stock_mean_rv['mean_realized_volatility'].quantile(0.25)
q3 = overall_stock_mean_rv['mean_realized_volatility'].quantile(0.75)
```

```
iqr = q3 - q1
lower_bound = q1 - VOLATILITY_IQR_MULTIPLIER * iqr
upper_bound = q3 + VOLATILITY_IQR_MULTIPLIER * iqr

# filter stocks within bounds and above tiny volatility threshold
epsilon_vol = 1e-7
filtered_stocks_info = overall_stock_mean_rv[
    (overall_stock_mean_rv['mean_realized_volatility'] >= lower_bound) &
    (overall_stock_mean_rv['mean_realized_volatility'] <= upper_bound) &
    (overall_stock_mean_rv['mean_realized_volatility'] > epsilon_vol)
]

# report filtering outcome
n_original_stocks = df['stock_id'].nunique()
n_filtered_stocks = filtered_stocks_info['stock_id'].nunique()
print(f"Original number of stocks: {n_original_stocks}")
print(f"Number of stocks after volatility filtering: {n_filtered_stocks}")

if n_filtered_stocks == 0:
    print("Error: No stocks remaining after filtering. Adjust VOLATILITY_IQR_MULTIPLIER or cl
```

8.2.8 K-means Clustering

```
stock_time_id_features_filtered = stock_time_id_features[
    stock_time_id_features['stock_id'].isin(filtered_stocks_info['stock_id'])
]

# selected features for clustering
cluster_feature_cols = [
    'realized_volatility', 'realized_skewness', 'autocorrelation_log_returns',
    'tick_frequency', 'mean_micro_price', 'mean_spread1', 'mean_spread2',
    'mean_imbalance_size1', 'mean_book_pressure',
    'mean_bid_size1', 'mean_ask_size1', 'mean_bid_size2', 'mean_ask_size2'
]

# aggregate features at stock level
stock_meta_features_df = stock_time_id_features_filtered.groupby('stock_id')[cluster_feature_

print("Meta-features for clustering (mean of time_id features per stock):")
print(stock_meta_features_df.head())
```

```
# clustering stocks based on meta-features
scaler = StandardScaler()
scaled_meta_features = scaler.fit_transform(stock_meta_features_df)

print(f"\nPerforming K-means clustering with K={N_CLUSTERS}...")
kmeans = KMeans(n_clusters=N_CLUSTERS, random_state=RANDOM_STATE, n_init='auto')
stock_meta_features_df['cluster'] = kmeans.fit_predict(scaled_meta_features)

print("Clustering results (stock_id and assigned cluster):")
print(stock_meta_features_df[['cluster']].head())
```

8.2.9 Individual R^2 and QLIKE evaluation

```
r_squared_feature_cols = [
    'realized_volatility', 'mean_spread1', 'mean_imbalance_size1',
    'mean_book_pressure', 'mean_micro_price'
]
stock_scores_list = []
stock_time_id_features_filtered = stock_time_id_features_filtered.sort_values(['stock_id', '

for stock_id in filtered_stocks_info['stock_id']:
    stock_data = stock_time_id_features_filtered[stock_time_id_features_filtered['stock_id']

    if len(stock_data) < MIN_PERIODS_FOR_MODEL:
        print(f"Stock {stock_id}: Insufficient data ({len(stock_data)} periods) for R2/QLIKE
        stock_scores_list.append({'stock_id': stock_id, 'r_squared': np.nan, 'qlike': np.nan
        continue

    for col in r_squared_feature_cols:
        stock_data[f'prev_{col}'] = stock_data[col].shift(1)

    stock_data = stock_data.dropna()

    if len(stock_data) < 2:
        print(f"Stock {stock_id}: Insufficient data after lagging for R2/QLIKE, skipping.")
        stock_scores_list.append({'stock_id': stock_id, 'r_squared': np.nan, 'qlike': np.nan
        continue

    y_true_r2_all = []
    y_pred_r2_all = []
```

```
    y_true_qlike_all = []
    y_pred_qlike_all = []

    start_prediction_idx = max(2, MIN_PERIODS_FOR_MODEL // 2)

    for i in range(start_prediction_idx, len(stock_data)):
        train_df = stock_data.iloc[:i]
        current_period_data = stock_data.iloc[i]

        X_train = train_df[[f'prev_{col}' for col in r_squared_feature_cols]]
        y_train = train_df['realized_volatility']

        X_current = pd.DataFrame(current_period_data[[f'prev_{col}' for col in r_squared_feat
        y_current_true_r2 = current_period_data['realized_volatility']

        if len(X_train) >= 2:
            try:
                model = LinearRegression()
                model.fit(X_train, y_train)
                y_current_pred_r2 = model.predict(X_current)[0]

                y_true_r2_all.append(y_current_true_r2)
                y_pred_r2_all.append(y_current_pred_r2)
            except Exception:
                pass

        historical_rv_for_qlike = train_df['realized_volatility']
        if not historical_rv_for_qlike.empty:
            forecast_rv_qlike = historical_rv_for_qlike.mean()
            y_current_true_qlike = current_period_data['realized_volatility']

            y_true_qlike_all.append(y_current_true_qlike)
            y_pred_qlike_all.append(forecast_rv_qlike)

    r_squared_stock = np.nan
    if len(y_true_r2_all) >= 2 and len(set(y_true_r2_all)) > 1:
        r_squared_stock = r2_score(y_true_r2_all, y_pred_r2_all)

    qlike_stock = np.nan
    if y_true_qlike_all:
        qlike_stock = qlike_loss(np.array(y_true_qlike_all), np.array(y_pred_qlike_all))
```

```
    stock_scores_list.append({
        'stock_id': stock_id,
        'r_squared': r_squared_stock,
        'qlike': qlike_stock
    })
    print(f"Stock {stock_id}: R^2 = {r_squared_stock:.4f}, QLIKE = {qlike_stock:.4f} (from {
```

```
stock_scores_df = pd.DataFrame(stock_scores_list)
stock_scores_df = pd.merge(stock_scores_df, stock_meta_features_df[['cluster']].reset_index()
print("\nCalculated R-squared and QLIKE scores:")
print(stock_scores_df.head())
```

```
# remove incomplete results
stock_scores_df = stock_scores_df.dropna(subset=['r_squared', 'qlike'])

if stock_scores_df.empty:
    print("Error: No stocks remaining after calculating R-squared/QLIKE (all NaNs or empty).
    exit()

# combine scores using weighted formula
stock_scores_df['combined_score'] = R2_WEIGHT * stock_scores_df['r_squared'] - \
                                    QLIKE_WEIGHT * stock_scores_df['qlike']
# rank and select top stocks
top_stocks = stock_scores_df.sort_values(by='combined_score', ascending=False)

print(f"\nTop 30 stocks based on combined score ({R2_WEIGHT}*R^2 - {QLIKE_WEIGHT}*QLIKE):")
N_TOP_STOCKS = 30
final_selection = top_stocks.head(N_TOP_STOCKS)
print(final_selection)
```

8.2.10 Saving Selected Stocks

```
# For Reference only, choose from final_selection in practice.
selected_stock_ids = [1, 5, 7, 8, 22, 27, 32, 44, 50, 55,
                      59, 62, 63, 73, 75, 76, 78, 80, 81, 84,
                      85, 86, 89, 96, 97, 101, 102, 109, 115, 120]

df = pd.read_parquet("Data/112Stocks.parquet")
df = df[df["stock_id"].isin(selected_stock_ids)]
df.to_parquet("Data/30Stocks.parquet")
```

8.2.11 Feature Engineering

```
# Only for top 30 stocks
df = pd.read_parquet("Data/30stocks.parquet")

def make_features(df: pd.DataFrame) -> pd.DataFrame:
    df = df.copy()

    df['mid_price'] = (df['bid_price1'] + df['ask_price1']) / 2
    df['spread']    = df['ask_price1'] - df['bid_price1']

    with np.errstate(divide='ignore', invalid='ignore'):
        num  = df['bid_size1'] - df['ask_size1']
        den  = df['bid_size1'] + df['ask_size1']
        df['imbalance'] = np.where(den > 0, num / den, np.nan)

        num2 = (df['bid_size1'] + df['bid_size2']) - (df['ask_size1'] + df['ask_size2'])
        den2 = df[['bid_size1','bid_size2','ask_size1','ask_size2']].sum(axis=1)
        df['book_pressure'] = np.where(den2 > 0, num2 / den2, np.nan)

    df['normalized_spread'] = df['spread'] / df['mid_price'].replace(0, np.nan)
    df['OBI_L2'] = np.where(den2 > 0, (df['bid_size1'] + df['bid_size2']) / den2, np.nan)

    sizes = df[['bid_size1','bid_size2','ask_size1','ask_size2']].astype(float).values
    total = sizes.sum(axis=1, keepdims=True)
    p = np.divide(sizes, total, where=total != 0)
    entropy = -np.nansum(np.where(p > 0, p * np.log(p), 0), axis=1)
    df['LOB_entropy'] = entropy
    df['LOB_entropy_normalized'] = entropy / np.log(4)

    df['log_return'] = (
        df.groupby('time_id')['mid_price']
          .transform(lambda x: np.log(x / x.shift(1)))
    )

    df['realized_volatility'] = (
        df.groupby('time_id')['log_return']
        .transform(lambda x: np.sqrt(
            ((x.shift(1) ** 2)
                .rolling(30, min_periods=1)
                .sum()
            ).clip(lower=0)
```

```
        ))
    )

    df['rv_future'] = (
        df.groupby('time_id')['realized_volatility'].shift(-30)
    )

    df['bipower_var'] = (
        df.groupby('time_id')['log_return']
          .transform(lambda x: x.abs().shift(1)
                       .rolling(2, min_periods=1)
                       .apply(lambda r: r[0] * r[1], raw=True)
                       .rolling(30, min_periods=1)
                       .mean())
    )

    df['wap'] = (
        (df['bid_price1'] * df['ask_size1'] + df['ask_price1'] * df['bid_size1']) /
        (df['bid_size1'] + df['ask_size1']).replace(0, np.nan)
    )

    df['log_wap_return'] = (
        df.groupby('time_id')['wap']
          .transform(lambda x: np.log(x / x.shift(1)))
    )

    for col in ['imbalance', 'book_pressure', 'log_return']:
        df[f'{col}_lag1'] = df.groupby('time_id')[col].shift(1)
        df[f'{col}_lag2'] = df.groupby('time_id')[col].shift(2)

    df['rolling_vol_30'] = (
        df.groupby('time_id')['log_return']
          .transform(lambda x: x.shift(1).rolling(30, min_periods=1).std())
    )
    df['rolling_imbalance_mean_30'] = (
        df.groupby('time_id')['imbalance']
          .transform(lambda x: x.shift(1).rolling(30, min_periods=1).mean())
    )

    df = df.dropna()
    df = df.replace([np.inf, -np.inf], np.nan)
```

```
    theta = 2 * np.pi * df['seconds_in_bucket'] / 600 # period = 600
    df['sec_sin'] = np.sin(theta)
    df['sec_cos'] = np.cos(theta)

    for c in ['bid_size1','ask_size1','bid_size2','ask_size2']:
        df[c + '_log'] = np.log1p(df[c])
        df.drop(columns=c, inplace=True)

    return df

df = make_features(df)
```

8.2.12 Type Conversion

```
df = df.astype({col: 'float32' if df[col].dtype == 'float64' else 'int32'
                for col in df.columns
                if df[col].dtype in ['float64', 'int64']})
```

8.2.13 Variance Thresholding

```
X = df.drop(columns=['rv_future'])
```

```
selector = VarianceThreshold(threshold=0.0)
X_reduced = selector.fit_transform(X)
selected_columns = X.columns[selector.get_support()]
X_reduced_df = pd.DataFrame(X_reduced, columns=selected_columns, index=X.index)
```

```
dfR = X_reduced_df.astype({col: 'float32' if X_reduced_df[col].dtype == 'float64' else 'int3
                for col in X_reduced_df.columns
                if X_reduced_df[col].dtype in ['float64', 'int64']})
```

8.2.14 Spearman Correlation

```
corr = dfR.corr(method='spearman').abs()
to_drop = {c for c in corr.columns for r in corr.columns
```

```
if r != c and corr.loc[r, c] > .98 and corr.loc[r].sum() < corr.loc[c].sum()}
to_drop
```

```
dfR = dfR.drop(columns=list(to_drop))

dfR['rv_future'] = df['rv_future']

dfR = dfR.sort_values(
    ['stock_id', 'time_id', 'seconds_in_bucket'],
    ascending=[True, True, True]
).reset_index(drop=True)

dfR.to_parquet("Data/FE30Stocks.parquet")
```

8.2.15 Weighted Least Square

```
def qlike_loss(actual, pred, eps=1e-12):
    a = np.clip(actual, eps, None)
    f = np.clip(pred,   eps, None)
    r = a / f
    return np.mean(r - np.log(r) - 1.0)
```

```
# Feature and target column names
feature_cols = ['stock_id','mid_price', 'spread', 'imbalance',
       'book_pressure', 'LOB_entropy', 'log_return', 'bipower_var',
       'log_wap_return', 'imbalance_lag1', 'imbalance_lag2',
       'book_pressure_lag1', 'book_pressure_lag2', 'log_return_lag1',
       'log_return_lag2', 'rolling_vol_30', 'rolling_imbalance_mean_30',
       'sec_sin', 'sec_cos', 'bid_size1_log', 'ask_size1_log', 'bid_size2_log',
       'ask_size2_log']
target_col = 'rv_future'
```

```
# Load data
df = pd.read_parquet("DATA3888/Optiver-07/Data/FE30Stocks.parquet")
```

```
# Prepare features, target, and weights (inverse variance as heteroscedastic weights)
X = df[feature_cols].astype('float32')
y = df[target_col].astype('float32')
w = 1.0 / (y.rolling(2000, min_periods=1).var().fillna(y.var()))
```

```
# Train-test split
split_idx = int(len(df) * 0.8)
X_train, X_test = X.iloc[:split_idx],  X.iloc[split_idx:]
y_train, y_test = y.iloc[:split_idx],  y.iloc[split_idx:]
w_train, w_test = w.iloc[:split_idx],  w.iloc[split_idx:]
```

```
# Add intercept term
X_train_c = sm.add_constant(X_train, has_constant='add')
X_test_c  = sm.add_constant(X_test,  has_constant='add')

# Weighted Least Squares regression
model     = sm.WLS(y_train, X_train_c, weights=w_train)
results   = model.fit()
print(results.summary())
```

```
# Predict and evaluate
y_pred = results.predict(X_test_c)
r2     = r2_score(y_test, y_pred)
qlike  = qlike_loss(y_test.values, y_pred)

print(f"Out-of-sample R²   : {r2:0.4f}")
print(f"Out-of-sample QLIKE: {qlike:0.6f}")
```

8.2.16 Random Forest

```
df = pd.read_parquet("DATA3888/Optiver-07/Data/FE30Stocks.parquet")

feature_cols_mod = ['stock_id', 'mid_price', 'spread', 'imbalance',
       'book_pressure', 'LOB_entropy', 'log_return', 'bipower_var',
       'log_wap_return', 'imbalance_lag1', 'imbalance_lag2',
       'book_pressure_lag1', 'book_pressure_lag2', 'log_return_lag1',
       'log_return_lag2', 'rolling_vol_30', 'rolling_imbalance_mean_30',
       'sec_sin', 'sec_cos', 'bid_size1_log', 'ask_size1_log', 'bid_size2_log',
       'ask_size2_log']
target_col = "rv_future"

df['rv_future_log'] = np.log1p(df[target_col])
target_col_mod   = 'rv_future_log'
```

```
unique_sessions = np.sort(df['time_id'].unique())
split_idx       = int(len(unique_sessions) * 0.8)

train_val_sessions = unique_sessions[:split_idx]
test_sessions      = unique_sessions[split_idx:]

train_val_df = df[df['time_id'].isin(train_val_sessions)].copy()
test_df      = df[df['time_id'].isin(test_sessions)].copy()

val_cut        = int(len(train_val_sessions) * 0.9)
train_sessions = train_val_sessions[:val_cut]
val_sessions   = train_val_sessions[val_cut:]
```

```
train_df = train_val_df[train_val_df['time_id'].isin(train_sessions)]
val_df   = train_val_df[train_val_df['time_id'].isin(val_sessions)]

X_train = train_df[feature_cols_mod].values
y_train = train_df[target_col_mod].values.ravel()

X_val   = val_df[feature_cols_mod].values
y_val   = val_df[target_col_mod].values.ravel()

X_test  = test_df[feature_cols_mod].values
y_test  = test_df[target_col_mod].values.ravel()
```

```
rf = RandomForestRegressor(
    n_estimators=500,
    max_depth=None,
    max_features='sqrt',
    min_samples_leaf=3,
    bootstrap=True,
    n_jobs=-1,
    random_state=42,
    verbose=1
)
rf.fit(X_train, y_train)
```

```
val_pred  = rf.predict(X_val)
val_rmse  = root_mean_squared_error(y_val, val_pred)
print(f"Validation RMSE: {val_rmse:.6f}")
```

```
pred = rf.predict(X_test)
rmse = root_mean_squared_error(y_test, pred)
print(f"Out-of-sample RMSE = {rmse:.6f}")
```

```
y_true_raw = y_test
y_pred_raw = rf.predict(X_test)

rmse = root_mean_squared_error(y_true_raw, y_pred_raw)
r2 = r2_score(y_true_raw, y_pred_raw)

def qlike_safe(actual, forecast, eps=1e-8):
    a = np.clip(actual, eps, None)
    f = np.clip(forecast, eps, None)
    r = a / f
    return np.mean(r - np.log(r) - 1.0)

ql = qlike_safe(y_true_raw, y_pred_raw)

print(f"Out-of-sample RMSE: {rmse:.6f}")
print(f"R² score          : {r2:.6f}")
print(f"QLIKE             : {ql:.6f}")
```

8.2.17 LSTM

```
random.seed(3888)
np.random.seed(3888)
tf.random.set_seed(3888)
```

```
df = pd.read_parquet("DATA3888/Optiver-07/Data/FE30Stocks.parquet")

feature_cols = ['mid_price', 'spread',
       'imbalance', 'book_pressure', 'LOB_entropy', 'log_return',
       'bipower_var', 'log_wap_return', 'imbalance_lag1', 'imbalance_lag2',
       'book_pressure_lag1', 'book_pressure_lag2', 'log_return_lag1',
       'log_return_lag2', 'rolling_vol_30', 'rolling_imbalance_mean_30',
       'sec_sin', 'sec_cos', 'bid_size1_log', 'ask_size1_log', 'bid_size2_log',
       'ask_size2_log']
target_col = "rv_future"

df['rv_future_log'] = np.log1p(df['rv_future'])
```

```
feature_cols_mod = [c for c in feature_cols if c!='rv_future']
target_col_mod     = 'rv_future_log'
```

```
unique_sessions = df["time_id"].sort_values().unique()
split_idx       = int(len(unique_sessions) * 0.8)
train_sessions  = unique_sessions[:split_idx]
test_sessions   = unique_sessions[split_idx:]
```

```
train_df = df[df["time_id"].isin(train_sessions)].copy()
test_df  = df[df["time_id"].isin(test_sessions)].copy()
```

```
x_scaler = MinMaxScaler().fit(train_df[feature_cols_mod])
y_scaler = MinMaxScaler(feature_range=(0,1)).fit(train_df[[target_col_mod]])

train_df[feature_cols_mod] = x_scaler.transform(train_df[feature_cols_mod])
test_df[feature_cols_mod]  = x_scaler.transform(test_df[feature_cols_mod])
train_df[target_col_mod]   = y_scaler.transform(train_df[[target_col_mod]])
test_df[target_col_mod]    = y_scaler.transform(test_df[[target_col_mod]])
```

```
def build_sequences(df_part: pd.DataFrame, feature_cols, target_col, seq_len):
    X, y = [], []
    for _, session in df_part.groupby("time_id"):
        data   = session[feature_cols].values
        target = session[target_col].values
        for i in range(len(session) - seq_len):
            X.append(data[i : i + seq_len])
            y.append(target[i + seq_len])
    return np.asarray(X), np.asarray(y)

X_train, y_train = build_sequences(train_df, feature_cols, target_col, SEQ_LEN)
X_test,  y_test  = build_sequences(test_df,  feature_cols, target_col, SEQ_LEN)
```

```
val_split_idx = int(len(train_sessions) * 0.9)
val_sessions  = train_sessions[val_split_idx:]
train_sessions= train_sessions[:val_split_idx]

val_df = train_df[train_df["time_id"].isin(val_sessions)]
train_df = train_df[train_df["time_id"].isin(train_sessions)]

X_train, y_train = build_sequences(train_df, feature_cols_mod, 'rv_future_log', SEQ_LEN)
```

```
X_val,   y_val   = build_sequences(val_df,   feature_cols_mod, 'rv_future_log', SEQ_LEN)
X_test,  y_test  = build_sequences(test_df,  feature_cols_mod, 'rv_future_log', SEQ_LEN)
```

```
def build_lstm_model(seq_len, num_features):
    model = Sequential()
    model.add(LSTM(64, return_sequences=True, input_shape=(seq_len, num_features)))
    model.add(Dropout(0.2))
    model.add(LSTM(32))
    model.add(Dropout(0.2))
    model.add(Dense(16, activation='relu'))
    model.add(Dense(1))
    return model

NUM_FEATURES = X_train.shape[2]
model = build_lstm_model(SEQ_LEN, NUM_FEATURES)

callbacks = [
    EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)
]

model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
    loss='mse',
    metrics=['mae', 'mse']
)
model.summary()
```

```
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=50,
    batch_size=128,
    callbacks=callbacks,
    verbose=1
)
```

```
pred_scaled = model.predict(X_test, verbose=1).flatten()
actual_scaled = y_test.flatten()
predictions = y_scaler.inverse_transform(pred_scaled.reshape(-1, 1)).flatten()
actuals     = y_scaler.inverse_transform(actual_scaled.reshape(-1, 1)).flatten()
```

```
ql = qlike_loss(actuals, predictions)
mse   = np.mean((predictions - actuals) ** 2)
rmse  = np.sqrt(mse)
r2 = r2_score(actuals, predictions)
print(f"Test RMSE (volatility): {rmse:.9f}")
print(f"R² score (  prediction): {r2:.6f}")
print("QLIKE:", ql)
```

8.2.18 Transformer

```
df = pd.read_parquet("DATA3888/Optiver-07/Data/FE30Stocks.parquet")

feature_cols = ['mid_price', 'spread',
       'imbalance', 'book_pressure', 'LOB_entropy', 'log_return',
       'bipower_var', 'log_wap_return', 'imbalance_lag1', 'imbalance_lag2',
       'book_pressure_lag1', 'book_pressure_lag2', 'log_return_lag1',
       'log_return_lag2', 'rolling_vol_30', 'rolling_imbalance_mean_30',
       'sec_sin', 'sec_cos', 'bid_size1_log', 'ask_size1_log', 'bid_size2_log',
       'ask_size2_log']
target_col = "rv_future"

df['rv_future_log'] = np.log1p(df['rv_future'])
feature_cols_mod = [c for c in feature_cols if c!='rv_future']
target_col_mod     = 'rv_future_log'
```

```
unique_sessions = df["time_id"].sort_values().unique()
split_idx       = int(len(unique_sessions) * 0.8)
train_sessions  = unique_sessions[:split_idx]
test_sessions   = unique_sessions[split_idx:]
```

```
train_df = df[df["time_id"].isin(train_sessions)].copy()
test_df  = df[df["time_id"].isin(test_sessions)].copy()
```

```
x_scaler = MinMaxScaler().fit(train_df[feature_cols_mod])
y_scaler = MinMaxScaler(feature_range=(0,1)).fit(train_df[[target_col_mod]])

train_df[feature_cols_mod] = x_scaler.transform(train_df[feature_cols_mod])
test_df[feature_cols_mod]  = x_scaler.transform(test_df[feature_cols_mod])
train_df[target_col_mod]   = y_scaler.transform(train_df[[target_col_mod]])
test_df[target_col_mod]    = y_scaler.transform(test_df[[target_col_mod]])
```

```
X_train, y_train = build_sequences(train_df, feature_cols, target_col, SEQ_LEN)
X_test,  y_test  = build_sequences(test_df,  feature_cols, target_col, SEQ_LEN)
```

```
def build_transformer_model(seq_len, num_features, d_model=64, num_heads=4, num_layers=2):
    inputs = layers.Input(shape=(seq_len, num_features))
    x = layers.Dense(d_model)(inputs)
    for _ in range(num_layers):
        attn_output = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=d_model)(x, x)
        x = layers.Add()([x, attn_output])
        x = layers.LayerNormalization(epsilon=1e-6)(x)
        ffn_out = layers.Dense(d_model * 4, activation="relu")(x)
        ffn_out = layers.Dense(d_model)(ffn_out)
        x       = layers.Add()([x, ffn_out])
        x       = layers.LayerNormalization(epsilon=1e-6)(x)
    x = layers.GlobalAveragePooling1D()(x)
    output = layers.Dense(1)(x)
    return models.Model(inputs, output)

model = build_transformer_model(SEQ_LEN, len(feature_cols))
model.compile(optimizer=tf.keras.optimizers.Adam(1e-3), loss="mse")
model.summary()
```

```
val_split_idx = int(len(train_sessions) * 0.9)
val_sessions  = train_sessions[val_split_idx:]
train_sessions= train_sessions[:val_split_idx]

val_df = train_df[train_df["time_id"].isin(val_sessions)]
train_df = train_df[train_df["time_id"].isin(train_sessions)]

X_train, y_train = build_sequences(train_df, feature_cols_mod, 'rv_future_log', SEQ_LEN)
X_val,   y_val   = build_sequences(val_df,   feature_cols_mod, 'rv_future_log', SEQ_LEN)
X_test,  y_test  = build_sequences(test_df,  feature_cols_mod, 'rv_future_log', SEQ_LEN)
```

```
num_feats = X_train.shape[2]
model = build_transformer_model(SEQ_LEN, num_feats)

model.compile(
    optimizer=tf.keras.optimizers.Adam(1e-3),
    loss="mse"
)
```

```
history = model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=50,
    batch_size=32,
    callbacks=[callbacks.EarlyStopping(
        monitor="val_loss", patience=15, restore_best_weights=True
    )],
    verbose=1,
)
```

```
pred_scaled = model.predict(X_test, verbose=1).flatten()
actual_scaled = y_test.flatten()
predictions = y_scaler.inverse_transform(pred_scaled.reshape(-1, 1)).flatten()
actuals     = y_scaler.inverse_transform(actual_scaled.reshape(-1, 1)).flatten()
mse   = np.mean((predictions - actuals) ** 2)
rmse  = np.sqrt(mse)
print(f"Test RMSE (volatility): {rmse:.9f}")
```

```
r2 = r2_score(actuals, predictions)
print(f"R² score (  prediction) = {r2:.6f}")
```

```
def qlike_safe(actual, forecast, eps=1e-12):
    a = np.clip(actual,   eps, None)
    f = np.clip(forecast, eps, None)
    r = a / f
    return np.mean(r - np.log(r) - 1.0)

ql = qlike_safe(actuals, predictions)
print("QLIKE:", ql)
```

8.3 Plots

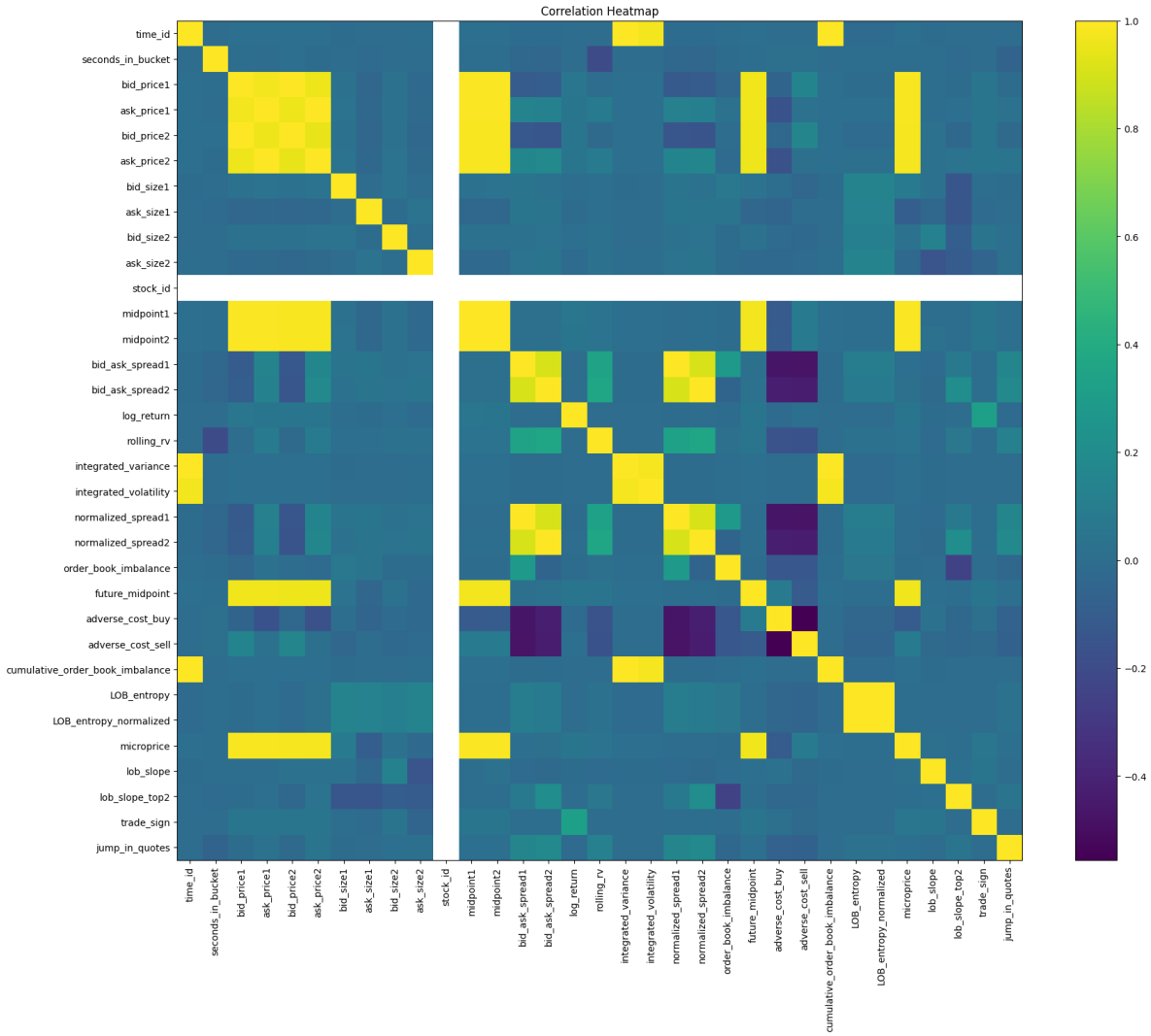

Figure 2: Correlation Plot

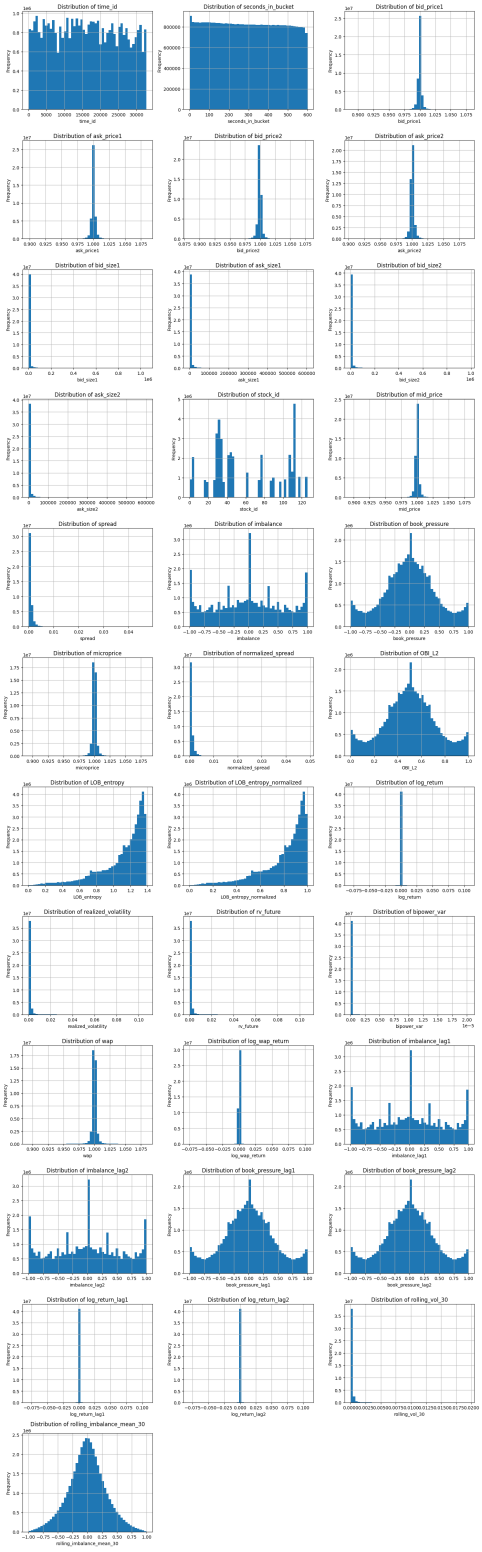

Figure 3: Data Distribution