THE AKRATEO TRADING ENGINE

Akhram Aramayo Salem, Mateo Saettone Ascenzo

University of South Florida

Department of Science & Engineering

Tampa, Florida

April 4th, 2025

Table of Contents

Overview

Introduction

An ensemble of "Parent-Child" algorithms, that combines deep learning and machine learning, developed generate resources to handle real-time decisions when day trading American out-of-the-money options.

This ensemble consists of 19 predictive models that fall under four different architectures, each designed to forecast a different aspect of next-day price and volatility behavior for a desired stock, in this case, The Coca Cola Company stock (KO).

The objective is to generate trading signals and manage positions intraday using data-driven strategies.

Focus

The models are designed to predict various price and volatility metrics of the underlying stock rather than directly forecasting the option price. By estimating future stock movements and probability, the framework enables informed intraday option trading decisions.

Additional features affecting option pricing during the day are incorporated into the training process of the algorithms as input variables and accounted for operational action in the execution strategy.

This approach isolates underlying stock price and volatility forecasting from option-specific complexities, ensuring that market signals are derived mostly from underlying asset behavior while maintaining adaptability for real-time decision-making.

Execution Framework

Each trading day at exactly 9:27 AM Eastern Time, the system initiates a live snapshot of key market features from a data provider and transmits the information to a cloud-based compute node optimized for real-time inference.

This node, built for high-throughput processing, rapidly filters the incoming data stream to extract the most predictive signals using a dynamic feature selection framework.

By the time the data arrives, the infrastructure is primed for immediate analysis, minimizing delay between data retrieval and model activation.

From there, the system engages a tightly structured but parallelized forecasting pipeline. A series of deep learning and ensemble models are deployed in succession to generate high-confidence predictions for both stock returns and volatility.

These outputs are unified into a structured payload, which is then adapted to match the format requirements of the trading engine.

While the underlying models have been rigorously detailed in earlier sections, the real challenge here lies in the orchestration—ensuring that all forecasting layers, data transformations, and formatting steps occur within a stringent three-minute service window.

Finally, the signal is routed to a live trading platform that continuously monitors for new predictions. As soon as the signal arrives, the system validates it, prepares the necessary order types, and submits them through a direct brokerage API—in this case, via a native integration with Charles Schwab.

Feedback from the brokerage is looped back to the compute node in real time, completing the full execution cycle.

The system is engineered to maintain synchronized clocks, fault tolerance, and end-to-end timing precision, ensuring that market decisions are not only data-driven but also delivered within the temporal thresholds that define institutional-grade execution.

General Feature Selection

A broad selection of technical, macroeconomic, and statistical indicators underpins the modeling process.

This comprehensive selection baseline accounts for the key drivers of stock price and volatility movements. The selection of variables will be addressed more specifically in the *Data Analysis & Feature Selection* section, where a detailed examination of the most relevant features for predictive accuracy will be conducted.

Models Approach

Nineteen models form the core of the prediction engine and their targets defined as:

Hybrid Bidirectional Stacked LSTMs

Stock Price

1. $xLSTM_{(Close)} = \left(\frac{S_{(Close)}}{S_{(Open)}}\right) - 1$
2. $xLSTM_{(High)} = \left(\frac{S_{(High)}}{S_{(Open)}}\right) - 1$
3. $xLSTM_{(Low)} = \left(\frac{S_{(Low)}}{S_{(Open)}}\right) - 1$

Stock Volatility

1. $xLSTM_{(Close)} = \log\left(\frac{\sigma_{(Close)}}{\sigma_{(Open)}}\right)$
2. $xLSTM_{(Max)} = \log\left(\frac{\sigma_{(Max)}}{\sigma_{(Open)}}\right)$
3. $xLSTM_{(Min)} = \log\left(\frac{\sigma_{(Min)}}{\sigma_{(Open)}}\right)$

Vanilla Temporal Fusion Transformers

Stock Price

1. $TFT_{(Close)} = \left(\frac{S_{(Close)}}{S_{(Open)}}\right) - 1$
2. $TFT_{(High)} = \left(\frac{S_{(High)}}{S_{(Open)}}\right) - 1$

3. $TFT_{(Low)} = \left(\frac{S_{(Low)}}{S_{(Open)}}\right) - 1$

Stock Volatility

1. $TFT_{(Close)} = \log\left(\frac{\sigma_{(Close)}}{\sigma_{(Open)}}\right)$
2. $TFT_{(Max)} = \log\left(\frac{\sigma_{(Max)}}{\sigma_{(Open)}}\right)$
3. $TFT_{(Min)} = \log\left(\frac{\sigma_{(Min)}}{\sigma_{(Open)}}\right)$

Meta Learner with XGBoost Structure (Regression)

Stock Price

1. $ML_{(Close)} = \left(\frac{S_{(Close)}}{S_{(Open)}}\right) - 1$
2. $ML_{(High)} = \left(\frac{S_{(High)}}{S_{(Open)}}\right) - 1$
3. $ML_{(Low)} = \left(\frac{S_{(Low)}}{S_{(Open)}}\right) - 1$

Stock Volatility

1. $ML_{(Close)} = \log\left(\frac{\sigma_{(Close)}}{\sigma_{(Open)}}\right)$
2. $ML_{(Max)} = \log\left(\frac{\sigma_{(Max)}}{\sigma_{(Open)}}\right)$
3. $ML_{(Min)} = \log\left(\frac{\sigma_{(Min)}}{\sigma_{(Open)}}\right)$

XGBoost (Classification)

Stock Price

1. $\hat{Y}_{(Column)} = if\left(\frac{S_{(Close)}}{S_{(Open)}}\right) > 0 \Longrightarrow 1\ if\ not \Longrightarrow\ 0$

$$XGB_{(Close)} = P\left(\frac{S_{(Close)}}{S_{(Open)}}\right) > 0$$

By combining these models, the framework aims to capture the direction and magnitude of the stock's next price and volatility.

Introduction to Expanding Window

The Expanding Window methodology constitutes the cornerstone of the proposed modeling framework.

It is designed to be employed comprehensively across all constructed singular algorithms, including their training processes, real-time implementations, and analytical testing within the *Data Analytics & Feature Selection* process.

This approach involves daily retraining of models with every new data point available, adhering strictly to the predetermined data periodicity—in this instance, daily.

Conventional algorithmic structures with static model frameworks, are compelled to generalize, primarily to accommodate that these models are developed and tested on datasets limited to a fixed historical window.

Consequently, they inherently neglect the most recent data dynamics emerging over time. The Expanding Window method directly addresses this limitation by recalibrating models daily with every newly obtained data point.

This daily retraining ensures continuous realignment of the models with the latest data-driven market dynamics.

As a result of this continuous adjustment, Expanding Window removes the necessity for periodic recalibration, which is an obligatory practice in traditional modeling techniques.

Conventional models must undergo periodic recalibrations to revert to optimal efficiency levels, accommodating recent data which was not present during their initial training phase and thus realigning the original relationships and patterns.

In contrast, Expanding Window methodology inherently achieves daily recalibration, thereby consistently capturing maximal predictive efficiency.

Further theoretical details and a comprehensive discussion of this methodological approach can be found within a dedicated subsection under the *Novel Theorems* section.

Overview Summary

This framework combines deep learning and traditional machine learning through an ensemble of nineteen predictive models to generate real-time trading signals for American out-of-the-money options.

The models forecast different price and volatility movements of the underlying stock, guiding intraday options trading decisions.

The methodology incorporates an Expanding Window approach to adapt to evolving market conditions, with detailed model integration and execution strategies.

By leveraging Bloomberg's data for both backtesting and live trading, and utilizing QuantConnect's capabilities, the system ensures consistent execution.

Further details of each aspect will be discussed in the following sections.

Novel Theorems

This section introduces novel theorems developed specifically for this study, providing foundational concepts crucial to understanding the methodologies detailed throughout the paper.

Three primary theorems are described: Expanding Window, Sae-Mayo, and Arattone theorem. Each addresses a distinct and critical component of predictive modeling.

The Expanding Window theorem establishes a principle of dynamic adaptation, enabling models to continuously recalibrate using the most recent market data. This approach ensures models remain highly responsive to changing market conditions, thereby maintaining maximal predictive accuracy and robustness over time.

The Sae-Mayo theorem quantitatively defines the optimal number of registers necessary for effective model training, ensuring predictive models reach maximum precision without minimal, ever-lower returns that lead to computational overhead or redundancy.

The Arattone theorem identifies the optimal dimensionality for models handling multidimensional datasets, suggesting that more data does not always lead to better performance.

Collectively, these novel theorems form a strong theoretical basis that enhances precision, adaptability, and efficiency in the ensemble forecasting framework.

In the following subsections we will detail each of the previously stated theorems.

Expanding Window

Expanding Window methodology entails the daily retraining of predictive models utilizing the entire historical dataset available up to the most recent observation. Each day, new data points are incorporated, prompting a complete recalibration of the model to reflect the latest market dynamics.

In contrast, traditional modeling approaches typically partition the dataset into fixed training and testing subsets.

These models, once trained, generate predictions for subsequent future periods $(t + 1, t + 2 \dots t + N)$ without recalibration, resulting in diminishing predictive accuracy as market dynamics evolve beyond the initial training period.

Such static models progressively lose predictive performance as they fail to accommodate changing market patterns and relationships.

Expanding Window directly addresses this limitation by incorporating daily retraining. Specifically, algorithms are configured to utilize all available historical observations for learning, and there is not a testing set.

The model is initially trained, and subsequently, the last data point not available for data alignment (shift(-1) is reintegrated into a contextualization phase, providing the prediction for the next period $(t + 1)$.

The following day, a new data point is added, and this retraining cycle repeats indefinitely. This systematic daily retraining inherently negates the necessity for periodic manual recalibrations and sustains predictive performance continuously.

The Overfitting Constraint & Management

A critical aspect of Expanding Window implementation is the inevitable overfitting resulting from training on extensive historical and not including a test set.

While overfitting conventionally signifies a negative phenomenon characterized by excessive adherence to specific dataset patterns, it becomes strategically beneficial within the Expanding Window framework.

Generalizable models, which seek broader applicability, inherently sacrifice precision in capturing specific temporal dynamics. Conversely, the deliberate overfitting inherent in Expanding Window allows models to precisely capture contemporary market nuances.

Without daily retraining, such overfitted models experience rapid performance deterioration described mathematically by a negative logarithmic function; predictive accuracy diminishes sharply with time.

However, by continuously truncating the temporal dimension through daily retraining, Expanding Window converts this negative logarithmic decay into a slightly positive logarithmic trajectory, ensuring sustained incremental improvements in predictive accuracy.

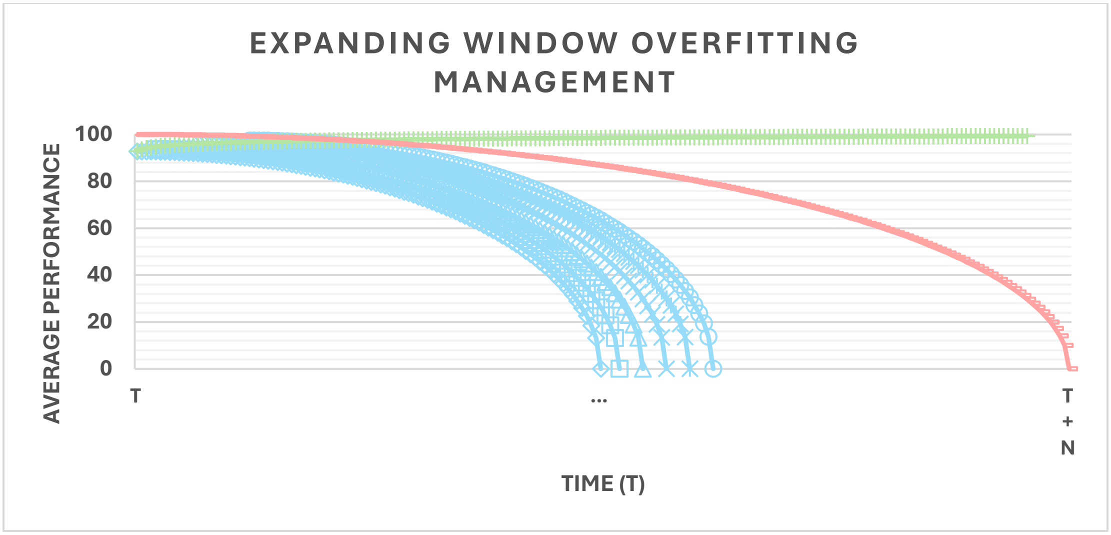

Figure 1

Figure 1 illustrates this relationship, where negative logarithmic functions depict the rapid decline of predictive accuracy in overfitted models without Expanding Window, contrasted by the slightly positive logarithmic trajectory maintained by the Expanding Window methodology. The red logarithmic function represents the relationship between performance and time of a non-overfitted conventional model.

Consequently, incorporating regularization techniques (L1/L2 penalization, early stopping, among others) would prove detrimental, prematurely constraining the model's adaptability and limiting its ability to absorb emerging market patterns rapidly.

Within Expanding Window contexts, overfitting is thus strategically advantageous, and regularization methods that inhibit rapid adjustments undermine performance.

Backtesting and Data Distribution

The unique characteristics of Expanding Window present specific challenges to conventional backtesting practices, traditionally reliant upon extensive fixed-size test datasets.

Since Expanding Window does not utilize a test set at each recalibration, traditional backtesting frameworks become ineffective.

To address this, Expanding Window backtesting replicates the forward-looking methodology retrospectively. An initial subset of historical observations (Data Available for First Training - DAFFT) is selected, from which the first out-of-sample prediction is generated.

Subsequently, each additional historical observation sequentially expands the training dataset, replicating the daily recalibration process and providing continuous out-of-sample predictions. This backward replication emulates real-time predictive conditions.

The backtesting data distribution and Expanding Window replication process are visualized in *Figure 2*, delineating the roles of Data Available for First Training (DAFFT), Outputs Considering Expanding Window (OCEX), and Backtesting datasets (BT).

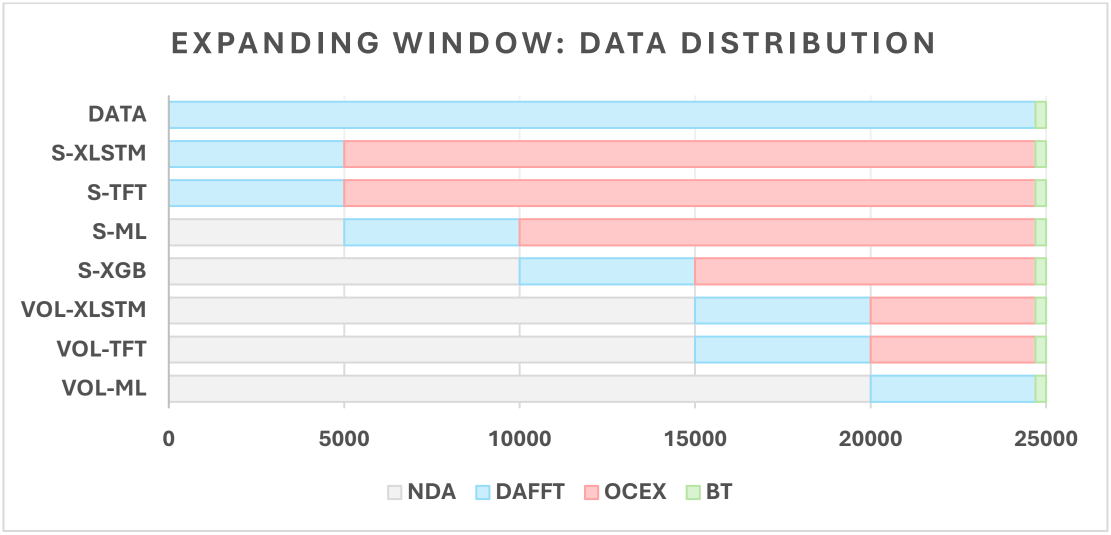

Figure 2

Nevertheless, an inherent constraint in Expanding Window backtesting is reduced predictive accuracy in initial predictions due to limited initial data availability.

Empirical evidence suggests a logarithmic relationship between predictive performance and training set size, characterized by diminishing marginal improvements beyond a certain threshold.

The Sae-Mayo theorem empirically identifies approximately 5,000 observations as an optimal threshold, beyond which additional data provides minimal incremental benefit to model accuracy.

Sae-Mayo Theorem for Observational Efficiency

The *Sae-Mayo Observational Efficiency* is an empirically derived principle that quantifies the logarithmic relationship between a model's predictive performance and the cumulative size of its training dataset.

It asserts that while performance generally increases with data volume, marginal improvements diminish past a certain data volume threshold.

The relationship between the number of observations and predictive performance follows a classic logarithmic growth curve. Initially, each additional observation provides a substantial boost in accuracy, but the rate of improvement decreases progressively.

The curve approaches an asymptote where performance gains become negligible. Through extensive empirical trials across multiple financial time series and forecasting scenarios, the Sae-Mayo theorem identifies 5,000 observations as the critical inflection point.

This value marks the transition between high-magnitude performance improvements and the plateau of diminishing returns.

The expressed relationship is visualized in *Figure 3*.

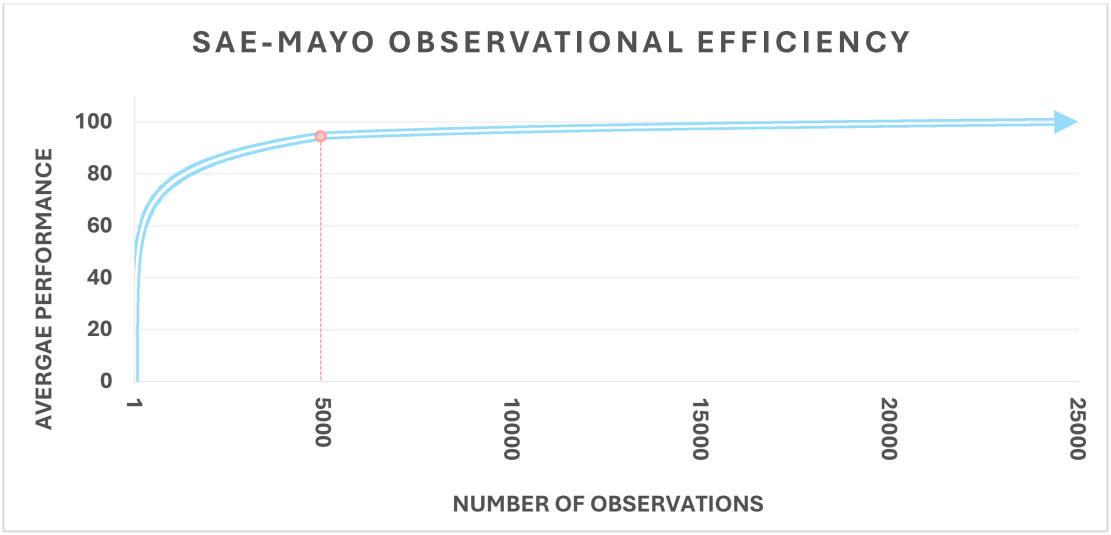

Figure 3

In early stages of Expanding Window backtesting, the training set contains 5,000 data points or more, as a result initial predictions may exhibit a minimal suboptimal accuracy due to insufficient sample size.

However, as the expanding dataset increases, the marginal gains in predictive power decrease logarithmically, stabilizing model performance and mitigating early-stage volatility.

This empirical ceiling ensures that backtesting outcomes can be interpreted with higher statistical confidence, as the model operates near peak efficiency.

Arattone Theorem for High Dimensionality

The Arattone Theorem for High Dimensionality formalizes the relationship between model performance and the dimensionality of the input feature space. Here, dimensionality refers to the number of predictor variables utilized by the model. This theorem characterizes the inherent trade-off between feature augmentation and predictive accuracy through a non-monotonic, concave quadratic function.

The core premise assumes a prior ranking of input variables based on their marginal predictive power—i.e., the most informative variable is ranked first, the second most informative second, and so on.

This ranking is derived through techniques discussed in the *Data Analysis & Feature Selection* section (e.g., mutual information, univariate correlation with the target, or recursive feature elimination).

The theorem posits that when features are sequentially added to a predictive model according to this ranked order, the performance—initially improving—eventually plateaus and then deteriorates.

This phenomenon reflects the classical bias-variance tradeoff and the risk of overparameterization, where redundant or noisy features begin to obscure signal.

Formally, the relationship between performance P and dimensionality N_V is modeled as:

$$P\left(N_{(V)}\right) = aN_{(V)}^2 + bN_{(V)} + c$$

- Where $P(N_V)$ denotes the normalized performance of the model when using the top N_V features, and $a > 0$ ensures concavity.

The optimal number of predictors, denoted N_V^* corresponds to the vertex of the parabola and is computed via:

$$\frac{dP}{dN_{(V)}} = 2aN_{(V)} + b = 0 \Rightarrow \; N_{(V)}^* = -\frac{b}{2a}$$

Beyond N_V^*, the marginal addition of variables introduces noise, multicollinearity, and spurious correlations that degrade generalization. This deterioration is consistent with phenomena such as the curse of dimensionality and information redundancy in high-dimensional statistical learning.

To quantify model performance P, the theorem leverages the log-likelihood (LL) under the assumption of Gaussian-distributed prediction errors. Let $\epsilon_{(i)} = Y_{(i)} - \hat{Y}_{(i)} \sim N(0, \sigma^2)$, then:

$$LL = -\frac{N}{2}\log(2\pi\sigma^2) - \frac{1}{2a^2}\sum_{i=1}^{N}\epsilon_{(i)}^2$$

To align with the normalized scale proposed by the theorem, performance is redefined as a scaled relative log-likelihood:

$$P\left(N_{(V)}\right) = \frac{LL\left(N_{(V)}\right) - LL(1)}{LL\left(N_{(V)}^*\right) - LL(1)}$$

This normalization anchors the performance scale such that:

- $P(1) = 0$, representing the baseline performance using only the single most predictive feature.
- $P(N^*_{(V)}) = 1$, indicating maximal model performance at the optimal dimensionality,
- $P(N_{(V)}) \longrightarrow \infty < 0$, capturing performance degradation beyond the optimal threshold.

For visual interpretability, as demonstrated in *Figure 4*, all values of $P(N_{(V)})$ are scaled by 100:

- $P(1) = 0$, signifies the initial performance baseline.
- $P(N^*_{(V)}) = 100$, marks the performance peak.
- $P(N_{(V)}) < 0$, beyond this point denotes regression below baseline due to superfluous or deleterious variables.

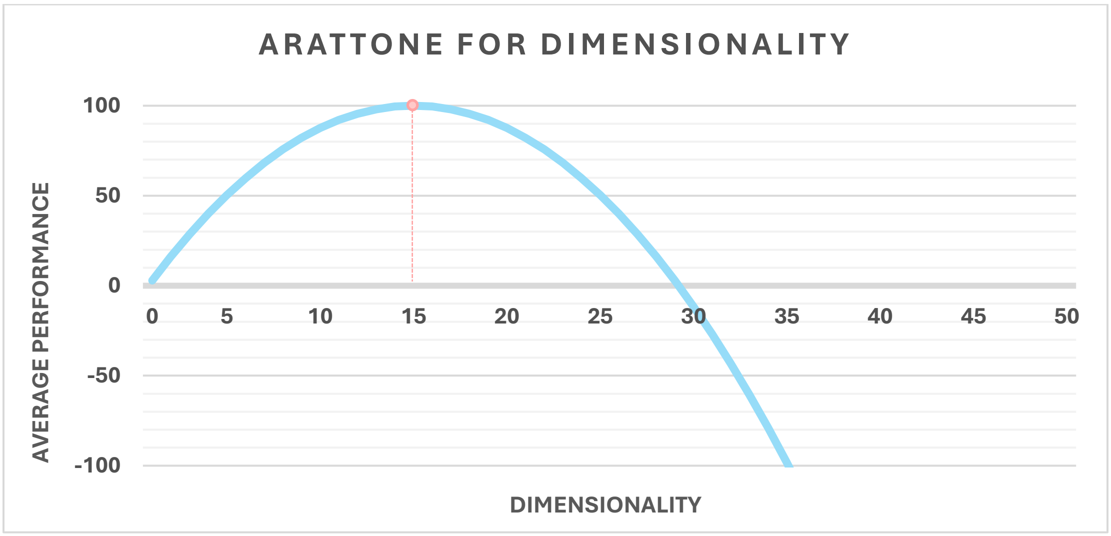

Figure 4

Figure 4 illustrates this concave relationship based on empirical data derived from extensive model evaluations, showing that the optimal number of features is approximately 15 in the observed average setting.

The Arattone Theorem therefore functions as both a diagnostic tool and a prescriptive framework for determining the ideal feature subset size, particularly in scenarios involving large-scale feature spaces.

Note: Precisions metrics from Expanding Window and Sae-Mayo Theorem for Observational Efficiency are the same as the recently detailed.

Data Analysis & Feature Selection

The objective of this section is to provide the models with optimal input data in terms of relevance, temporal adaptability, and predictive power.

An initial structured approach is essential to map and filter between 500 and 600 key variables towards their respective goals, ensuring that each feature contributes significant predictive value while avoiding spurious correlations and misleading causalities.

Given the scale of the dataset, various challenges related to data integrity and consistency are expected. Addressing these issues is crucial to mitigate biases that could undermine the model's robustness.

This section outlines the anticipated data challenges, the need for resolution, and the preprocessing and selection methodologies that will be implemented to ensure high-quality input data before model training begins.

Data Quality

Data quality refers to the accuracy of time series data, ensuring proper chronological order. It primarily focuses on the absence of missing values, infinite values, or an excessive number of outliers.

Since chronological integrity should already have been addressed at this stage, the focus now shifts to handling the presence of outliers and missing or non-interpretable data.

Missing Data & Non-Interpretable Data

Potential Impact

xLSTM

Missing or non-interpretable values (NaN, inf, or -inf) disrupt the training process of LSTMs by causing numerical instabilities during both forward and backward propagation.

Since LSTMs rely on temporally coherent sequences, interruptions in the continuity of the data hinder proper weight updates, preventing convergence and diminishing the model's ability to learn meaningful temporal dependencies.

Temporal Fusion Transformers

Missing or non-interpretable values (NaN, inf, or -inf) compromise the attention mechanisms and gating layers within Temporal Fusion Transformers.

These anomalies obstruct the model’s capacity to assign dynamic importance across time steps and input variables, leading to unstable hidden representations and degraded temporal learning.

As a result, TFTs exhibit diminished interpretability and predictive performance, particularly in scenarios where input completeness is critical to temporal context awareness.

XGBoost

XGBoost is sensitive to NaN and infinite values during split evaluation, as these disrupt the histogram-based gradient boosting algorithm.

When non-interpretable values are present, tree construction becomes biased or incomplete, resulting in suboptimal partitions and unstable gain computation.

This degrades the model’s ability to distinguish signal from noise and can severely impair classification confidence.

Addressing the Problem: INFNA Treatment

The INFNA treatment aims to identify and correct non-interpretative values, such as infinity $(+\infty, -\infty)$ and null (NA) values in data records before training the algorithms.

The process is divided into three stages: Identification, Treatment, and Verification. Below is a detailed explanation of the methodology.

Identification

For each variable $j \in V$ in a dataset D with T records, problematic values are identified and classified into three categories.

Null Values

$$p_{(j)}^{NA} = \frac{\sum_{t=1}^{T} 1\left(NA\left(y_{(j,t)}\right)\right)}{T}$$

- Where $y_{(j,t)}$ is the value of a variable j at record t, and $1\left(NA\left(y_{(j,t)}\right)\right)$ is and indicator function that returns 1 if $y_{(j,t)}$ is NA and 0 otherwise.

Positive Infinity Values

$$p_{(j)}^{+\infty} = \frac{\sum_{t=1}^{T} 1\left(y_{(j,t)} = +\infty\right)}{T}$$

Negative Infinity Values

$$p_{(j)}^{-\infty} = \frac{\sum_{t=1}^{T} 1\left(y_{(j,t)} = -\infty\right)}{T}$$

In cases where infinity values are not directly identified, there is a verification of the presence of values near $\pm\infty$.

A numerical tolerance range ϵ is defined, and values falling within the range $\left[\frac{1}{\epsilon}, \epsilon\right]$ are checked and treated as $\pm\infty$.

- The machine epsilon ϵ, is the smallest value that, when added to 1, changes the result. It's a measure of the precision of floating-point arithmetic in a given system.

$$\epsilon = min\{\epsilon \mid 1 + \epsilon > 1\}$$

- Once ϵ is defined, check if values fall within the tolerance range. If values are found to be near $\pm\infty$, the appropriate replacement is applied.

$$1_{(near\ \inf)}\left(y_{(j,t)}\right) = 1\left(\left|y_{(j,t)}\right| \geq \frac{1}{\epsilon}\right)$$

$$y_{(j,t)} = \left\{\begin{matrix} +\infty & if & y_{(j,t)} \geq \frac{1}{\epsilon} \\ -\infty & if & y_{(j,t)} \leq -\frac{1}{\epsilon} \end{matrix}\right\}$$

Treatment: Before Procedure

An evaluation is performed based on the total proportion of problematic values $p_{(j)}^{total}$ for each variable.

$$p_{(j)}^{total} = \frac{p_{(j)}^{NA} + p_{(j)}^{+\infty} + p_{(j)}^{-\infty}}{T}$$

If the percentage of problematic values is greater than or equal to an adjustable threshold ζ, the variable is removed.

$$if \quad p_{(j)}^{total} \geq \zeta \quad \Longrightarrow \quad j \notin V \in D$$

Threshold ζ depends on the statistical characteristics of the dataset.

- An initial range for ζ is defined after the total proportion of problematic values per variable $p_{(j)}^{total}$ is defined.

$$0.5\% \leq \zeta \leq 1\%$$

- For each variable j, the value $p_{(j)}^{total}$ is evaluated, and the threshold ζ is adjusted from the 90th percentile of the $p_{(j)}^{total}$ distribution.

$$\zeta = percentile_{90\%}\left(p_{(j)}^{total}\right)$$

$$if \quad p_{(j)}^{total} \geq \zeta \quad \Longrightarrow \quad j \notin V \in D$$

If the variable is not eliminated, the treatment for infinity and null values is performed.

Treatment: Infinity Values

Positive infinity values $+\infty$ are replaced with the smallest representable number greater than ϵ to avoid distortion in the models.

$$\epsilon_{(next)} = \epsilon(1 + \epsilon)$$

$$y_{(j,t)} = \begin{Bmatrix} \epsilon_{(next)} \quad if \quad y_{(j,t)} = +\infty \\ y_{(j,t)} \quad if \quad y_{(j,t)} \neq +\infty \end{Bmatrix}$$

Negative infinity values $-\infty$ are replaced with $-1(\epsilon_{next})$.

$$y_{(j,t)} = \begin{Bmatrix} -1\left(\epsilon_{(next)}\right) \quad if \quad y_{(j,t)} = -\infty \\ y_{(j,t)} \quad if \quad y_{(j,t)} \neq -\infty \end{Bmatrix}$$

Treatment: Null Values

For null values $NA\left(y_{(j,t)}\right)$, an analysis of cause is performed before imputation.

- Mathematical errors are generated by feature transformation. They are replaced with $\pm\epsilon_{(next)}$, depending on the sign of the previous value of $y_{(j,t)}$.

$$y_{(j,t)} = \begin{Bmatrix} +\epsilon_{(next)} & if & y_{(j,t)} = math\ error & and & y_{(j,t-1)} > 0 \\ -\epsilon_{(next)} & if & y_{(j,t)} = math\ error & and & y_{(j,t-1)} < 0 \\ y_{(j,t)} & if & y_{(j,t)} \neq math\ error & & \end{Bmatrix}$$

- Data source errors occur if the null value is due to a source error. Data point is replaced with the value of the verification source.

$$y_{(j,t)} = y_{(j,t)}(from\ verification\ source) \quad if \quad y_{(j,t)} = data\ error$$

- General Data Errors occur if the null value cannot be attributed to a specific error. Data point is replaced with the median of the last 5 periods.

$$y_{(j,t)} = med(y_{(j,t-5)}, \dots, y_{(j,t-1)}) \quad if \quad y_{(j,t)} = general\ data\ error$$

Verification: Context

After imputations are performed, to ensure that the imputed values do not significantly alter the statistical properties or temporal trends of the variables, a validation process is conducted.

Verification: Distribution Analysis

For each variable j, the mean and standard deviation are calculated before and after imputations.

$$\mu_{(before)} = \frac{1}{T}\sum_{t=1}^{T} y_{(j,t)}^{before} \quad \wedge \quad \mu_{(after)} = \frac{1}{T}\sum_{t=1}^{T} y_{(j,t)}^{after}$$

$$\sigma_{(before)} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}\left(y_{(j,t)}^{before} - \mu_{(before)}\right)^2} \quad \wedge \quad \sigma_{(after)} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}\left(y_{(j,t)}^{after} - \mu_{(after)}\right)^2}$$

The calculated mean and standard deviation values are compared before and after imputations using paired t-test for means and F-test for standard deviations.

Paired t-test for means:

$$t = \frac{\mu_{(before)} - \mu_{(after)}}{\sqrt{\frac{s_{(before)}^2}{T_{(before)}} + \frac{s_{(after)}^2}{T_{(after)}}}}$$

- Where $s_{(before)}^2$ and $s_{(after)}^2$ are the sample variances before and after imputation.

F-test for standard deviations:

$$F = \frac{s_{(before)}^2}{s_{(after)}^2}$$

- Where $s_{(before)}^2$ and $s_{(after)}^2$ represent the sample variances before and after imputations.

If the $p_{(value)}$ from the paired t-test is less than 0.05, the means are significantly different. If the $p_{(value)}$ from the F-test is less than 0.05, the standard deviations are significantly different.

$$if \quad t\left(p_{(value)}\right) < 0.05 \quad \Longrightarrow \quad means\ are\ significantly\ different$$

$$if \quad F\left(p_{(value)}\right) < 0.05 \quad \Longrightarrow \quad standard\ deviations\ are\ significantly\ different$$

Verification: Trend Analysis

For each variable j a linear regression model to the time series data before and after imputations is estimated.

$$\hat{\beta}_{(before)} = \arg\min_{\beta} \sum_{t=1}^{T} \left(y_{(j,t)}^{before} - (\alpha + \beta t) \right)^2$$

$$\hat{\beta}_{(after)} = \arg\min_{\beta} \sum_{t=1}^{T} \left(y_{(j,t)}^{after} - (\alpha + \beta t) \right)^2$$

The standard error of the estimated slopes $\hat{\beta}_{(before)}$ and $\hat{\beta}_{(after)}$ is calculated for both the before and after imputation regression models.

$$SE_{(\beta)} = \frac{\sqrt{\sum_{t=1}^{T} \frac{\left(y_{(j,t)} - \hat{y}_{(j,t)}\right)^2}{T-2}}}{\sqrt{\sum_{t=1}^{T} (t - \bar{t})^2}}$$

The threshold for the difference in slopes is defined as 5% of the standard error.

$$Threshold = 0.05(SE_{(\beta)})$$

If the difference in slopes exceeds the threshold, there is a significant difference between trends before and after imputations.

$$if \quad \left|\hat{\beta}_{(before)} - \hat{\beta}_{(after)}\right| \geq Threshold \quad \Longrightarrow \quad trends\ are\ significantly\ different$$

Verification: Interpolation or Elimination

If either distribution or trend are significantly different, a method of hybrid interpolation is performed to replace problematic values ($+\infty, -\infty, and\ NA$).

For each data point that requires imputation, linear interpolation using the neighboring valid data points is performed.

$$y_{(problematic)} = y_{(start)} + \frac{\left(t_{(problematic)} - t_{(start)}\right)\left(y_{(end)} - y_{(start)}\right)}{t_{(end)} - t_{(start)}}$$

If the slope between neighboring points is relatively constant, linear interpolation should suffice.

For each data point that requires imputation, cubic spline interpolation is performed.

$$S(t) = \alpha_{(0)} + \alpha_{(1)}(t) + \alpha_{(2)}(t^2) + \alpha_{(3)}(t^3)$$

The spline coefficients $\alpha_{(0)}, \alpha_{(1)}, \alpha_{(2)}, \alpha_{(3)}$ are calculated based on the surrounding data points ensuring smoothness at the boundaries.

Residual error between the imputed linear and non-linear values with the original values for neighborhood points is calculated.

$$\hat{\mu}_{(j)} = \hat{y}_{(j)} - y_{(j)}$$

To obtain a total mean of error, the sum of squared residuals for both linear and non-linear interpolations is calculated.

$$SSE_{(linear)} = \sum_{i=1}^{n} (\hat{\mu}_{(linear)})^2 \qquad SSE_{(spline)} = \sum_{i=1}^{n} (\hat{\mu}_{(spline)})^2$$

If $SSE_{(linear)}$ is smaller than $SSE_{(spline)}$, retain linear interpolation. If $SSE_{(spline)}$ is smaller than $SSE_{(linear)}$, retain spline (non-linear) interpolation.

$$if \quad SSE_{(linear)} < SSE_{(spline\,)} \implies y_{(problematic)} = \hat{y}_{(linear)}$$

$$if \quad SSE_{(spline)} < SSE_{(linear)} \implies y_{(problematic)} = \hat{y}_{(spline)}$$

Both Distribution and Trend analysis are performed again. If statistical properties are still significantly different, variable j is eliminated.

$$if \quad t\big(p_{(value)}\big)_{(new\ imputation)} < 0.05 \quad \implies \quad j \notin V \in D \quad otherwise \quad j \in V \in D$$

The treated variables that remained in the set will proceed to the next stage to determine its suitability as an input feature.

Outliers: No Treatment Needed

The proposed hybrid model is designed for high-precision financial time series forecasting at using return-based inputs.

Given this structural setup, outlier treatment would severely impair the model's performance, predictive accuracy, and interpretability for the following reasons.

Loss of Informational Asymmetry in Heavy-Tailed Distributions

Financial return distributions are non-Gaussian, heavy-tailed, and often exhibit power-law properties.

Standard outlier treatment techniques implicitly assume an underlying normality or a well-defined statistical threshold, which is inherently flawed for financial time series.

In this hybrid architecture, where xLSTM captures long-term dependencies, TFT extracts dynamic temporal patterns, and XGBoost exploits non-linear interactions, removing or modifying extreme observations disrupts the statistical structure of volatility clustering and fat tails.

Structural Breaks and Regime Shifts Obfuscation

Financial markets undergo regime shifts, structural breaks, and latent state transitions, which are often reflected in extreme returns.

The Meta Learner, integrating deep learning representations with gradient boosting structure, relies on outliers to recognize rare but critical market behaviors.

Suppressing these values effectively smooths out market anomalies, eliminating the key volatility dynamics that xLSTMs and TFT are specifically engineered to model.

Distortion of Heteroskedasticity and Auto-Correlated Residual Structures

Returns exhibit conditional heteroskedasticity and temporal autocorrelation. xLSTM and TFT rely on capturing these nonlinear dependencies, while XGBoost exploits feature interactions from lagged return distributions.

Modifying outliers skews the tail dependency structure, leading to a misrepresentation of variance persistence.

Since the ensemble model leverages recurrent and attention-based mechanisms to adapt to market state transitions, outlier removal breaks the integrity of learned feature mappings, degrading predictive stability.

Adverse Impact on Bayesian Optimization and Meta-Learning Calibration

The Meta Learner integrates multi-model predictions via weighted ensembling, where XGBoost dynamically adjusts weights based on relative model confidence.

In an adversarial market environment with stochastic volatility, extreme observations serve as crucial calibration anchors.

Filtering these values distorts the Bayesian posterior distributions underlying the Meta Learner's dynamic weight allocation, leading to suboptimal parameter tuning and deteriorated robustness in adversarial conditions.

Stationarity

Stationarity refers to a property of a time series where its statistical characteristics, such as mean, variance, and autocovariance, remain constant over time.

This implies that the process generating the data does not exhibit trends or seasonal effects, making it more amenable to modeling and forecasting.

Potential Impact

xLSTM

The lack of stationarity affects LSTM by introducing variability in the statistical properties and temporal dependencies of the input data.

Non-stationary trends and regime shifts distort the first and second-order statistics, hindering the model's ability to accurately capture long-term dynamics.

This leads to unstable gradients during backpropagation due to unstructured fluctuations in the data, complicating consistent weight updates within the LSTM cells. As a result, the model learns spurious relationships

Temporal Fusion Transformers

Non-stationary inputs interfere with the internal temporal dynamics of Temporal Fusion Transformers, leading to unstable attention scores and unreliable temporal context modeling.

Since TFTs utilize learned time-varying attention to modulate both short-term and long-term dependencies, statistical shifts distort the gating behavior and temporal relevance estimation.

As a result, the model is prone to capturing transient noise as persistent signal, degrading forecast accuracy.

XGBoost

XGBoost models assume a quasi-stationary data-generating process for consistent gradient-based optimization.

Non-stationary trends and structural breaks degrade tree stability by shifting the distribution of features and targets across boosting iterations.

This results in splits that are optimal only locally in time, causing temporal drift in model behavior and diminishing predictive robustness.

Addressing the Problem: Augmented Dickey-Fuller Test

The Augmented Dickey-Fuller (ADF) test is a statistical test used to determine whether a time series is stationary or non-stationary.

It is based on the idea that a non-stationary time series contains a unit root in its generating process, implying that the series does not revert to its mean over time.

- Null Hypothesis $\left(H_{(0)}\right)$: The time series has a unit root (non-stationary).
- Alternative Hypothesis $\left(H_{(1)}\right)$: The time series does not have a unit root (stationary).

The general form of the regression used in the ADF test is as follows:

$$\Delta y_{(t)} = \alpha + \beta t + \rho y_{(t-1)} + \sum_{i=1}^{p} \gamma_{(i)} \Delta y_{(t-i)} + \epsilon_{(t)}$$

Where:

- $y_{(t)}$: Value of the time series at time t.
- $\Delta y_{(t)}$: Difference between two consecutive values of the series (change over time).
- α: Intercept or constant of the regression.
- βt: Linear trend in the series.
- $\rho y_{(t-1)}$: Relationship between the previous and current value of the series (unit root).
- $\sum_{i=1}^{p} \gamma_{(i)} \Delta y_{(t-i)}$: Lag term; sum of the differences of the time series in previous periods.
- $\epsilon_{(t)}$: Error or random variability unexplained by the model.

The model is estimated using Ordinary Least Squares (OLS) to obtain the coefficients ρ, $\gamma_{(i)}$, α, and β. The general formula for the Ordinary Least Squares method is as follows:

$$\widehat{coef} = \arg\min_{coef} \sum_{t=1}^{T} \left(\Delta y_{(t)} - \left(\alpha + \beta t + \rho y_{(t-1)} + \sum_{i=1}^{p} \gamma_{(i)} \Delta y_{(t-i)} \right) \right)^2$$

- Where $\widehat{coef}$ represents the estimated value of the corresponding coefficient.

It is verified whether the coefficient ρ is significantly different from 0. To do this, the t-statistic is calculated.

$$t_\rho = \frac{\hat{\rho}}{\sqrt{\frac{\hat{\sigma}^2}{\sum_{t=1}^{T} (y_{(t-1)} - \bar{y})^2}}}$$

The obtained value is compared with the critical values from the Dickey-Fuller distribution, which depend on the sample size and the desired level of significance (0.05).

$$if\colon \ t_{(p)} > C_{(0.05)} \implies reject\ H_{(0)}$$

$$if\colon \ t_{(p)} \leq C_{(0.05)} \implies fail\ to\ reject\ H_{(0)}$$

The p-value is calculated to determine the evidence against the null hypothesis. A p-value less than 0.05 indicates that there is sufficient evidence to reject the null hypothesis.

$$p_{(value)} = P\left(\left|t_{(\rho)}\right| > \left|\hat{t}_{(\rho)}\right|\right)$$

$$if\colon \ p_{(value)} < 0.05 \implies reject\ H_{(0)}$$

$$if\colon \ p_{(value)} \geq 0.05 \implies fail\ to\ reject\ H_{(0)}$$

Failing to reject the null hypothesis states that the autoregressive coefficient $\rho = 0$, indicating the presence of a unit root.

This suggests that the time series does not have a significant autoregressive relationship and follows a non-stationary stochastic process, where the values do not revert to a constant mean.

Rejecting the null hypothesis implies that p is significantly different from zero, indicating that the series is stationary and does not exhibit a unit root.

$$fail\ to\ reject\ H_{(0)} = (\rho = 0) \implies series\ is\ non\ stationary$$

$$reject\ H_{(0)} = (\rho \not\equiv 0) \implies series\ is\ stationary$$

In the context of feature evaluation and selection for the models, the test will be performed for each potential input variable.

If the variable's time series is non-stationary, it will be discarded as an input feature for the models.

If the time series of the variable is stationary, it will proceed to the next stage to determine its suitability as an input feature.

$$\forall\, X_{(n)} \in \{input\ features\}:$$

$$ADF\ test\ for\ time\ series\ \{y_{(t)}^{X}\}$$

$$if: \rho_{(X)} = 0 \quad \Longrightarrow \quad eliminate\ X_{(n)}$$

$$if: \rho_{(X)} \neq 0 \quad \Longrightarrow \quad keep\ X_{(n)}$$

Multivariate Dependence

Multivariate dependence refers to the statistical interdependence between multiple predictor variables, where the joint distribution of one or more variables is influenced by the values of others.

This includes both linear and non-linear relationships, characterized by correlations or functional dependencies that exist across multiple dimensions of the feature space.

Potential Impact

xLSTM

Multivariate dependence in LSTM induces correlations among input features, which results in high collinearity at the input layer.

This causes the hidden states and cell states to become entangled with redundant information, leading to ineffective gradient propagation during backpropagation in the vanishing gradient problem.

The internal gating mechanisms (input, forget, and output gates) fail to correctly differentiate feature contributions, causing biased updates to weights and hidden states.

The LSTM struggles to effectively capture temporal dependencies, leading to poor representation learning.

Temporal Fusion Transformers

In Temporal Fusion Transformers, dependency among predictors disrupts both the static covariate encoders and the dynamic variable selection mechanisms.

As the gating layers attempt to assign feature-level relevance, highly dependent inputs produce unstable or redundant selection weights, leading to inflated attention scores.

This undermines the interpretability of the temporal attention maps and degrades the model's ability to disentangle distinct causal contributions across time.

XGBoost

In XGBoost, high multivariate dependency leads to repeated or redundant feature splits across multiple trees, reducing ensemble diversity and interpretability.

Highly dependent predictors generate shallow gain improvements during node splits, causing inefficiencies in the boosting process and diminishing model compactness.

Moreover, feature importance scores become biased, misrepresenting true predictive influence and introducing fragility to input perturbations.

Addressing the Problem: Mutual Information Elimination Test

Mutual Information is a measure used to quantify the amount of information that one random variable contains about another.

The mutual information between variables is symmetrical, meaning that mutual information between $X_{(i)}$ and $Y_{(i)}$ is the same as between $Y_{(i)}$ and $X_{(i)}$.

$$I\left(X_{(n)}; Y_{(i)}\right) = I\left(Y_{(i)}; X_{(n)}\right)$$

Mutual Information: Predictors with Target

Mutual information is calculated between every variable $X_{(n)} \in V \in D$ and the target $Y_{(i)}$.

$$\forall\, X_{(n)} \in V \in D:$$

$$I\left(X_{(n)}; Y_{(i)}\right) = \int p(x,y) \log\left(\frac{p(x,y)}{p(x)p(y)}\right) dxdy$$

Where:

- $p(x,y)$ is the joint probability density function of $X_{(n)}$ and $Y_{(i)}$.
- $p(x)$ and $p(y)$ are the marginal probability density functions of $X_{(n)}$ and $Y_{(i)}$, respectively.

Output leads to a set of values $\left\{I\left(X_{(1)}; Y_{(i)}\right), I\left(X_{(2)}; Y_{(i)}\right), \ldots, I\left(X_{(n)}; Y_{(i)}\right)\right\}$, where each entry represents the mutual information between a particular predictor variable $X_{(n)}$ and the target variable $Y_{(i)}$.

A matrix is constructed where the first and only column contains the mutual information values $I\left(X_{(n)}; Y_{(i)}\right)$ for each $X_{(n)}$.

Mutual Information: Predictors with Predictors

Mutual Information is calculated between every pair of predictor variables $X_{(i)} \in P \in D$.

$$\forall\, (X_{(n)}, X_{(m)}) \in P \in D:$$

$$I\left(X_{(n)}; X_{(m)}\right) = \int p\left(x_{(n)}, x_{(m)}\right) \log\left(\frac{p\left(x_{(n)}, x_{(m)}\right)}{p\left(x_{(n)}\right)p\left(x_{(m)}\right)}\right) dx_{(n)}dx_{(m)}$$

Output leads to a set of values $\left\{I\left(X_{(1)}; X_{(2)}\right), I\left(X_{(1)}; X_{(3)}\right), \ldots, I\left(X_{(n)}; X_{(m)}\right)\right\}$, where each entry represents the mutual information between a particular every pair of predictor variables $X_{(i)}$.

A matrix is constructed where columns contains the mutual information values $I\left(X_{(n)}; X_{(m)}\right)$ for each $(X_{(n)}, X_{(m)})$ where $n \neq m$.

Evaluation & Elimination

The computed mutual information values $I\left(X_{(n)};Y_{(i)}\right)$ and $I\left(X_{(n)};X_{(m)}\right)$ are normalized to a range between 0 and 1, where 1 corresponds to the mutual information between two identical random variables $I\left(Z_{(i)};Z_{(i)}\right)$.

$$\forall\, I\left(X_{(n)};Y_{(i)}\right) \,\wedge\, I\left(X_{(n)};X_{(m)}\right):$$

$$I\left(X_{(n)};Y_{(i)}\right) = \frac{I\left(X_{(n)};Y_{(i)}\right)}{I\left(Z_{(i)};Z_{(i)}\right)} \quad ; \quad I\left(X_{(n)};X_{(m)}\right) = \frac{I\left(X_{(n)};X_{(m)}\right)}{I\left(Z_{(i)};Z_{(i)}\right)}$$

A combined matrix is constructed, where the first column contains the mutual information values $I\left(X_{(n)};Y_{(i)}\right)$ and the rest contain the mutual information values $I\left(X_{(n)};X_{(m)}\right)$ for each corresponding pair $(X_{(n)},X_{(m)})$ where $n \neq m$.

$$I(X_n;Y_i) \cup I(X_n;X_m)_{matrix} = \begin{array}{c|cccccc} \boldsymbol{V} & \boldsymbol{Y_i} & \boldsymbol{X_1} & \boldsymbol{X_2} & \boldsymbol{X_3} & \dots & \boldsymbol{X_n} \\ \hline \boldsymbol{X_1} & I(X_1;Y_i) & 1 & I(X_1;X_2) & I(X_1;X_3) & \dots & I(X_1;X_n) \\ \boldsymbol{X_2} & I(X_2;Y_i) & I(X_2;X_1) & 1 & I(X_2;X_3) & \dots & I(X_2;X_n) \\ \boldsymbol{X_3} & I(X_3;Y_i) & I(X_3;X_1) & I(X_3;X_2) & 1 & \dots & I(X_3;X_n) \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \boldsymbol{X_n} & I(X_n;Y_i) & I(X_n;X_1) & I(X_n;X_2) & I(X_n;X_3) & \dots & 1 \end{array}$$

The mutual information $I\left(X_{(n)};X_{(m)}\right)$ between each pair $(X_{(n)},X_{(m)})$ where $n \neq m$ in the matrix is compared. If the mutual information $I\left(X_{(n)};X_{(m)}\right)$ is greater or equal to 0.5, a second comparison between $I\left(X_{(n)};Y_{(i)}\right)$ and $I\left(X_{(m)};Y_{(i)}\right)$ is performed.

The variable $X_{(i)} \in V \in D$ that shares the greater mutual information with $Y_{(i)}$ is retained, the other is eliminated.

$$\forall\, I\left(X_{(n)};X_{(m)}\right) \geq 0.5:$$

$$\max_{I\left(X_{(i)};Y_{(i)}\right)} \left(I\left(X_{(n)};Y_{(i)}\right), I\left(X_{(m)};Y_{(i)}\right)\right) \implies \textit{corresponding } X_{(i)} \textit{ remains}$$

$$\min_{I\left(X_{(i)};Y_{(i)}\right)} (I\left(X_{(n)};Y_{(i)}\right), I\left(X_{(m)};Y_{(i)}\right)) \implies \textit{corresponding } X_{(i)} \textit{ is eliminated}$$

The final set $V \in D$ of variables $X_{(i)}$ is composed of those, where for every pair $(X_{(n)},X_{(m)})$, mutual information $I\left(X_{(n)};X_{(m)}\right)$ does not exceeds 0.5, and every $X_{(i)}$ shares the maximum mutual information with $Y_{(i)}$.

Variables in the set proceed to the next stage to determine its suitability as an input feature.

Causal Dependence

Causal dependence refers to a direct influence of one variable on another, where changes in the independent variable induce changes in the dependent variable.

This can be linear or nonlinear, with the relationship characterized by a specific functional form.

It can also involve multidimensional interactions, where multiple variables influence each other across different dimensions, creating complex dependencies.

Potential Impact

LSTM

The absence of causal dependence with the target in an LSTM impedes the model's ability to propagate relevant information through the hidden states.

This disrupts the gradient flow during backpropagation and exacerbating issues such as gradient vanishing or explosion.

Without a causal relationship, the model fails to establish meaningful temporal dependencies between input features and the target variable, leading to ineffective updates.

This results in suboptimal parameter convergence, inefficient learning of long-term dependencies, and a failure to capture dynamic temporal structures, degrading LSTM's performance.

Temporal Fusion Transformers

The absence of causal dependence undermines the Temporal Fusion Transformer's ability to allocate meaningful variable importance scores through its variable selection network.

Without true causal links, the model's static encoders and gating mechanisms overfit to spurious correlations, leading to noisy attention distributions.

This distorts the temporal relevance captured by the multi-head attention layers, resulting in an inflated sensitivity to non-informative features and decreased interpretability of the learned temporal structures.

XGBoost

For XGBoost, the absence of causal dependence causes the algorithm to form splits on predictors that exhibit statistical association but offer no real explanatory power.

These spurious features inflate tree depth without improving generalization, increasing the risk of unnecessary model complexity.

The final ensemble becomes sensitive to noise and data irregularities, with inflated feature importance scores that obscure the true structure of the underlying process.

Addressing the Problem: Integrated Causality Combined Rank (ICCR)

Integrated Causality Combined Rank is an approach designed to identify optimal input variables for the algorithms prior to model training.

The methodology integrates three causality models: Transfer Entropy, Causal Bayesian Networks, and Convergent Cross Mapping (Causal Inference).

The use of multiple causality models to identify relationships between predictors and the target variable offers the following advantages:

- Capture Non-linear Relationships.
- Handle Stochastic Nature and High Dimensionality.
- Minimize False Positives.
- Provide Multi-scale Temporal Insights.

The results from the individual models will be integrated using a Combined Rank approach to facilitate the selection of the most relevant input variables.

The following outlines the detailed process.

Transfer Entropy

Entropy is defined as a measure of disorder, uncertainty, or information within a system in thermodynamics, information theory, and statistical mechanics.

It is related to the number of possible configurations or the level of disorder in a system: “The invisible force that brings disorder to the universe”.

1. Entropy in Thermodynamics: Boltzmann’s Entropy

In thermodynamics, entropy measures the disorder or randomness of a physical system. It is associated with the number of possible microstates within a given system.

The relationship is expressed by the formula:

$$S = k_{(B)} \ln \Omega$$

- Where S represents the entropy of the system, $k_{(B)}$ is the Boltzmann constant $(1.380649 \times 10^{-23}\, J/K)$, and Ω denotes the number of possible microstates.

2. Entropy in Information Theory: Shannon’s Entropy

In information theory, entropy is used to measure the uncertainty associated with a random variable.

Shannon's entropy for continuous variables deals with probability distributions defined over a continuous set of values.

The formula for continuous entropy is:

$$H(X) = -\int_{-\infty}^{\infty} fx(x) \log_{(2)} fx(x) dx$$

- Where $H(X)$ represents the entropy of the continuous variable X, $fx(x)$ is the probability density function of X, and the logarithm is taken in base 2, making the unit of $H(X)$ the bits.

If the distribution $fx(x)$ is more concentrated, the entropy will be lower because the uncertainty about the value of X decreases.

3. Entropy in Statistical Mechanics: Gibbs Entropy

Gibbs entropy is an extension of the Boltzmann entropy for systems that are not in equilibrium, and it is used for stochastic or non-equilibrium systems.

The formula is as follows:

$$S = -k_{(B)} \sum_{i=1}^{\Omega} p_{(i)} \ln p_{(i)}$$

- Where S is the entropy of the system, $p_{(i)}$ is the probability of the i-th microstate, and Ω is the total number of possible microstates.

The Gibbs entropy accounts for the probability of each non-equably probable microstate and quantifies the disorder of the system.

4. Entropy in Quantum Information Theory: Von Neuman Entropy

Von Neumann entropy is the quantum extension of Shannon's entropy, used to describe the degree of uncertainty in a quantum system.

$$S(\rho) = -\mathrm{Tr}(\rho \log \rho)$$

- Where ρ is the density matrix of the quantum system, Tr denotes the trace of the matrix (the sum of the diagonal elements), and $\log \rho$ is the logarithm of the matrix ρ.

Von Neumann Entropy measures the quantum uncertainty of a quantum state. If ρ represents a pure state, then $S(\rho) = 0$.

Entropy for the model: Transfer Entropy

1. Why Transfer Entropy

Subsequently, various branches of entropy have been defined across multiple scientific disciplines.

However, these measures presents limitations when applied to causal analysis in continuous, stationary time series; they fail to account for directional and asymmetric information transfer between variables.

Transfer Entropy quantifies the directional flow of information between variables, measuring the reduction in uncertainty of a target variable based on the past states of a source variable.

2. Transfer Entropy: Application

Transfer Entropy effectively addresses the limitations present in other entropy measures. To calculate Transfer Entropy, Shannon Entropy is used.

Shannon Entropy alone, is inherently symmetric and does not account for the directional flow of information. This symmetry limits its capability to infer causality in stochastic systems.

When applied in Transfer Entropy, it incorporates temporal dependencies and directionality, making it suitable for detecting causal relationships between time series.

The process is as follows:

First, the differential Shannon Entropy is computed for the target Stock:

$$H(Y) = -\int_{-\infty}^{\infty} p(y_{(i)}) \log_{(2)} p(y_{(i)}) dy$$

- Where Y represents the target time series, $p(y_{(i)})$ is the probability density function of Y, and $H(Y)$ quantifies the total uncertainty inherent in Y.

The conditional Shannon Entropy is computed to quantify the information transfer from Y to X, measuring the reduction in entropy of Y given X:

$$H(Y|X) = -\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} p\left(y_{(i)}, x_{(i)}\right) \log_{(2)} p\left(y_{(i)}\middle|x_{(i)}\right) dydx$$

- Where $p(y_{(i)}, x_{(i)})$ is the joint probability density function of X and Y, and $p\left(y_{(i)}\middle|x_{(i)}\right)$ is the conditional probability density function Y of given X.

Since $H(Y|X) \leq H(Y)$, a reduction in entropy signifies that knowledge of X contributes to a more precise characterization of Y.

Transfer Entropy is then computed as:

$$T_{(X\rightarrow Y)} = H(Y) - H(Y|X)$$

- Where $T_{(X\rightarrow Y)}$ represents the Transfer Entropy from X to Y. The difference between $H(Y)$ and $H(Y|X)$ quantifies the reduction in entropy of Y when is X included.

A larger difference implies that X significantly reduces the stochastic uncertainty in Y, capturing the directionality of information transfer.

Given that Transfer Entropy is asymmetric $T_{(X\rightarrow Y)} \neq T_{(Y\rightarrow X)}$, it provides a framework for detecting causality rather than mere dependence or correlation.

This asymmetry arises from the fact that the conditioning structure explicitly accounts for the influence of past values of X on Y, while the reverse computation captures a potentially different influence of Y on X.

To incorporate temporal dependencies, Transfer Entropy is computed across time lags:

$$T_{(X\rightarrow Y)} = H\left(Y_{(t)}\right) - H\left(Y_{(t)}|X_{(t-k)}\right)$$

or equivalently,

$$T_{(X\rightarrow Y)} = H\left(Y_{(t)}|Y_{(t-k)}\right) - H\left(Y_{(t)}|Y_{(t-k)}, X_{(t-k)}\right)$$

- Where k denotes the lag.

The formulation ensures that only the unique contribution of X to Y beyond its own past values is captured.

The resulting transfer entropy value is continuous over $[0, \infty]$, with higher values indicating greater information transfer.

Quantification allows for an explicit evaluation of the extent to which X contributes to reducing the uncertainty in Y, thus providing a measure of directional influence in target Stock's complex stochastic systems.

The process is repeated for all the possible input variables $X_{(n)}$.

Since this process is repeated for all candidate explanatory variables $X_{(n)}$ with respect to Y, normalization is necessary to ensure comparability.

A normalized transfer entropy measure is defined as:

$$T'_{(X \to Y)} = \frac{T_{(X \to Y)}}{\max\limits_{T_{X \to Y}}(T_{(X_{(1)} \to Y)}, T_{(X_{(2)} \to Y)}, T_{(X_{(3)} \to Y)} \cdots T_{(X_{(n)} \to Y)}}$$

- Where the maximum transfer entropy observed across all input variables is used as a scaling factor.

Normalization ensures that $T'_{(X \to Y)}$ remains within the range $[0,1]$, where 0 indicates no information transfer and 1 corresponds to the highest detected entropy transfer.

Causal Bayesian Networks

Bayesian Networks

A Bayesian Network (BN) is a probabilistic graphical model that represents dependencies among a set of random variables using a Directed Acyclic Graph (DAG). It provides a compact and structured representation of the joint probability distribution (JPD), enabling efficient inference.

A Bayesian Network is formally defined

$$\mathcal{B} = (G, P)$$

- Where $G = (V, E)$ is a Directed Acyclic Graph (V represents the nodes and E represents the set of directed edges encoding probabilistic dependencies), and P is a set of Conditional Probability Distributions (CPDs), encoding the local conditional dependencies.

A DAG is a graph $G = (V, E)$ where:

- Each node V represents a random variable.
- Each directed edge E encodes a conditional dependency.

The graph is acyclic, meaning there exists no directed path forming a cycle. Mathematically, a DAG satisfies the following condition: If there exists a directed path from $X_{(a)}$ to $X_{(b)}$, then there cannot be a directed path from $X_{(b)}$ back to $X_{(a)}$.

The joint probability distribution factorizes according to the structure of the DAG as:

$$P\big(X_{(1)}, X_{(2)} \dots X_{(3)}\big) = \prod_{i=1}^{n} P(X_{(i)} | \mathrm{Pa}\big(X_{(i)}\big))$$

- Where $\mathrm{Pa}\big(X_{(i)}\big)$ denotes the set of parent nodes of $X_{(i)}$ in the DAG.

Given the DAG $A \longrightarrow B \longrightarrow C$, the joint probability decomposes as:

$$P(A, B, C) = P(A)P(B|A)P(C|B)$$

Factorization is computationally efficient compared to direct representation, where the full JPD would require $O(2^n)$ parameters for binary variables.

Example: Consider three variables: E = "Has flu", F = "Has fever", T = "Test is positive". The relationships can be structured as:

$$E \rightarrow F, \quad E \rightarrow T$$

Which results in the factorization:

$$P(E, F, T) = P(E)P(F|E)P(T|E)$$

- Where $P(E)$ is the marginal probability of having flu, $P(F|E)$ is the conditional probability of having fever giving flu, and $P(T|E)$ is the conditional probability of the test being positive given flu.

The output of the Bayesian Network is determined by Bayesian Inference. Inference involves computing posterior probabilities given observed evidence using Bayes' Theorem:

$$P(H|E) = \frac{P(E|H)P(H)}{P(E)}$$

- Where $P(H|E)$ is the probability of the hypothesis after observing the evidence, $P(E|H)$ is the likelihood of evidence given hypothesis, $P(H)$ is the prior probability of the hypothesis, and $P(E)$ is the marginal likelihood.

Building a Bayesian Network follows these structured steps:

1. Definition of the variables as nodes
2. Construction of the Directed Acyclic Graph
3. Specification of Conditional Probability Distribution
4. Factorization of Joint Probability Distribution
5. Bayesian Inference
6. Interpretation of the output: Before the test $P(H) = 0.1$ and after the test $P(H|E) = 0.5$.

Causal Bayesian Networks

Standard Bayesian Networks model probabilistic dependencies via conditional probability distributions (CPDs), allowing inference based on observed evidence. However, they do not distinguish between correlation and causation.

Causal Bayesian Networks (CBNs) resolved the limitation by incorporating the do-operator $do(X)$. It simulates an external intervention on a system.

By this, do-operator serves pre-existing causal dependencies from the parent-nodes, allowing estimation of the direct causal effect of X on Y and the elimination of confounding influence.

Observational probability:

$$P(Y|X) = \sum_{Z} P(Y|X,Z)P(Z)$$

- Represents passive inference based on co-occurrence patterns in data.

Interventional Probability:

$$P(Y|do(X)) = \sum_{Z} P(Y|X,Z)P(Z)$$

- Differs as it removes confounding effects by simulating a controlled manipulation of X.

The Backdoor Adjustment Criterion corrects for confounding for the correct estimation of $(Y|do(X))$:

$$P\big(Y\big|do(X)\big) = \sum_{Z} P(Y|X,Z)P(Z)$$

- Where Z represents confounders, variables that simultaneously influence both X and Y. The formula ensures the effect measured is due to X alone.

The application of $do(X)$ removes all incoming edges to X in the DAG. This modifies the factorization structure, yielding a new, intervention-specific probability distribution.

The structured step by step is as follows:

1. Definition of the variables as nodes
   - *Identical as Standard Bayesian Networks.*
2. Construction of the Directed Acyclic Graph
   - *Arcs represents causal mechanisms, not correlations.*
3. Specification of Conditional Probability Distribution
   - *Includes both direct influences and potential confounders.*
4. Factorization of Joint Probability Distribution
   - *Before Intervention:*

$$P\big(X_{(1)}, X_{(2)}, X_{(3)} \dots X_{(n)}\big) = \prod_{i} P(X_{(i)}|Pa\big(X_{(i)}\big))$$

   - *After intervention $do(X)$:*

$$P\left(X_{(1)}, X_{(2)}, X_{(3)} \ldots X_{(n)}\right) = \prod_{i, X_{(i)} = X} P(X_{(i)} | Pa\left(X_{(i)}\right))$$

- $P(X|Pa(X))$ *is eliminated, as* X *is now externally set.*

5. Bayesian Inference
   - *Apply do-operator (Backdoor Adjustment). Computation of* $P(Y|do(X))$ *using backdoor adjustment:*

$$P\left(Y\middle|do(X)\right) = \sum_{Z} P(Y|X,Z)P(Z)$$

6. Interpretation of the output
   - $P(Y|X)$ describes association but $P(Y|do(X))$ quantifies causal impact X on Y.

Implementation

A detailed explanation of the process and each part of the implementation process will be provided below to ensure an understanding of the approach.

1. Step 1: Import data frame

Hypothetical code assumes a pandas data frame is already provided. It is the reference data frame of all the candidates input features and the target stock.

2. Step 2: Learn Dag Structure

The structure of the Bayesian Network is determined using the Hill Climbing Search (HC) algorithm.

HC is a greedy optimization method that iteratively modifies the network structure by adding, removing, or reversing edges in the Directed Acyclic Graph (DAG).

Search is guided by the Bayesian Information Criterion (BIC) score, which ensures that the model balances complexity with explanatory power.

Resulting model object contains the learned causal structure, which is printed to verify the discovered relationships.

3. Step 3: Learn Conditional Probability Distributions

A Bayesian model is created using the structure learned in the previous step.

Maximum Likelihood Estimator (MLE) is then applied to learn the Conditional Probability Distributions (CPDs) from the dataset.

For each node in the DAG, the probability of its values is estimated based on its parent nodes, allowing for probabilistic inference.

4. Step 4: Model Verification (Checkpoint)

Validity of the Bayesian model is checked. Function *check_model()* ensures that the DAG remains acyclic and that all probability distributions are well-defined and sum to one.

If the structure is valid, the modeling process can proceed.

5. Step 5: Bayesian Inference Standard (Checking Correlation-Based Probability)

Standard Bayesian inference is performed using Variable Elimination to estimate conditional probabilities.

Involves the computation of $P(Y|X_{(i)} = 1)$ for each predictor X_i by querying the model with evidence that $X_{(i)} = 1$.

Output probabilities reflect statistical association but do not provide yet causal impact.

6. Step 6: Apply Backdoor Adjustment for Causal Effect

Backdoor Adjustment method is applied to determine the causal effect of each variable on Y. First step is to identify confounders.

These are found automatically by checking for common parents in the DAG structure. If confounders exist, the backdoor formula is applied:

$$P\big(Y\big|do(X)\big) = \sum_{Z} P(Y|X,Z)P(Z)$$

Where Z represents confounding variables. Probability $P(Y|X,Z)$ is computed for each possible value of Z, weighted by the probability $P(Z)$.

If no confounders exist, direct conditional probability $P(Y|X)$ is used.

7. Step 7: Sort Variables by Causal Impact

Variables are ranked by causal impact on Y. Computed causal effects are sorted in descending order. This helps distinguish between genuine causal relationships and correlations.

8. Sample Output and Interpretation:

$$P(Y \mid X1 = 1)\text{: } 0.72$$

$$P(Y \mid X2 = 1)\text{: } 0.51$$

$$P(Y \mid X3 = 1)\text{: } 0.65$$

$$P(Y \mid X4 = 1)\text{: } 0.80$$

$$P(Y \mid do(X1 = 1))\text{: } 0.68$$

$$P(Y \mid do(X2 = 1))\text{: } 0.49$$

$$P(Y \mid do(X3 = 1))\text{: } 0.60$$

$$P(Y \mid do(X4 = 1))\text{: } 0.78$$

First part of the output displays the standard Bayesian inference probabilities $P(Y|X = 1)$ which indicate correlations but do not distinguish causation from spurious associations.

Second part displays the causal effect estimations $P(Y|do(X=1))$ which isolate the direct causal impact of each variable on Y.

Convergent Cross Mapping

Complex systems, such as financial market time series, follow nonlinear and chaotic dynamics.

Classical causality methods, like Granger causality, are based on linear autoregressive models, assuming that causality manifests through delayed correlations.

These methods fail when relationships are nonlinear or when the system is multidimensional.

Convergent Cross Mapping (CCM) is a method that detects causality in nonlinear dynamic systems, based on embedding theory in phase space, following Takens' theorem.

CCM is based on the assumption that if X causes Y, the information from X must be embedded in the evolution of Y.

Phase Space

Phase space is an abstract representation that describes all possible states of a system at each moment in time.

For a one-dimensional variable $Y_{(t)}$, phase space is visualized in a two-dimensional graph if, for example, the position $X_{(t)}$ and velocity $A_{(t)}$ of a ball in rectilinear motion are considered.

For more complex systems, phase space is multidimensional, and the state of the system is represented as a vector:

$$S_{(t)} = (S_{1,(t)}, S_{2,(t)}, S_{3,(t)}, \dots, S_{d,(t)})$$

- Where d is the system's dimension and $S_{i,(t)}$ is the variable that describe the system at time t.

When not all variables are observed (stock systems), Takens' theorem allows the reconstruction of phase space from a single variable $Y_{(t)}$ through the embedding process.

Phase Space Embedding

Embedding consists of creating state vectors from a single observed variable. Each vector $V_{(t)}$ s is constructed using observations of the variable $Y_{(t)}$ at different time points.

State vectors take the form:

$$V_{(t)} = (Y_{(t)}, Y_{(t+\tau)}, Y_{(t+2\tau)}, \dots Y_{(t+(m-1)\tau)})$$

- Where m is the embedding dimension and τ is the time delay.

If time series $Y = \{Y_{(1)}, Y_{(2)}, Y_{(3)}, \dots, Y_{(n)}\}$, and $m = 3$ and $\tau = 1$, state vectors will take the form:

$$V = \{\big(Y_{(1)}, Y_{(2)}, Y_{(3)}\big), (Y_{(2)}, Y_{(3)}, Y_{(4)}), (Y_{(3)}, Y_{(4)}, Y_{(5)}) \dots\}$$

I.E.

$$V = \{(Y_{(t)}, Y_{(t+\tau)}, Y_{(t+2\tau)})\}$$

Process allows to represent the dynamic system in a multidimensional form, capturing its underlying dynamics.

Determination of m

The dimension m is the number of dimensions used in phase space to represent the system.

A small value of m leads to an incomplete reconstruction of the system, and a high value of m leads to leads to redundancy, as more dimensions does not contribute meaningful information about the system.

False Nearest Neighbors (FNN) is used to determine the value of m. It is based on the idea that if a point in the reconstructed phase space is too far from its neighbor in the real space, the dimension m is insufficient to represent the system.

The technique is used to identify and minimize "false neighbors," i.e., points that should be close in real space but are not due to insufficient embedding.

Determination of τ

Time delay τ controls the temporal separation between successive observations of the time series in the phase space reconstruction.

The optimal delay method is used with the autocorrelation function:

$$ACF_{(\tau)} = \frac{\sum_{t=1}^{N-\tau}(Y_{(t)} - \bar{Y})(Y_{(t+\tau)} - \bar{Y})}{\sum_{t=1}^{N-\tau}(Y_{(t)} - \bar{Y})^2}$$

- Where $ACF_{(\tau)}$ is the autocorrelation at delay τ, $Y_{(t)}$ is the value of the variable at time t, $\bar{Y}$ is the mean of the time series, and N is the total number of observations.

The value of τ is selected so that $ACF_{(\tau)}$ is sufficiently low, indicating that the observations no longer contain significant dependence after that delay.

Attractor

After the process, the phase space $M_{(Y)}$ is constructed such that:

$$M_{(Y_{(t)})} = (Y_{(t)}, Y_{(t+\tau)}, Y_{(t+2\tau)}, \dots, Y_{(t+(m-1)\tau)})$$

- Where $M_{(Y_{(t)})}$ is the reconstructed phase space of $Y_{(t)}$ (Shadow Manifold), m is the embedding dimension, and τ is the optimal time delay.

The attractor of the Shadow Manifold $M_{(Y_{(t)})}$ is the geometrical structure in the phase space which the system tends to evolve overtime.

It is a set of points to which the system approaches, and within which the system's trajectories remain as time progresses.

Shadow Manifold has the same geometrical structure as the original phase space, meaning it preserves the structure of the attractor of the original system, but it does not represent it completely.

Simply, the attractor of the Shadow Manifold $M_{(Y_{(t)})}$ is the "shape" in the phase state plane.

Prediction of $\hat{X}_{(t)}$ with $M_{(Y_{(t)})}$

Once the attractor of $M_{(Y_{(t)})}$, has been reconstructed, a specific point $Y_{(t)} = (Y_{(t)}, Y_{(t+\tau)}, Y_{(t+2\tau)}, \ldots, Y_{(t+(m-1)\tau)})$ is selected.

Nearest neighbors to the selected point $Y_{(t)}$ in the attractor of $M_{(Y_{(t)})}$ are identified by computing the distance between $Y_{(t)}$ and other points in the attractor.

K-nearest neighbors (k-NN) algorithm is employed, which calculates the distance between $Y_{(t)}$ and all other points in the phase space of Y selecting the closest points.

The distance between two points $Y_{(t)}$ and $Y_{(i)}$ in the phase space is computed using the Euclidean distance:

$$\varrho\left(Y_{(t)}, Y_{(i)}\right) = \sqrt{\sum_{j=1}^{m} (Y_{(t+j)} - Y_{(i+j)})^2}$$

- Where $\varrho\left(Y_{(t)}, Y_{(i)}\right)$ is the Euclidean distance between $Y_{(t)}$and $Y_{(i)}$, and j represents the position of the components within a state vector in the phase space.

Each nearest neighbor is assigned with a weight $\omega_{(i)}$ which depends on the distance between the points $Y_{(t)}$ and $Y_{(i)}$:

$$\omega_{(i)} = \frac{1}{\varrho\left(Y_{(t)}, Y_{(i)}\right)^{\rho}}$$

- Where ρ is a parameter that controls the decay of the weight $\omega_{(i)}$ as the distance increases. The optimal value of ρ is 2.

The prediction of X is performed using the actual values of the time series X that correspond to the neighbor points of $Y_{(t)}$ in the attractor of $M_{(Y_{(t)})}$.

The computation is a weighted combination of the values of $X_{(i)}$ using the weights $\omega_{(i)}$:

$$\hat{X}_{(t)} = \sum_{i=1}^{k} (\omega_{(i)}) X_{(i)}$$

- Where k is the number of nearest neighbors of $Y_{(t)}$, $X_{(i)}$ are the values of X corresponding to those neighbors, and $\omega_{(i)}$ are the weights previously calculated.

Causality evaluation

Evaluation of the causality of X and Y is conducted by calculating the mutual information between the real values of X and the predicted values of X ($\hat{X}_{(t)}$):

$$I\left(\hat{X}_{(t)}, X\right) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} p(\hat{X}_{(t)}, X)\log\left(\frac{p(\hat{X}_{(t)}, X)}{p\left(\hat{X}_{(t)}\right), p(X)}\right) d\hat{X}_{(t)} dX$$

If $I\left(\hat{X}_{(t)}, X\right)$ is high, it indicates that Y contains sufficient information about X to predict it, suggesting that X is causal for Y, thereby supporting causal inference.

CCM Process for the Model

For the model in question, the CCM process is as follows:

1. *Embedding Construction*

Realization of embedding for the target stock time series, generating the Shadow Manifold $M_{(Y_{(t)})}$ and its attractor.

2. *Cross Mapping*

For every possible input feature for the model $X_{(i)}$, causality with Y is tested. To determine if $X_{(i)} \longrightarrow Y$, $\hat{X}_{(t,i)}$ is calculated using $M_{(Y_{(t)})}$.

3. *Convergence of Causality score*

For every possible input feature for the model, mutual information between $X_{(i)}$ and $\hat{X}_{(t,i)}$ is calculated.

For the sake of interpretation, $I\left(\hat{X}_{(t)}, X\right)$ of every $X_{(i)}$ is normalized such that:

$$NMI\left(\hat{X}_{(t)}, X\right) = \frac{I\left(\hat{X}_{(t)}, X\right)}{\frac{H\left(\hat{X}_{(t)}\right) + H(X)}{2}}$$

- Where $H(X)$ and $H(\hat{X}_{(t)})$ are Shannon's entropy of X and $\hat{X}_{(t)}$correspondingly.

Normalization ensures mutual information $NMI\left(\hat{X}_{(t)}, X\right)$ for every $X_{(i)}$ lie between the interval $[0,1]$.

Causality Integration: Combined Rank

Various robust integration methods were evaluated. However, these approaches imposed probabilistic assumptions on inherently non-probabilistic scores, compromising their accuracy and distorting the optimal selection process.

Given the high dimensionality of the feature set and the need for a final filtering step to retain only the most relevant variables as input features for the model, the Combined Rank method was selected.

Despite its simplicity, this approach is robust as it:

- Avoids scale-dependent distortions.

- Leverages consensus across multiple methods.
- Ensures that only consistently high-ranking predictors are selected, reducing noise.

The process of the approach is as follows:

Given n predictors $X_{(1)}, X_{(2)}, X_{(3)}, \dots, X_{(n)}$ and three causal inference methods producing scores:

- $S_{(1)}(X_{(i)})$ from Transfer Entropy.
- $S_{(2)}(X_{(i)})$ from Causal Bayesian Networks.
- $S_{(3)}(X_{(i)})$ from Convergent Cross Mapping.

A ranking for each predictor $X_{(i)}$ within each method is constructed:

$$R_{(j)}\left(X_{(i)}\right) = Rank\ of\ X_{(i)}\ based\ on\ S_{(j)}$$

- Where $j \in \{1,2,3\}$ represent the causality method. The highest score in each method gets $rank = 1$, the second highest $rank = 2$, and so on.

The approach removes differences in magnitude and methodology across methods and makes comparable scores regardless of their original distribution.

Once the ranks are computed for each method, a combined rank score for each predictor is calculated using Borda Count method (Rank-based Voting):

$$R_{borda}\left(X_{(i)}\right) = \sum_{j=1}^{3}\left(N - R_{(j)}\left(X_{(i)}\right)\right)$$

- Where N is the total number of predictors.

Borda Count approach rewards predictors that consistently rank high across methods without disregarding the score of any of them individually.

A new rank is generated such that the predictor $X_{(i)}$ with the lowest $R_{borda}\left(X_{(i)}\right)$ score gets $rank = 1$, the second lowest $rank = 2$, and so on:

$$R_{final}\left(X_{(i)}\right) = Rank(R_{borda}\left(X_{(i)}\right))$$

- Where $R_{(borda)}\left(X_{(i)}\right)$ is the Borda count score of the predictor X_i, and the rank is assigned in ascending order of $R_{(borda)}\left(X_{(i)}\right)$, i.e., the smallest Borda Count receives $rank = 1$.

$R_{(final)}\left(X_{(i)}\right)$ will be used for the selection of the final features for the models in the Eschaton Selection Test.

Eschaton Selection Test (EST)

Unlike the detailed tests that follow daily retraining procedures for the dynamic updating of predictive models, the Eschaton Selection Test (EST) is executed only once. Its primary purpose is to assess the sensitivity of predictive performance with respect to the number of input features, $N_{(V)}$.

EST serves as a structural evaluation aimed at identifying the optimal relationship between model dimensionality and performance, P, grounded in the *Arattone Theorem for High Dimensionality*.

After performing the Arattone Relation analysis, the optimal number of input features, $N^*_{(V)}$ is determined.

The calculated ranking $R_{(final)}\left(X_{(i)}\right)$ orders all candidate input variables by their individual predictive power with respect to the target variable Y.

The variable corresponding to $N^*_{(V)}$ in $R_{(final)}\left(X_{(i)}\right)$ is the cutoff, so that only the top $N^*_{(V)}$ features are selected for the model.

Formally, if $N^*_{(V)} = N$, the selected set of input features is defined as:

$$\left\{X_{(1)}, X_{(2)} X_{(3)}, \dots, X_{(N)}\right\}$$

- Where $X_{(i)}$ denotes the i-th most predictive variable according to $R_{(final)}\left(X_{(i)}\right)$.

Modelling

Theoretical Framework

This section delineates the rationale behind the selection of the proposed architectures and their modifications, as well as the language that will be employed among the requisite proposed inputs to address the necessary requirements.

Python as the Main Engine

Python is unrivaled in numerical computation and data manipulation due to its mature libraries. These tools offer efficient vectorized operations and robust data structures that are indispensable for processing extensive time series data and executing advanced statistical analyses.

The availability of deep learning frameworks like TensorFlow and PyTorch is a critical factor for the proposed model.

They enable the construction, training, and fine-tuning of intricate neural architectures—including LSTM, Transformers and XGBoost's—by providing automatic differentiation and native GPU acceleration, thereby optimizing performance for high-dimensional modeling.

Python's ecosystem supports seamless integration of heterogeneous methodologies. Its interoperability facilitates the combination of ensemble approaches, gradient boosting, and meta-learning, ensuring that disparate model outputs can be aggregated cohesively.

The language benefits from an extensive repository of open-source tools. This ecosystem guarantees continuous library enhancements and comprehensive documentation.

Python also excels in system integration and deployment. Its compatibility with external APIs, such as Bloomberg, and its support for modern deployment frameworks enable the construction of scalable, real-time trading systems that seamlessly transition from prototyping to production environments.

Hybrid Bidirectional Stacked Long-Short Term Memory Model (xLSTM)

The Hybrid Bidirectional Stacked LSTM (Long Short-Term Memory) architecture represents an optimal solution for forecasting volatility and stock price at time $t + 1$ within financial time series, primarily due to its capabilities in capturing complex temporal dependencies and dynamic market structures.

Bidirectional Stacked LSTMs differ from standard unidirectional LSTMs by simultaneously processing data sequences in both forward and backward temporal directions.

This dual-directional processing allows the architecture to integrate historical and future contextual information at each timestep, enhancing the model's predictive capability and temporal context-awareness.

Within financial markets characterized by volatile and non-linear patterns, capturing the holistic sequence context is crucial.

By considering both preceding and succeeding temporal events, the bidirectional approach provides superior performance, particularly for volatility prediction where recent and immediate future data points heavily influence forecasting accuracy.

Stacked layers further augment the LSTM by increasing network depth, thus enhancing the model's representational power.

Each additional layer captures hierarchical temporal dependencies at different granularities, disentangling short-term fluctuations from long-term trends.

Consequently, stacked architectures have been empirically shown to deliver superior accuracy in complex time series tasks compared to shallow or single-layer networks.

Financial volatility is fundamentally stochastic, exhibiting regime shifts, conditional heteroskedasticity, and volatility clustering. Traditional linear models, such as ARIMA or GARCH variants, often fail to capture non-linear, long-range temporal dependencies and dynamic volatility regimes.

Conversely, the gating mechanisms in LSTM cells effectively handle gradient propagation across extensive temporal lags, mitigating the vanishing gradient problem that hampers traditional recurrent neural networks.

These gating mechanisms allow the model to selectively retain or discard temporal information, efficiently modeling volatility persistence and clustering phenomena intrinsic to financial time series.

Moreover, the bidirectional component substantially contributes to stock forecasting by embedding both past volatility shocks and anticipated market conditions into its temporal representations. This anticipatory context is critical for accurately capturing volatility spikes, regime transitions, and directional changes often precipitated by macroeconomic announcements or sudden market events.

The incorporation of an Expanding Window retraining strategy amplifies the effectiveness of Hybrid Bidirectional Stacked LSTMs.

This daily recalibration aligns model parameters continuously with the most recent market dynamics. Traditional static models exhibit a decline in predictive accuracy due to their inability to adapt rapidly to evolving market conditions.

In contrast, the Expanding Window methodology maintains sustained predictive robustness, particularly advantageous in volatile financial environments where statistical relationships rapidly evolve.

Furthermore, the strategic overfitting permitted by the Expanding Window methodology benefits the Hybrid Bidirectional Stacked LSTM model, as it allows precise adaptation to current market nuances.

Conventional models apply regularization methods to avoid overfitting to historical data, but within the context of financial forecasting, such overfitting—managed through continuous retraining—enables the model to capture short-lived market anomalies and swiftly incorporate new market regimes.

Regularization in this context would constrain adaptability, undermining predictive performance by prematurely limiting the model's responsiveness to emerging patterns.

Additionally, this architecture seamlessly integrates with ensemble methods and meta-learning frameworks. By generating intermediate forecasts that feed into subsequent ensemble layers, Hybrid Bidirectional Stacked LSTMs enhance predictive stability and reliability, particularly in multi-model setups.

Their outputs become robust input features for preceding models, further refining ensemble forecasts through hierarchical integration and feature augmentation.

Its bidirectional processing captures comprehensive temporal contexts, stacked layers effectively model hierarchical dependencies, and integration with dynamic retraining strategies like Expanding Window ensures persistent adaptability.

Vanilla Temporal Fusion Transformers (TFT)

The Vanilla Temporal Fusion Transformer (TFT) represents a sophisticated deep learning architecture specifically designed to address the complexities inherent in financial time-series forecasting, making it suited for predicting intraday stock price and volatility movements.

Its capability stems from its unique design integrating attention mechanisms, gating layers, and time-aware processing, ensuring superior predictive accuracy and interpretability in dynamic market environments.

Temporal Fusion Transformers (TFTs) excel due to their hybrid architecture, combining recurrent neural networks (RNNs) and transformer-based attention mechanisms.

This design allows TFT to leverage both the temporal dependencies modeled by RNNs and the context-aware focus provided by attention layers.

Specifically, the model employs a gated recurrent unit (GRU) encoder-decoder structure alongside static covariate encoders, variable selection networks, and temporal self-attention layers, which handle heterogeneity in temporal data, thus enhancing predictive performance for price and volatility forecasts.

The primary advantage of TFT arises from its dynamic variable selection mechanism.

By incorporating variable selection networks, the TFT dynamically identifies the most informative features at each time step.

TFT not only captures stable historical relationships but also adjusts swiftly to regime shifts and structural breaks characteristic of financial markets, thereby ensuring continuous alignment with real-time market dynamics.

Unlike conventional forecasting models that weigh historical observations uniformly or based solely on recency, TFT assigns context-specific weights to each past observation.

This self-attention capability enables the TFT to selectively focus on historical periods most similar to the current market conditions, significantly enhancing forecast accuracy.

Furthermore, TFT inherently handles both static and dynamic covariates. Static covariates, such as inherent stock characteristics, sector classifications, or macroeconomic indicators, provide foundational context and stability.

TFT simultaneously processes these covariates through distinct encoding layers, allowing it to integrate diverse information sources seamlessly, further enhancing prediction robustness.

The inherent gating mechanisms of TFT, particularly GLU (Gated Linear Units) and GRU-based gates, ensure the model effectively manages the flow of information through its layers.

These gates dynamically control the significance of inputs at each layer, preventing irrelevant or noisy information from distorting the predictive outputs.

Unlike many black-box deep learning architectures, TFT provides clear and intuitive insights into feature relevance and temporal dependencies via attention maps.

These maps explicitly visualize which time steps and features most significantly impact forecasts, offering valuable transparency for stakeholders and decision-makers.

Enhanced interpretability facilitates informed decision-making processes, crucial in intraday trading scenarios where understanding the underlying drivers of price and volatility movements is paramount.

Given its structure, TFT seamlessly integrates with the Expanding Window methodology utilized in the predictive ensemble framework.

This integration leverages the full capacity of TFT to dynamically learn temporal relationships without sacrificing adaptability or responsiveness to emerging market trends.

The Vanilla Temporal Fusion Transformer constitutes the optimal predictive architecture for forecasting intraday stock price and volatility, particularly within the proposed ensemble modeling framework.

Its dynamic variable selection, temporal attention mechanisms, interpretability, and compatibility with evolving market dynamics ensure maximal predictive accuracy, adaptability, and strategic value in real-time financial decision-making contexts.

Extreme Gradient Boosting (XGB)

To forecast the volatility and price and volatility of a stock at $t + 1$ in financial time series, employing Extreme Gradient Boosting (XGBoost) presents substantial theoretical and practical advantages.

Given the intrinsic characteristics of financial markets, notably their complex, nonlinear dependencies, volatility clustering, and conditional heteroscedasticity, the predictive power, flexibility, and robustness of XGBoost make it an optimal choice within the proposed model ensemble architecture.

XGBoost is a gradient-boosted decision tree framework, renowned for its superior predictive performance, and scalability.

It systematically addresses the limitations inherent in linear and parametric models by capturing intricate nonlinearities and interactions among features, attributes prevalent in financial data.

Its core strength emanates from sequentially building decision trees, each subsequent tree refining the residual errors of preceding trees through gradient descent optimization. Such iterative refinement significantly enhances accuracy by minimizing both bias and variance in forecasting outcomes.

XGBoost imposes no restrictive assumptions about data distribution or linearity. Consequently, it adeptly navigates the volatile and often chaotic patterns characteristic of financial time series, such as fat-tailed distributions, leptokurtosis, and volatility clustering

The gradient-boosting algorithm inherently optimizes for predictive stability by preventing undue sensitivity to random noise, which frequently pervades market data.

The algorithm's inherent adaptability and predictive agility arise from its flexible tree structure. Each tree adaptively partitions the input space to isolate regions with homogeneous responses, making XGBoost particularly potent in capturing localized volatility spikes and sharp price movements.

The application of ensemble learning—aggregating multiple decision trees—reduces variance, stabilizes predictions, and consequently yields robust estimates of future price and volatility dynamics.

The *Arattone Theorem for High Dimensionality* elucidates that beyond a specific threshold, increased dimensionality negatively impacts predictive accuracy due to redundant, irrelevant, or noisy variables. XGBoost inherently addresses this through automatic feature selection.

It assigns feature importance scores by evaluating gain-based metrics during tree construction, thereby naturally performing dimensionality reduction by isolating the most impactful predictors.

This process ensures that only features with strong causal and predictive links to stock price and volatility variations are retained, enhancing interpretability and model efficiency.

Financial markets inherently involve intricate dependencies, where macroeconomic indicators, technical metrics, and historical volatility measures interact nonlinearly.

The algorithm inherently accommodates these interactions through hierarchical decision paths within its trees, systematically capturing interaction effects up to arbitrary complexities without manual intervention.

XGBoost continuously recalibrates through the Expanding Window methodology, wherein daily model retraining integrates the latest market information. This dynamism allows XGBoost to promptly adjust model parameters, efficiently adapting to recent market dynamics and structural shifts without manual recalibration.

Incorporating the Expanding Window technique aligns seamlessly with XGBoost's sequential learning approach, thereby perpetually updating its internal decision criteria based on emerging patterns.

Within the proposed predictive ensemble architecture—comprising Hybrid Bidirectional Stacked LSTMs, Temporal Fusion Transformers, and Meta Learners—XGBoost occupies a critical role due to its classification capabilities.

It precisely estimates the probability of directional price movements, thereby generating actionable binary outputs.

Its probabilistic output not only complements continuous forecasts produced by deep learning models but also significantly enhances ensemble predictive diversity, reducing systematic biases inherent in homogeneous model compositions.

It adeptly navigates financial time series' inherent complexity, structural volatility, and informational asymmetries, thereby delivering accurate, robust, and actionable forecasts.

XGBoost Infused Meta Learner

The XGBoost-Infused Meta Learner operates as a second-order model within the ensemble, exclusively ingesting outputs from the Hybrid Bidirectional Stacked LSTM (xLSTM) and Vanilla Temporal Fusion Transformer (TFT) models.

Unlike conventional ensemble frameworks that may incorporate base-layer diversity through raw input features, this Meta Learner strategically restricts its input space to refined, pre-learned representations.

This design choice enables the meta-model to function as a higher-order synthesis engine, integrating the latent temporal dynamics captured by xLSTM with the attention-based contextual weighting of TFTs, thereby producing highly consolidated forecasts for price and volatility at $t + 1$.

The Meta Learner employs a regression-based Extreme Gradient Boosting (XGBoost) model. This internal logic distinguishes it from the classification-focused XGBoost, as its objective is not probabilistic inference, but direct point estimation.

Its function is not to classify directional movement (e.g., Up vs. Down) but to quantify the magnitude of expected returns and volatility shifts, enabling continuous-valued output appropriate for quantitative trading execution layers.

Base learners—xLSTM and TFT—act as low-bias, high-variance estimators trained to capture independent structural features from the same temporal domain. The Meta Learner then operates as a bias-correcting layer, using XGBoost's gradient-based optimization to weigh, calibrate, and reconcile the respective outputs.

In contrast to naive averaging or fixed weight ensembling techniques, the XGBoost regression logic introduces dynamic conditional weighting.

It learns when to prioritize weighting of the two input architectures based on historical performance patterns.

This conditional meta-learning behavior is particularly salient in financial time series, where performance attribution is regime dependent.

The ensemble structure benefits from the variance-reducing properties of meta-learning.

While both xLSTM and TFT architectures are inherently susceptible to temporal misalignments and overfitting on specific lag patterns—especially under high-frequency retraining via the Expanding Window approach—the Meta Learner filters such idiosyncrasies.

It calibrates predictions through residual minimization, aligning cross-model inconsistencies while preserving each base model's distinctive contribution to the aggregate forecast.

These techniques introduce sparsity and penalty-based feature weighting, effectively nullifying unreliable or overconfident signals from base models during certain market conditions.

The result is a dynamic, self-correcting system that resists overfitting despite daily retraining, enabling high reactivity to market regimes without sacrificing out-of-sample robustness.

Importantly, the Meta Learner does not interact with raw market inputs; its feature space is intentionally abstracted and fully dependent on learned signals.

This abstraction allows the Meta Learner to operate under a structurally lower variance, whereas LSTM and TFT must learn from noisy, high-dimensional financial features with inherent stochasticity and exogeneity, the Meta Learner functions in a noise-reduced signal space, optimizing its regression surface over smoother, architecture-defined input distributions.

The regression paradigm within the Meta Learner also aligns with the Expanding Window theorem.

As model inputs are refreshed daily with updated xLSTM and TFT forecasts, the Meta Learner undergoes daily retraining to realign prediction weights with the latest dynamics.

This process eliminates temporal staleness and ensures that the hierarchical error-correcting logic embedded in the XGBoost architecture reflects the evolving efficacy of each base learner.

In contrast to fixed-weight ensembles, the Meta Learner maintains high predictive elasticity across time, a property indispensable for intraday decision-making in non-stationary markets.

The XGBoost-Infused Meta Learner introduces a critical final layer of abstraction and performance reconciliation within the proposed ensemble framework.

Its structure capitalizes on the distinct strengths and error landscapes of xLSTM and TFT models while leveraging the non-parametric, regularized regression logic of XGBoost to produce robust, interpretable, and continuous forecasts.

Adaptive Hyper-parameter Recalibration via Optuna

Optuna, a Bayesian optimization framework tailored for define-by-run hyperparameter optimization, serves as the meta-heuristic engine that operationalizes second-order adaptability within the Expanding Window retraining loop.

Unlike conventional model tuning paradigms which decouple hyperparameter selection from temporal model evolution, Optuna integrates directly into the temporal retraining cycle, instantiating a closed-loop system wherein model architecture, optimization schema, and regularization dynamics are co-evolved in synchrony with market conditions.

This transition from static hyperparameter spaces to dynamic, context-conditioned search spaces introduces a higher-order learning structure to the ensemble—one that recursively refines its own structural priors in response to regime volatility, drifted distributions, and evolving feature importances.

Technically, Optuna leverages Tree-structured Parzen Estimators (TPE), a non-parametric density estimation approach that models the conditional probability distribution $p(x|y)$ of hyperparameters x given an objective value y, allowing it to approximate the likelihood of success across complex, non-convex, high-dimensional spaces.

This is critical in the ensemble architecture, where the search space includes both continuous (e.g., learning rates, dropout rates), categorical (e.g., activation functions, attention heads), and ordinal (e.g., number of LSTM layers, tree depth) parameters, each contributing nonlinear sensitivities to model variance and convergence behavior.

Additionally, Optuna employs adaptive pruning algorithms, notably the Median Stopping Rule and Successive Halving, which terminate underperforming trials during evaluation, minimizing unnecessary computational expenditure while maintaining search granularity over more promising hyperparameter regimes.

In the case of the Hybrid Bidirectional Stacked LSTM, Optuna continuously reoptimizes gating ratios, hidden state dimensionality, depth of stacking, bidirectionality toggling, learning rate schedulers (e.g., cosine annealing vs. exponential decay), and sequence truncation lengths, all of which modulate how temporal dependencies are hierarchically encoded.

Particularly, sequence horizon length—typically treated as exogenous—is re-evaluated per retraining iteration, thus allowing the model to dynamically recalibrate its memory depth according to recent volatility clustering and autocorrelation decay rates.

For the TFT, the search space becomes more intricate due to the presence of multi-branch input pipelines and attention layers.

Optuna fine-tunes GRU hidden units, temporal self-attention head count, gating threshold levels in GLUs, and variable selection network depth.

It also evaluates parameter interactions across temporal encoder-decoder stages, such as tuning separate dropout rates for static and time-varying covariate pipelines or assigning differentiated learning rates to encoder vs. decoder subnetworks via multi-optimizer policies.

In the XGBoost-Infused Meta Learner, Optuna refines both regularization parameters (L1 alpha, L2 lambda) and functional parameters (e.g., maximum depth, subsample, colsample_bytree, and minimum child weight), not as static values, but as temporally sensitive levers that respond to shifts in the residual error surface generated by the base models. In this context, the regression-based Meta Learner functions as a continuously reformulating estimator of model synergy.

As xLSTM and TFT outputs shift their error distributions in response to recent data, Optuna adjusts the Meta Learner's structure to recalibrate its weight assignment logic, re-tuning the ensemble's internal bias-variance trade-off at each iteration.

This avoids structural ossification—a key failure mode in ensembles operating over non-stationary domains—and transforms the Meta Learner into a stochastic optimizer over base model feature spaces, maintaining realignment without any manual intervention or human-in-the-loop governance.

Optuna's define-by-run flexibility is particularly relevant in this ensemble architecture because of its capacity to optimize conditional search spaces—i.e., hierarchically nested hyperparameters whose relevance is dependent on the activation of others.

For instance, in the TFT model, attention dropout is only meaningful if multi-head attention is enabled, and in the Meta Learner, tree depth becomes functionally relevant only under high error variance in base model deltas.

Optuna accommodates these conditional dependencies through its dynamic search graph, which constructs the hyperparameter topology on the fly at each retraining loop.

This is essential for replicable optimization under Expanding Window, as newly incorporated data may alter the marginal utility of entire branches of the hyperparameter tree.

From a theoretical standpoint, this design instantiates a meta-learning protocol, wherein not just weights and structures, but optimization strategies themselves evolve.

By embedding Optuna into the retraining pipeline of every model component, the ensemble constructs a hierarchical optimization lattice: local optimizers (Adam, RMSProp, gradient boosting) operate at the parametric level, while Optuna governs the meta-structure of hyperparameter surfaces, and the Expanding Window governs the temporal boundary of data regimes.

This multi-tiered optimization framework aligns with the logic of endogeneity and self-reference in complex systems theory, where agents (models) adapt not only to their environment (market data), but also to the structure of their own adaptation processes.

Finally, this approach establishes a foundation for adaptive regularization—a critical functionality in noisy, heteroskedastic financial domains.

Rather than pre-specifying L1 (Lasso) or L2 (Ridge) penalties, Optuna reselects these coefficients dynamically, balancing model complexity against predictive performance across regimes.

In periods of increased volatility, Optuna may reduce regularization to allow deeper trees or more expressive networks; in calmer regimes, it may impose tighter constraints to prevent instability.

This results in a non-parametric, volatility-sensitive regularization schema, one that augments robustness while preserving reactivity.

In totality, the integration of Optuna formalizes a meta-optimization layer that is continuous, architecture-aware, and temporally responsive.

It operationalizes the Expanding Window principle not merely as a data-sampling heuristic, but as a deep structural law governing every layer of the ensemble—from base model neurons and attention layers to tree splits and pruning thresholds.

This framework enables a form of intelligent architecture evolution, wherein the entire predictive system becomes an adaptive, self-refining entity, capable of learning not only from data, but from its own historical failures in navigating that data.

Diversity Advantage

The proposed architecture leverages model diversity not as a redundancy measure but as a deliberate mechanism for predictive enhancement and structural abstraction.

Central to this design is the strategic use of cross-validation across independent yet hierarchically dependent models, ensuring that the information encoded by each architecture is both preserved in isolation and leveraged in integration.

Diversity begins at the level of raw-data-trained models. Hybrid Bidirectional Stacked LSTM and Temporal Fusion Transformer (TFT) architectures are independently trained to predict next-day open-relative returns for the stock's high, low, and close prices.

These predictions are exclusively derived from raw engineered features as stated in the filtered via the *Data Analysis & Feature Selection* section. No inter-model dependencies are introduced at this stage, preserving the individual learning bias of each architecture.

This segregation ensures that each model specializes in capturing different temporal structures: LSTM targets sequential continuity and long-range dependencies, while TFT exploits short-term contextual weighting via attention mechanisms and dynamic covariate selection.

The outputs of these two primary architectures then become the inputs to a Meta Learner structured as a regression-based XGBoost model.

Rather than treating each forecast in isolation, the Meta Learner consolidates their predictive outputs to infer continuous-valued price return forecasts at $t + 1$.

At this stage, the learning process is restricted to model outputs only—raw features are excluded—to ensure that the Meta Learner's abstraction layer reflects the comparative weight of each underlying model's signal and not the original data redundancy.

Once this three-tier stock modeling structure is formed, an additional XGBoost model is trained using a combined feature set: the raw features, LSTM outputs, TFT outputs, and Meta Learner predictions.

This XGBoost classifier acts as a robustness enhancer, estimating the probability of positive directional return with all available predictive evidence.

Unlike the Meta Learner, this model incorporates raw feature interactions and nonlinear data-level partitions, allowing it to capture interaction effects missed by the preceding learners.

The role of this XGBoost classifier is not to overwrite the base models, but to supplement the directional component of price movement with probabilistic confidence under high-dimensional nonlinear transformations.

The outputs from all four architectures—LSTM, TFT, Meta Learner, and XGBoost—then serve as inputs to the volatility modeling process.

Separate Hybrid LSTM and TFT models are trained to predict next-day volatility measures (at-close, maximum, minimum). However, unlike the price models, these volatility models are trained not only on raw observed volatility data but also on the full output layer of the stock forecasting ensemble.

As such, the volatility prediction network is explicitly conditioned on the expected structure of price evolution.

This dependency hierarchy is essential for modeling implied variance, which is known to be path-dependent on return structure.

The final Meta Learner for volatility, implemented again using XGBoost regression, is trained on the volatility outputs of the LSTM and TFT models alone.

It does not interact with raw features or price-based outputs, maintaining its role as a high-level calibrator focused strictly on volatility-specific representations.

Importantly, there is no parallel XGBoost classifier for volatility.

The absence of a volatility-based probabilistic classifier reflects the design decision to avoid classification in domains where variance is inherently continuous and more prone to estimation noise when discretized.

All model outputs—raw returns, probabilities, and volatility forecasts—are preserved and stored in an indexed database with a consistent schema. Such architecture is necessary to support both retrospective backtesting and real-time strategy execution in live markets.

Each model's inclusion in the ensemble is therefore not incidental, but structurally motivated by its unique ability to abstract from a different input space or learning bias.

Cross-validation across time and architectures ensures that error surfaces are independently minimized, while ensemble integration enables joint signal maximization.

This compound system of separation and recombination defines the ensemble's core advantage: the ability to extract orthogonal yet complementary signal components that are individually predictive and jointly robust.

The graphical representation of this integration hierarchy is shown in *Figure 5*, which delineates the architecture's layered structure and training flow dependencies.

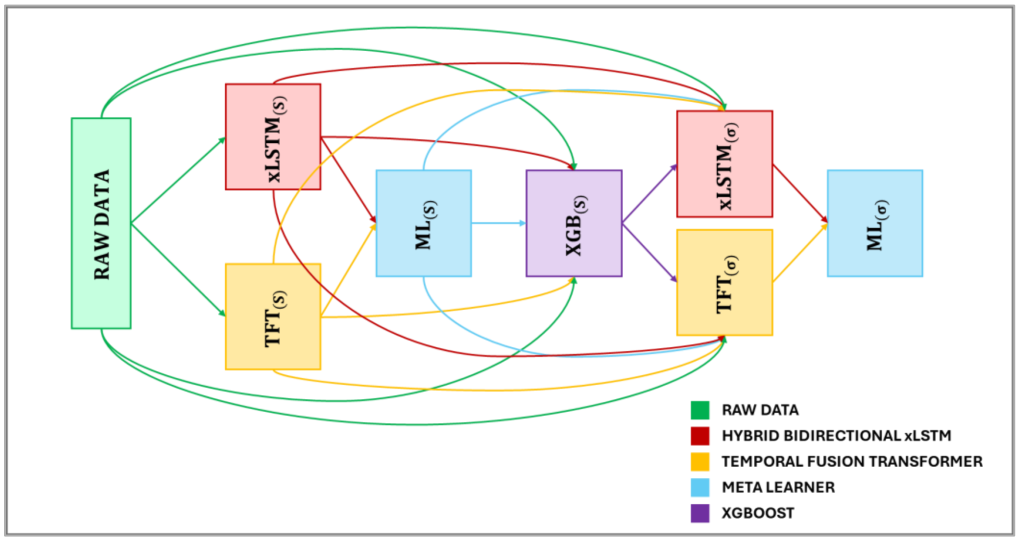

Figure 5

The ensemble's strength lies in its layered diversity—each model captures distinct market dynamics, and their structured integration ensures robustness across market regimes.

While higher layers abstract from raw features, they do not discard prior outputs; every prediction from every model, whether from raw-data-trained parents or their descendants, is retained as a validated source of information.

This design treats all outputs as autonomous signal contributors, enabling the system to leverage both hierarchical relationships and cross-model verification, maximizing informational efficiency without redundancy.

Modelling for Stock Price

This section presents the sample code implementations and backtesting procedures for each of the proposed predictive architectures. The objective is to offer technical transparency regarding the operational flow and structural logic behind the generation of next-day forecasts for the underlying stock price.

Each code sample is accompanied by a detailed breakdown of its constituent processes. Together, these examples illustrate how the theoretical framework is translated into executable routines suitable for both retrospective analysis and live deployment.

Hybrid Bidirectional Stacked LSTM (xLSTM)

Functional Sample Code

```python
import pandas as pd
import numpy as np
import optuna
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, BatchNormalization, Bidirectional
from tensorflow.keras.optimizers import Adam

###################################################
############ Step 1: Data Frame Set Up ############
###################################################

# Create a copy of the original dataframe for out-of-sample prediction
df_model_original = df_model.copy()

# Perform shift on the target variable to predict Y(t+1) using X(t)
df_model['Y'] = df_model['Y'].shift(-1)

# Remove the last row, as it has no target value
df_model = df_model.dropna()

############################################################
############ Step 2: Data Preprocessing ####################
############################################################

# Define the predictor variables (X) and the target variable (Y)
X = df_model.drop(columns=['Y'])
Y = df_model['Y']

# In this scheme, all data is used for training (without separating a test set)
X_train = X.values
Y_train = Y.values

# Reshape X_train to 3D for LSTM: (samples, timesteps, features)
X_train = X_train.reshape((X_train.shape[0], 1, X_train.shape[1]))

#####################################################
############ Step 3: Optuna Optimization ############
#####################################################

# Define the objective function for Optuna
def objective(trial):
    model = Sequential()

    # Hyperparameter optimization
    num_layers = trial.suggest_int('num_layers', 20, 60)
    units = trial.suggest_int('units', 100, 500)
    dropout_rate = trial.suggest_float('dropout_rate', 0.0, 0.05)
    learning_rate = trial.suggest_float('learning_rate', 1e-15, 1e-10, log=True)
    activation = trial.suggest_categorical('activation', ['relu', 'tanh', 'sigmoid',
                                                         'selu', 'elu'])
    batch_size = trial.suggest_categorical('batch_size', [16, 32, 64, 128, 256, 512])
    epochs = trial.suggest_int('epochs', 20, 80)
    loss_function = trial.suggest_categorical('loss_function', ['mse', 'mae', 'huber'])

    # Options for BatchNormalization and bidirectional LSTM
    use_batchnorm = trial.suggest_categorical('use_batchnorm', [True, False])
    use_bidirectional = trial.suggest_categorical('use_bidirectional', [True, False])

    ####################################################
    ############ Step 4: Model Architecture ############
    ####################################################

    # LSTM architecture with option for BatchNormalization and bidirectional layers
    for i in range(num_layers):
        lstm_layer = LSTM(units, activation=activation, return_sequences=(i < num_layers - 1))
        if use_bidirectional:
            lstm_layer = Bidirectional(lstm_layer)
        model.add(lstm_layer)
        if use_batchnorm:
            model.add(BatchNormalization())
        model.add(Dropout(dropout_rate))

    model.add(Dense(1))
    optimizer = Adam(learning_rate=learning_rate)
    model.compile(optimizer=optimizer, loss=loss_function)

    # Train the model on all available data (without separating a test set)
    history = model.fit(X_train, Y_train, epochs=epochs, batch_size=batch_size, verbose=0)

    # Evaluate on the same training set
    loss = model.evaluate(X_train, Y_train, verbose=0)
    return loss

# Execute optimization with Optuna
study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=100)

print("Mejor trial:")
trial = study.best_trial
print(trial.params)

##########################################################
############ Step 5: Definition of Best Model ############
##########################################################

# Get the best hyperparameters
best_params = study.best_params

# Build the model with the best hyperparameters
best_model = Sequential()
for i in range(best_params['num_layers']):
    lstm_layer = LSTM(best_params['units'], activation=best_params['activation'],
                      return_sequences=(i < best_params['num_layers'] - 1))
    if best_params['use_bidirectional']:
        lstm_layer = Bidirectional(lstm_layer)
    best_model.add(lstm_layer)
    if best_params['use_batchnorm']:
        best_model.add(BatchNormalization())
    best_model.add(Dropout(best_params['dropout_rate']))

best_model.add(Dense(1))
optimizer = Adam(learning_rate=best_params['learning_rate'])
best_model.compile(optimizer=optimizer, loss=best_params['loss_function'])

##########################################################
############ Step 6: Training Best Model #################
##########################################################

# Train the best model using all the training data
best_model.fit(X_train, Y_train, epochs=best_params['epochs'],
               batch_size=best_params['batch_size'], verbose=1)

################################################################################
############ Step 7: Out-of-Sample Prediction with Contextualization #########
################################################################################

# Use the last record from the original data (X_train) to contextualize the model
X_next = df_model_original.drop(columns=['Y']).iloc[-1:]

# Reshape X_next to 3D for LSTM input
X_next = X_next.values.reshape((X_next.shape[0], 1, X_next.shape[1]))

# Perform a forward pass with X_next so that the model "aligns" with the current state.
_ = best_model.predict(X_next)
prediction_next = best_model.predict(X_next)
print(f"Out of the sample prediction: {prediction_next}")
```

Step 1: Data Frame Set Up

The code creates a backup of the original dataframe to preserve unmodified data and then applies a temporal shift on the target variable so that each instance $X_{(t)}$ is paired with $Y_{(t+1)}$.

The final row is dropped to eliminate missing target values. This transformation ensures that the supervised learning problem is temporally consistent and accurately represents the forecasting horizon.

This approach deliberately uses the entire available historical data for training, foregoing the traditional train/test split to maximize the incorporation of recent dynamics.

Step 2: Data Preprocessing

Features and target variables are extracted from the shifted dataframe and converted into NumPy arrays.

The feature matrix is reshaped into a three-dimensional tensor with dimensions corresponding to (samples, timesteps, features) to meet the input requirements of an LSTM network.

This ensures that temporal dependencies are maintained, and the data preparation is consistent with the objective of forecasting the immediate next period based solely on the entire historical record.

Step 3: Optuna Optimization

Optuna is employed to optimize the hyperparameters of the LSTM model by minimizing the training loss over the entire dataset.

The objective function constructs a sequential model with a specified number of LSTM layers, units, dropout rate, learning rate, and optional regularization components (batch normalization and bidirectional wrappers).

The entire dataset is used for training in this context, which intentionally leads to overfitting for the purpose of maximizing short-term predictive accuracy in a continuously re-trained, expanding window framework.

Step 4: Model Architecture

The LSTM architecture is dynamically constructed based on the best hyperparameters obtained. Layers are sequentially added—each LSTM layer (potentially bidirectional) is followed by optional batch normalization and dropout.

A final dense layer outputs the prediction. This architecture is engineered not to generalize broadly but to capture the precise temporal patterns of the training data, intentionally overfitting the historical information to yield a highly specialized forecast for the imminent period.

Step 5: Definition of Best Model

The optimal hyperparameters are used to re-instantiate the final model with an identical architecture to that found during optimization.

This ensures that the final model is configured exactly as the best-performing model during the hyperparameter search, reinforcing consistency and reliability in its short-term predictions.

Step 6: Training Best Model

The final model is trained on the entire dataset without splitting out a test set. This deliberate design choice maximizes the utilization of all available data, intentionally overfitting to capture the most recent dynamics.

Such overfitting is not a drawback in this context because the system is designed for daily retraining; each new data point is incorporated in subsequent training cycles, thus continuously updating and refining the model’s predictions.

Step 7: Out-of-Sample Prediction with Contextualization

For forecasting $Y_{(t+1)}$, the most recent record from the original, unshifted dataset is extracted and reshaped to match LSTM input dimensions.

A forward pass is performed with this record to align the model’s internal state with the latest available data before generating the final prediction.

This method contextualizes the forecast by incorporating the current state, thereby enhancing responsiveness to recent trends in a real-time setting.

Backtesting Sample Code

```python
import pandas as pd
import numpy as np
import optuna
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout, BatchNormalization, Bidirectional
from tensorflow.keras.optimizers import Adam

####################################################
############ Step 1: Data Frame Set Up ############
####################################################

# Create a copy of the original dataframe for out-of-sample prediction
df_model_original = df_model.copy()

# Perform shift on the target variable to predict Y(t+1) using X(t)
df_model['Y'] = df_model['Y'].shift(-1)

# Remove the last row, as it has no target value
df_model = df_model.dropna().reset_index(drop=True)

# Prepare dataframe to store backtesting results
df_oots_predictions = pd.DataFrame(columns=['Index', 'Y_real', 'Y_pred'])

############################################################
############ Step 2: Backtesting with Expanding Window #####
############################################################

# Initial training size (first available window)
initial_size = 5000

# Loop through each expanding window and make prediction at t+1
for i in range(initial_size, len(df_model) - 1):

    # Slice data up to time t (exclude t+1)
    df_slice = df_model.iloc[:i].copy()

    # Define the predictor variables (X) and the target variable (Y)
    X = df_slice.drop(columns=['Y'])
    Y = df_slice['Y']

    # Reshape X_train to 3D for LSTM: (samples, timesteps, features)
    X_train = X.values.reshape((X.shape[0], 1, X.shape[1]))
    Y_train = Y.values

 #####################################################
 ############ Step 3: Optuna Optimization ############
 #####################################################

    # Define the objective function for Optuna
    def objective(trial):
        model = Sequential()

        # Hyperparameter optimization
        num_layers = trial.suggest_int('num_layers', 20, 60)
        units = trial.suggest_int('units', 100, 500)
        dropout_rate = trial.suggest_float('dropout_rate', 0.0, 0.05)
        learning_rate = trial.suggest_float('learning_rate', 1e-10, 1e-15, log=True)
        activation = trial.suggest_categorical('activation', ['relu', 'tanh', 'sigmoid',
                                                              'selu', 'elu'])
        batch_size = trial.suggest_categorical('batch_size', [16, 32, 64, 128, 256, 512])
        epochs = trial.suggest_int('epochs', 20, 80)
        loss_function = trial.suggest_categorical('loss_function', ['mse', 'mae', 'huber'])
        use_batchnorm = trial.suggest_categorical('use_batchnorm', [True, False])
        use_bidirectional = trial.suggest_categorical('use_bidirectional', [True, False])

        # LSTM architecture with option for BatchNormalization and bidirectional layers
        for j in range(num_layers):
            lstm_layer = LSTM(units, activation=activation,
                              return_sequences=(j < num_layers - 1))
            if use_bidirectional:
                lstm_layer = Bidirectional(lstm_layer)
            model.add(lstm_layer)
            if use_batchnorm:
                model.add(BatchNormalization())
            model.add(Dropout(dropout_rate))

        model.add(Dense(1))
        optimizer = Adam(learning_rate=learning_rate)
        model.compile(optimizer=optimizer, loss=loss_function)

        # Train the model on all available data (without separating a test set)
        model.fit(X_train, Y_train, epochs=epochs, batch_size=batch_size, verbose=0)

        # Evaluate on the same training set
        loss = model.evaluate(X_train, Y_train, verbose=0)
        return loss

    # Execute optimization with Optuna
    study = optuna.create_study(direction='minimize')
    study.optimize(objective, n_trials=100)

    best_params = study.best_params

    ##########################################################
    ############ Step 4: Definition of Best Model ############
    ##########################################################

    # Build the model with the best hyperparameters
    best_model = Sequential()
    for j in range(best_params['num_layers']):
        lstm_layer = LSTM(best_params['units'], activation=best_params['activation'],
                          return_sequences=(j < best_params['num_layers'] - 1))
        if best_params['use_bidirectional']:
            lstm_layer = Bidirectional(lstm_layer)
        best_model.add(lstm_layer)
        if best_params['use_batchnorm']:
            best_model.add(BatchNormalization())
        best_model.add(Dropout(best_params['dropout_rate']))

    best_model.add(Dense(1))
    optimizer = Adam(learning_rate=best_params['learning_rate'])
    best_model.compile(optimizer=optimizer, loss=best_params['loss_function'])

    ##########################################################
    ############ Step 5: Training Best Model #################
    ##########################################################

    # Train the best model using all the training data
    best_model.fit(X_train, Y_train, epochs=best_params['epochs'],
                   batch_size=best_params['batch_size'], verbose=0)

    ################################################################################
    ############ Step 6: Out-of-Sample Prediction with Contextualization #########
    ################################################################################

    # Use the last record from the original data (X_train) to contextualize the model
    X_next = df_model_original.drop(columns=['Y']).iloc[i].values.reshape(1, 1, -1)

    # Perform a forward pass with X_next so that the model "aligns" with the current state.
    _ = best_model.predict(X_next, verbose=0)
    prediction_next = best_model.predict(X_next, verbose=0)[0][0]

    # Retrieve the real Y(t+1)
    Y_real = df_model['Y'].iloc[i]

    # Store results
    df_oots_predictions = pd.concat([
        df_oots_predictions,
        pd.DataFrame({'Index': [i], 'Y_real': [Y_real], 'Y_pred': [prediction_next]})
    ], ignore_index=True)

# Print final out-of-sample prediction results
print(df_oots_predictions)
```

To enable sequential evaluation in the absence of a traditional test set, a time-aware expanding window strategy is implemented. This approach initializes the model using a fixed subset of historical observations and subsequently retrains it iteratively by incorporating one additional data point per cycle.

At each step t, the model is trained exclusively on data available up to time t, and the prediction is performed for $Y_{(t+1)}$, representing the true out-of-sample target. The corresponding predictor $X_{(t)}$ is introduced only after training, enabling contextualization with the most recent market conditions without contaminating the training window.

This temporal simulation loop mimics a realistic forecasting environment, where future outcomes are never exposed during training. It ensures that each predicted value is generated using only information that would have been available at that point in time, thus preserving the causal structure and avoiding lookahead bias.

The loop iterates from the end of the initial training window to the penultimate observation, producing a sequence of true out-of-sample predictions that can be used for backtesting. Without this structure, only a single forecast would be available for $Y_{(t+1)}$, limiting the model's evaluability.

Temporal Fusion Transformer (TFT)

Functional Sample Code

```python
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import optuna

####################################################
############ Step 1: Data Frame Set Up ############
####################################################

# Create a copy of the original dataframe for out-of-sample prediction
df_model_original = df_model.copy()

# Perform shift on the target variable to represent Y(t+1)
df_model['Y'] = df_model['Y'].shift(-1)

# Remove the last row, as it has no target value
df_model = df_model.dropna()

# All of df_model is used for training (without separating a test set)
X_train = df_model.drop(columns='Y')
y_train = df_model['Y']

print("Dimensions of X_train:", X_train.shape)
print("Dimensions of y_train:", y_train.shape)

############################################################
############ Step 2: Define Time Series Dataset ############
############################################################

class TimeSeriesDataset(Dataset):
    def __init__(self, X, y):
        # Convert to tensors and add sequence dimension: (n_samples, timesteps=1, n_features)
        self.X = torch.tensor(X.values, dtype=torch.float32).unsqueeze(1)
        self.y = torch.tensor(y.values, dtype=torch.float32)

    def __len__(self):
        # Return the total number of samples
        return len(self.X)

    def __getitem__(self, idx):
        # Retrieve the sample and corresponding target at the specified index
        return self.X[idx], self.y[idx]

################################################################
############ Step 3: Define VanillaTFT Architecture ############
################################################################
```

```python

class VanillaTFT(nn.Module):
    def __init__(self, input_dim, d_model, nhead, num_encoder_layers, dropout,
                 loss_type, activation, dim_feedforward):
        super(VanillaTFT, self).__init__()
        # Input projection: from input_dim to d_model
        self.input_projection = nn.Linear(input_dim, d_model)
        # Transformer encoder layer
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=d_model,
            nhead=nhead,
            dropout=dropout,
            activation=activation,
            dim_feedforward=dim_feedforward
        )
        # Stack the encoder layers
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer,
                                                         num_layers=num_encoder_layers)
        # Final layer for prediction (single output)
        self.fc = nn.Linear(d_model, 1)

        # Select the loss function based on loss_type
        self.loss_type = loss_type
        if loss_type == "MSE":
            self.criterion = nn.MSELoss()
        elif loss_type == "MAE":
            self.criterion = nn.L1Loss()
        else:
            self.criterion = nn.MSELoss()

    def forward(self, x):
        # x shape: (batch_size, timesteps, input_dim)
        x = self.input_projection(x)
        # Transpose to match Transformer input
        x = x.transpose(0, 1)
        out = self.transformer_encoder(x)
        # Take the output at the last timestep
        last_output = out[-1, :, :]
        output = self.fc(last_output)
        # Return a squeezed output to remove unnecessary dimensions
        return output.squeeze(1)

##################################################################
############ Step 4: Optuna Optimization and Training ############
##################################################################

def objective(trial):
    # Suggest hyperparameters for optimization
    d_model = trial.suggest_int("d_model", 16, 256)
    nhead = trial.suggest_int("nhead", 32, 64)
    # Ensure d_model is divisible by nhead
    if d_model % nhead != 0:
        raise optuna.TrialPruned("d_model must be divisible by nhead")
    num_encoder_layers = trial.suggest_int("num_encoder_layers", 16, 64)
    dropout = trial.suggest_float("dropout", 0.0, 0.05)
    lr = trial.suggest_float("lr", 1e-15, 1e-10, log=True)
    loss_type = trial.suggest_categorical("loss_type", ["MSE", "MAE"])
    batch_size = trial.suggest_int("batch_size", 32, 128, log=True)
    n_epochs = trial.suggest_int("n_epochs", 20, 80)
    activation = trial.suggest_categorical("activation", ["relu", "gelu"])
    dim_feedforward = trial.suggest_int("dim_feedforward", d_model * 2, d_model * 8)

    # Create DataLoader for training data
    train_dataset = TimeSeriesDataset(X_train, y_train)
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

    # Define the model with the suggested hyperparameters
    input_dim = X_train.shape[1]
    model = VanillaTFT(
        input_dim=input_dim,
        d_model=d_model,
        nhead=nhead,
        num_encoder_layers=num_encoder_layers,
        dropout=dropout,
        loss_type=loss_type,
        activation=activation,
        dim_feedforward=dim_feedforward
    )

    # Set the device to GPU if available, otherwise CPU
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    optimizer = optim.Adam(model.parameters(), lr=lr)

    # Training loop
    model.train()
    for epoch in range(n_epochs):
        for batch_x, batch_y in train_loader:
            # Move batch data to the appropriate device
            batch_x, batch_y = batch_x.to(device), batch_y.to(device)
            optimizer.zero_grad()
            preds = model(batch_x)
            loss = model.criterion(preds, batch_y)
            loss.backward()
            optimizer.step()

    # Evaluate the trial by calculating average loss on training data
    model.eval()
    total_loss = 0.0
    with torch.no_grad():
        for batch_x, batch_y in train_loader:
            batch_x, batch_y = batch_x.to(device), batch_y.to(device)
            preds = model(batch_x)
            loss = model.criterion(preds, batch_y)
            total_loss += loss.item()
    avg_loss = total_loss / len(train_loader)
    return avg_loss

# Execute optimization with Optuna
study = optuna.create_study(direction="minimize")
study.optimize(objective, n_trials=60)
```

```python

print("Best trial:")
trial = study.best_trial
print(trial.params)

####################################################################
############ Step 5: Train Best Model Using All Data ############
####################################################################

# Retrieve the best hyperparameters from the study
best_params = trial.params
d_model = best_params["d_model"]
nhead = best_params["nhead"]
num_encoder_layers = best_params["num_encoder_layers"]
dropout = best_params["dropout"]
lr = best_params["lr"]
loss_type = best_params["loss_type"]
batch_size = best_params["batch_size"]
n_epochs = best_params["n_epochs"]
activation = best_params["activation"]
dim_feedforward = best_params["dim_feedforward"]

# Define the model with the best hyperparameters
input_dim = X_train.shape[1]
model = VanillaTFT(
    input_dim=input_dim,
    d_model=d_model,
    nhead=nhead,
    num_encoder_layers=num_encoder_layers,
    dropout=dropout,
    loss_type=loss_type,
    activation=activation,
    dim_feedforward=dim_feedforward
)

# Set device and move the model to it
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

# Create DataLoader with all available data
train_dataset = TimeSeriesDataset(X_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

optimizer = optim.Adam(model.parameters(), lr=lr)

# Train the best model using the full training dataset
model.train()
for epoch in range(n_epochs):
    for batch_x, batch_y in train_loader:
        batch_x, batch_y = batch_x.to(device), batch_y.to(device)
        optimizer.zero_grad()
        preds = model(batch_x)
        loss = model.criterion(preds, batch_y)
        loss.backward()
        optimizer.step()

```

```
##################################################################################
############ Step 6: Out-of-Sample Prediction with Contextualization ############
##################################################################################

# To predict Y(t+1) use the last record from the original dataframe
X_next = df_model_original.drop(columns=['Y']).iloc[-1:]

# Convert to tensor with proper shape: (batch_size=1, timesteps=1, n_features)
X_next_tensor = torch.tensor(X_next.values, dtype=torch.float32).unsqueeze(1).to(device)

# Perform a forward pass to contextualize and obtain the final prediction
model.eval()
with torch.no_grad():
    # Contextualization - align the model with the current state
    _ = model(X_next_tensor)

    # Final Prediction
    y_pred = model(X_next_tensor)
    print(f"Out of the sample prediction): {y_pred.item():.4f}")
```

Step 1: Data Frame Set Up

A backup copy of the original unshifted dataset is created to enable out-of-sample inference using the most recent $X_{(t)}$. The target column $Y_{(t+1)}$ is shifted by -1 so that each input $X_{(t)}$ corresponds to the prediction target $Y_{(t+1)}$, conforming to a temporal forecasting framework.

The final row is dropped due to the absence of a valid label after shifting.

Importantly, the entire post-shift dataset is used for training—no holdout set is created—reflecting a deliberate decision to overfit the model on all available data, which is acceptable and even beneficial in a rolling re-training scenario where the model will be updated at each new timestep.

Step 2: Define Time Series Dataset

The input data is wrapped into a PyTorch Dataset object. Each input sample is reshaped to include a sequence length of 1, producing a tensor of shape $(n_{(samples)}, seq\_len = 1, n_{features})$.

This formatting satisfies the requirements of the Transformer encoder, which expects time-structured data. No temporal history beyond X_t is included, reinforcing that each prediction is made based solely on the current state, consistent with the single-step forecasting design.

Step 3: Define Vanilla TFT Architecture

A minimal Transformer-based forecasting architecture is constructed. Inputs are linearly projected to the model dimension $d_{(model)}$, processed by a configurable stack of Transformer encoder layers, and passed through a final linear output layer.

The model supports various activation functions, dropout rates, and attention configurations.

Crucially, the model is not optimized for generalization; instead, it is designed to memorize and exploit the structure of the current dataset fully, leveraging the overparameterization and depth of attention mechanisms to lock in the specific patterns present in the most recent state of the system.

Step 4: Optuna Optimization and Training

The entire training dataset is used to conduct an Optuna-based hyperparameter search. The objective function samples architectural and training parameters, fits the model over the full dataset, and computes the average loss—without using any validation split.

This overfitting is intentional and controlled: the model is not intended to generalize to unseen distributions but rather to produce one highly informed forecast before being retrained in the next cycle.

In this context, hyperparameter optimization ensures maximum short-term inference quality rather than long-term robustness.

Step 5: Train Best Model Using All Data

After determining the optimal architecture, the final model is retrained from scratch using the full dataset, with the best-found configuration. No data is reserved for testing or validation. This full-data training guarantees that the model's internal state reflects the entire available history.

This approach exploits overfitting as a feature, not a flaw, under the assumption that the model will be retrained daily using an expanding window that incorporates new observations sequentially.

Step 6: Out-of-Sample Prediction with Contextualization

The model performs a forward pass using the final unshifted record X_t from the original dataset, which was not used during training due to lack of $Y_{(t+1)}$.

This forward pass serves two functions: (1) contextualization, where the Transformer attends to the most up-to-date system state; and (2) prediction, where the output is treated as the inferred value of $Y_{(t+1)}$.

This design emulates real-world scenarios where the model has full knowledge of the past but must infer the future based solely on the most recent available data point. The lack of generalization is not a limitation, as the model will be retrained immediately after the ground truth for $Y_{(t+1)}$ becomes available.

Backtesting Sample Code

```
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, Dataset
import optuna

###################################################
############ Step 1: Data Frame Set Up ############
###################################################

# Create a copy of the original dataframe for out-of-sample prediction
df_model_original = df_model.copy()

# Perform shift on the target variable to represent Y(t+1)
df_model['Y'] = df_model['Y'].shift(-1)
```

```python

# Remove the last row, as it has no target value
df_model = df_model.dropna().reset_index(drop=True)

# Prepare dataframe to store backtesting results
df_oots_predictions = pd.DataFrame(columns=['Index', 'Y_real', 'Y_pred'])

############################################################
############ Step 2: Define Time Series Dataset ############
############################################################

class TimeSeriesDataset(Dataset):
    def __init__(self, X, y):
        # Convert to tensors and add sequence dimension: (n_samples, timesteps=1, n_features)
        self.X = torch.tensor(X.values, dtype=torch.float32).unsqueeze(1)
        self.y = torch.tensor(y.values, dtype=torch.float32)

    def __len__(self):
        # Return the total number of samples
        return len(self.X)

    def __getitem__(self, idx):
        # Retrieve the sample and corresponding target at the specified index
        return self.X[idx], self.y[idx]

################################################################
############ Step 3: Define VanillaTFT Architecture ############
################################################################

class VanillaTFT(nn.Module):
    def __init__(self, input_dim, d_model, nhead, num_encoder_layers, dropout,
                 loss_type, activation, dim_feedforward):
        super(VanillaTFT, self).__init__()
        # Input projection: from input_dim to d_model
        self.input_projection = nn.Linear(input_dim, d_model)
        # Transformer encoder layer
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=d_model,
            nhead=nhead,
            dropout=dropout,
            activation=activation,
            dim_feedforward=dim_feedforward
        )
        # Stack the encoder layers
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer,
                                                         num_layers=num_encoder_layers)
        # Final layer for prediction (single output)
        self.fc = nn.Linear(d_model, 1)

        # Select the loss function based on loss_type
        self.loss_type = loss_type
        if loss_type == "MSE":
            self.criterion = nn.MSELoss()
        elif loss_type == "MAE":
            self.criterion = nn.L1Loss()
        else:
            self.criterion = nn.MSELoss()

    def forward(self, x):
        # x shape: (batch_size, timesteps, input_dim)
        x = self.input_projection(x)
        # Transpose to match Transformer input
        x = x.transpose(0, 1)
        out = self.transformer_encoder(x)
        # Take the output at the last timestep
        last_output = out[-1, :, :]
        output = self.fc(last_output)
        # Return a squeezed output to remove unnecessary dimensions
        return output.squeeze(1)

#################################################################
############ Step 4: Backtesting with Expanding Window ##########
#################################################################

# Initial training size for expanding window
initial_size = 5000

# Simulate the evolution of the model through time (Machine of Time logic)
for i in range(initial_size, len(df_model) - 1):

    ###################################################
    ############ Step 5: Data Frame Set Up ############
    ###################################################

    # Use only data up to time i (excluding the record at i)
    df_slice = df_model.iloc[:i].copy()

    # All of df_slice is used for training (without separating a test set)
    X_train = df_slice.drop(columns='Y')
    y_train = df_slice['Y']

    print("Dimensions of X_train:", X_train.shape)
    print("Dimensions of y_train:", y_train.shape)

    ############################################################
    ############ Step 6: Define Time Series Dataset ############
    ############################################################

    train_dataset = TimeSeriesDataset(X_train, y_train)

    ##################################################################
    ############ Step 7: Optuna Optimization and Training ############
    ##################################################################

    def objective(trial):
        # Suggest hyperparameters for optimization
        d_model = trial.suggest_int("d_model", 16, 256)
        nhead = trial.suggest_int("nhead", 32, 64)
        if d_model % nhead != 0:
            raise optuna.TrialPruned("d_model must be divisible by nhead")
        num_encoder_layers = trial.suggest_int("num_encoder_layers", 16, 64)
        dropout = trial.suggest_float("dropout", 0.0, 0.05)
        lr = trial.suggest_float("lr", 1e-15, 1e-10, log=True)
        loss_type = trial.suggest_categorical("loss_type", ["MSE", "MAE"])
        batch_size = trial.suggest_int("batch_size", 32, 128, log=True)
        n_epochs = trial.suggest_int("n_epochs", 20, 80)
        activation = trial.suggest_categorical("activation", ["relu", "gelu"])
        dim_feedforward = trial.suggest_int("dim_feedforward", d_model * 2, d_model * 8)

        train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

        input_dim = X_train.shape[1]
        model = VanillaTFT(
            input_dim=input_dim,
            d_model=d_model,
            nhead=nhead,
            num_encoder_layers=num_encoder_layers,
            dropout=dropout,
            loss_type=loss_type,
            activation=activation,
            dim_feedforward=dim_feedforward
        )

        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        model.to(device)
        optimizer = optim.Adam(model.parameters(), lr=lr)

        model.train()
        for epoch in range(n_epochs):
            for batch_x, batch_y in train_loader:
                batch_x, batch_y = batch_x.to(device), batch_y.to(device)
                optimizer.zero_grad()
                preds = model(batch_x)
                loss = model.criterion(preds, batch_y)
                loss.backward()
                optimizer.step()

        model.eval()
        total_loss = 0.0
        with torch.no_grad():
            for batch_x, batch_y in train_loader:
                batch_x, batch_y = batch_x.to(device), batch_y.to(device)
                preds = model(batch_x)
                loss = model.criterion(preds, batch_y)
                total_loss += loss.item()
        avg_loss = total_loss / len(train_loader)
        return avg_loss

    study = optuna.create_study(direction="minimize")
    study.optimize(objective, n_trials=100)

    best_params = study.best_trial.params

    #################################################################
    ############ Step 8: Train Best Model Using All Data ############
    #################################################################

    input_dim = X_train.shape[1]
    model = VanillaTFT(
        input_dim=input_dim,
        d_model=best_params["d_model"],
        nhead=best_params["nhead"],
        num_encoder_layers=best_params["num_encoder_layers"],
        dropout=best_params["dropout"],
        loss_type=best_params["loss_type"],
        activation=best_params["activation"],
        dim_feedforward=best_params["dim_feedforward"]
    )

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    train_loader = DataLoader(train_dataset,
                              batch_size=best_params["batch_size"], shuffle=True)
    optimizer = optim.Adam(model.parameters(), lr=best_params["lr"])

    model.train()
    for epoch in range(best_params["n_epochs"]):
        for batch_x, batch_y in train_loader:
            batch_x, batch_y = batch_x.to(device), batch_y.to(device)
            optimizer.zero_grad()
            preds = model(batch_x)
            loss = model.criterion(preds, batch_y)
            loss.backward()
            optimizer.step()

    ################################################################################
    ############ Step 9: Out-of-Sample Prediction with Contextualization ############
    ################################################################################

    # To predict Y(t+1) use the record at index i from the original dataframe
    X_next = df_model_original.drop(columns=['Y']).iloc[i].values.reshape(1, 1, -1)
    X_next_tensor = torch.tensor(X_next, dtype=torch.float32).to(device)

    model.eval()
    with torch.no_grad():
        _ = model(X_next_tensor)
        y_pred = model(X_next_tensor).item()

    y_real = df_model['Y'].iloc[i]

    # Store the result
    df_oots_predictions = pd.concat([
        df_oots_predictions,
        pd.DataFrame({'Index': [i], 'Y_real': [y_real], 'Y_pred': [y_pred]})
    ], ignore_index=True)

# Print the full dataframe of out-of-the-sample predictions
print(df_oots_predictions)
```

To enable sequential evaluation in the absence of a traditional test set, a time-aware expanding window strategy is implemented. This approach initializes the model using a fixed subset of historical observations and subsequently retrains it iteratively by incorporating one additional data point per cycle.

At each step t, the model is trained exclusively on data available up to time t, and the prediction is performed for $Y_{(t+1)}$, representing the true out-of-sample target. The corresponding predictor $X_{(t)}$ is introduced only after training, enabling contextualization with the most recent market conditions without contaminating the training window.

This temporal simulation loop mimics a realistic forecasting environment, where future outcomes are never exposed during training. It ensures that each predicted value is generated using only information that would have been available at that point in time, thus preserving the causal structure and avoiding lookahead bias.

The loop iterates from the end of the initial training window to the penultimate observation, producing a sequence of true out-of-sample predictions that can be used for backtesting. Without this structure, only a single forecast would be available for $Y_{(t+1)}$, limiting the model's evaluability.

XGBoost Infused Meta Learner (ML)

Functional Sample Code

```
import optuna
import xgboost as xgb
import pandas as pd
from sklearn.metrics import mean_squared_error

####################################################
############ Step 1: Data Frame Set Up ############
####################################################

# Remove any NaN values.
df_meta = df_meta.dropna()

# Create a copy of the original dataframe for out-of-sample prediction (without applying shift)
df_meta_original = df_meta.copy()

# Apply shift to the target variable for alignment:
df_meta['Y'] = df_meta['Y'].shift(-1)
df_meta = df_meta.dropna()

####################################################
############ Step 2: Data Preprocessing ###########
####################################################

# Define the predictor variables (X) and the target variable (Y)
X = df_meta.drop(columns='Y')
y = df_meta['Y']

# In this approach, all available data is used for training (without a separate test set)
X_train = X.values
y_train = y.values

####################################################
############ Step 3: Optuna Optimization ##########
```

```python
######################################################

# The objective function trains on the entire dataset and returns the training error.
def objective(trial):
    params = {
        'n_estimators': trial.suggest_int("n_estimators", 300, 1500),
        'max_depth': trial.suggest_int("max_depth", 10, 30),
        'learning_rate': trial.suggest_float("learning_rate", 1e-3, 0.5, log=True),
        'subsample': trial.suggest_float("subsample", 0.7, 1.0),
        'colsample_bytree': trial.suggest_float("colsample_bytree", 0.7, 1.0),
        'gamma': trial.suggest_float("gamma", 0, 10),
        'reg_alpha': trial.suggest_float("reg_alpha", 0, 20),
        'reg_lambda': trial.suggest_float("reg_lambda", 0, 20),
        'min_child_weight': trial.suggest_int("min_child_weight", 1, 20),
        'tree_method': trial.suggest_categorical("tree_method", ["auto", "exact",
                                                                  "hist", "gpu_hist"]),
        'objective': trial.suggest_categorical("objective", ["reg:squarederror",
                                                              "reg:absoluteerror"]),
        'random_state': 42
    }

    model = xgb.XGBRegressor(**params)
    model.fit(X_train, y_train)
    preds = model.predict(X_train)
    loss = mean_squared_error(y_train, preds)
    return loss

# Execute optimization with Optuna
study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=100)

best_params = study.best_params
print("Best hyperparameters:")
print(best_params)

######################################################
############ Step 4: Training Final Model #########
######################################################

# Final training using all available data (without separating a test set)
final_model = xgb.XGBRegressor(**best_params)
final_model.fit(X_train, y_train)

######################################################
############ Step 5: Out-of-Sample Prediction #####
######################################################

# To predict Y(t+1)  use the last record of X from the original dataframe.
X_next = df_meta_original.drop(columns=['Y']).iloc[-1:]

# Perform the out-of-sample prediction
final_prediction = final_model.predict(X_next)[0]
print(f"Out of the sample prediction: {final_prediction:.4f}")
```

Step 1: Data Frame Set Up

The raw input data frame is cleaned to remove missing values. A full copy of the unshifted data is preserved for out-of-sample forecasting. A temporal shift is applied to the target column Y, aligning each input record $X_{(t)}$ with the label $Y_{(t+1)}$.

The final row is dropped due to an undefined target value. No data is held out for validation or testing.

The full shifted dataset is reserved exclusively for model fitting. This structure is intentional—it prioritizes capturing the most recent system dynamics at the expense of generalization, under the assumption that the model will be retrained as each new observation becomes available.

Step 2: Data Preprocessing

Features and targets are separated after the shift, and all available data is converted to NumPy arrays.

The entire dataset is designated as the training set. No test set is constructed. This approach facilitates complete exploitation of the available historical information, a deliberate design to encourage overfitting in service of hyper-local prediction accuracy, especially suitable for short-horizon, high-frequency inference with re-training after every timestep.

Step 3: Optuna Optimization

An Optuna-based hyperparameter search is performed. Each trial defines an XGBoost regression model with a specific configuration (e.g., depth, number of estimators, regularization strength, tree method).

Each candidate model is trained and evaluated on the same training data used for fitting. This strategy embraces overfitting rather than avoids it: the training loss becomes the optimization objective because the ultimate use case does not involve generalization across disjoint samples but rather precise next-step forecasting with model refresh cycles.

Step 4: Training Final Model

Once the best hyperparameters are identified, the final model is instantiated and trained on the entire dataset again—this time with the optimized configuration. No validation or testing data is held out.

This full-batch training strategy guarantees that the model has internalized the most complete and up-to-date representation of the time series. Since the next prediction will only be used once before the model is retrained, maximizing fit to the current sample is advantageous rather than problematic.

Step 5: Out-of-Sample Prediction

The model performs a forward inference using the last record from the original unshifted data frame, which corresponds to the most recent available input state $X_{(t)}$, for which the target $Y_{(t+1)}$ is unavailable.

This prediction is considered out-of-sample, not because it generalizes to unseen distributions, but because it is temporally beyond the training horizon. The model is thus contextualized with the present input, producing a one-step-ahead forecast that is tightly coupled to the latest system state.

This design simulates operational forecasting where predictions must be made before the ground truth is known, and the model is retrained as soon as the true value for $Y_{(t+1)}$ becomes available.

Backtesting Sample Code

```python
import optuna
import xgboost as xgb
import pandas as pd
from sklearn.metrics import mean_squared_error

#####################################################
############ Step 1: Data Frame Set Up ############
#####################################################

# Remove any NaN values.
df_meta = df_meta.dropna()

# Create a copy of the original dataframe for out-of-sample prediction (without applying shift)
df_meta_original = df_meta.copy()

# Apply shift to the target variable for alignment:
df_meta['Y'] = df_meta['Y'].shift(-1)
df_meta = df_meta.dropna().reset_index(drop=True)

# Prepare dataframe to store backtesting results
df_oots_predictions = pd.DataFrame(columns=['Index', 'Y_real', 'Y_pred'])

#####################################################
############ Step 2: Backtesting Loop #############
#####################################################

# Initial training size for expanding window
initial_size = 5000

for i in range(initial_size, len(df_meta) - 1):

    # Use data up to time i (exclusive)
    X_train = df_meta.iloc[:i].drop(columns='Y')
    y_train = df_meta.iloc[:i]['Y']

#####################################################
########### Step 3: Optuna Optimization ###########
#####################################################

    def objective(trial):
        params = {
            'n_estimators': trial.suggest_int("n_estimators", 300, 1500),
            'max_depth': trial.suggest_int("max_depth", 10, 30),
            'learning_rate': trial.suggest_float("learning_rate", 1e-3, 0.5, log=True),
            'subsample': trial.suggest_float("subsample", 0.7, 1.0),
            'colsample_bytree': trial.suggest_float("colsample_bytree", 0.7, 1.0),
            'gamma': trial.suggest_float("gamma", 0, 10),
            'reg_alpha': trial.suggest_float("reg_alpha", 0, 20),
            'reg_lambda': trial.suggest_float("reg_lambda", 0, 20),
            'min_child_weight': trial.suggest_int("min_child_weight", 1, 20),
            'tree_method': trial.suggest_categorical("tree_method", ["auto", "exact", "hist",
                                                                     "gpu_hist"]),
            'objective': trial.suggest_categorical("objective", ["reg:squarederror",
                                                                 "reg:absoluteerror"]),
            'random_state': 42
        }

        model = xgb.XGBRegressor(**params)
        model.fit(X_train, y_train)
        preds = model.predict(X_train)
        loss = mean_squared_error(y_train, preds)
        return loss

    study = optuna.create_study(direction='minimize')
    study.optimize(objective, n_trials=10)

    best_params = study.best_params

###################################################
########### Step 4: Train Final Model #############
###################################################

    final_model = xgb.XGBRegressor(**best_params)
    final_model.fit(X_train, y_train)

###################################################
########### Step 5: Out-of-Sample Prediction ######
###################################################

    # Predict Y(i+1) using X(i) from original unshifted data
    X_next = df_meta_original.drop(columns=['Y']).iloc[i:i+1]
    y_real = df_meta['Y'].iloc[i]
    y_pred = final_model.predict(X_next)[0]

    df_oots_predictions = pd.concat([
        df_oots_predictions,
        pd.DataFrame({'Index': [i], 'Y_real': [y_real], 'Y_pred': [y_pred]})
    ], ignore_index=True)

# Final print of all out-of-sample predictions
print(df_oots_predictions)
```

To enable sequential evaluation in the absence of a traditional test set, a time-aware expanding window strategy is implemented. This approach initializes the model using a fixed subset of historical observations and subsequently retrains it iteratively by incorporating one additional data point per cycle.

At each step t, the model is trained exclusively on data available up to time t, and the prediction is performed for $Y_{(t+1)}$, representing the true out-of-sample target. The corresponding predictor $X_{(t)}$ is introduced only after training, enabling contextualization with the most recent market conditions without contaminating the training window.

This temporal simulation loop mimics a realistic forecasting environment, where future outcomes are never exposed during training. It ensures that each predicted value is generated using only information that would have been available at that point in time, thus preserving the causal structure and avoiding lookahead bias.

The loop iterates from the end of the initial training window to the penultimate observation, producing a sequence of true out-of-sample predictions that can be used for backtesting. Without this structure, only a single forecast would be available for $Y_{(t+1)}$, limiting the model's evaluability.

Extreme Gradient Boosting (XGBoost)

Functional Sample Code

```python
import optuna
import xgboost as xgb
import pandas as pd
from sklearn.metrics import log_loss

####################################################
############ Step 1: Data Frame Set Up ############
####################################################

# Remove any NaN values.
df_meta = df_meta.dropna()

# Create a copy for out-of-sample prediction (without applying shift)
df_meta_original = df_meta.copy()

# Convert Y to binary:
df_meta['Y'] = (df_meta['Y'] > 0).astype(int)

# Apply shift to the target variable for alignment:
df_meta['Y'] = df_meta['Y'].shift(-1)
df_meta = df_meta.dropna()

#####################################################
############ Step 2: Data Preprocessing ############
#####################################################

# Define the predictor variables (X) and the target variable (Y)
X = df_meta.drop(columns='Y')
y = df_meta['Y']

# In this scheme, all available data is used for training (without a separate test set)
X_train = X.values
y_train = y.values

######################################################
############ Step 3: Optuna Optimization ############
######################################################

# The objective function trains on the entire dataset and returns the log loss
def objective(trial):
    params = {
        'n_estimators': trial.suggest_int("n_estimators", 300, 1500),
        'max_depth': trial.suggest_int("max_depth", 10, 30),
        'learning_rate': trial.suggest_float("learning_rate", 1e-15, 1e-10, log=True),
        'subsample': trial.suggest_float("subsample", 0.7, 1.0),
        'colsample_bytree': trial.suggest_float("colsample_bytree", 0.7, 1.0),
        'gamma': trial.suggest_float("gamma", 0, 10),
        'reg_alpha': trial.suggest_float("reg_alpha", 0, 20),
        'reg_lambda': trial.suggest_float("reg_lambda", 0, 20),
        'min_child_weight': trial.suggest_int("min_child_weight", 1, 20),
        'tree_method': trial.suggest_categorical("tree_method", ["auto", "exact",
                                                         "hist", "gpu_hist"]),
        # For binary classification, use 'binary:logistic'
        'objective': 'binary:logistic',
        'random_state': 42
    }

    model = xgb.XGBClassifier(**params)
    model.fit(X_train, y_train)

    # Obtain predicted probabilities for the positive class
    preds = model.predict_proba(X_train)[:,1]
    # Calculate the log loss (lower is better)
    loss = log_loss(y_train, preds)
    return loss

# Execute optimization with Optuna
study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=100)

best_params = study.best_params
print("Best hyperparameters:")
print(best_params)

###################################################
############ Step 4: Training Final Model #########
###################################################

# Final training using all available data (without a separate test set)
final_model = xgb.XGBClassifier(**best_params, objective='binary:logistic',
                                random_state=42)
final_model.fit(X_train, y_train)

###################################################
############ Step 5: Out-of-Sample Prediction #####
###################################################

# To predict Y(t+1) use the last record of X from the original dataset.
X_next = df_meta_original.drop(columns=['Y']).iloc[-1:]

# Perform the out-of-sample prediction: obtain the probability of class 1
final_prediction = final_model.predict_proba(X_next)[0,1]
print(f"Out of the sample prediction: {final_prediction:.4f}")
```

Step 1: Data Frame Set Up

The input data frame is sanitized by removing rows with missing values. A copy of the unshifted data is preserved for out-of-sample prediction. The target variable Y is transformed into a binary indicator: 1 if the original value was positive, 0 otherwise.

This recasts the problem as a binary classification task. A temporal shift is then applied to align each $X_{(t)}$ with its corresponding $Y_{(t+1)}$. The final row is dropped, as the shifted target is undefined.

No train/test split is performed: all data with a defined $Y_{(t+1)}$ is used for supervised learning. This setup enables direct modeling of event probability at the next timestep using the most recent available context.

Step 2: Data Preprocessing

After alignment, predictors and targets are extracted from the shifted dataset and converted to NumPy arrays. All records are included in the training set. No out-of-sample holdout is created.

This approach fully exploits available data to maximize exposure to short-term dynamics, under the principle that the model will be retrained with every new observation. Generalization is explicitly deprioritized in favor of immediate predictive accuracy.

Step 3: Optuna Optimization

The objective function constructs an XGBoost classifier using hyperparameters suggested by Optuna. The model is trained on the full dataset and evaluated using log loss, a probabilistic scoring rule that penalizes both incorrect classifications and overconfidence.

No validation set is used. Overfitting is acceptable and even encouraged because the model's purpose is to over-specialize to the current structure for immediate forecasting.

Each retraining event in the expanding window pipeline resets the model's exposure to reflect updated conditions, making generalization irrelevant for this use case.

Step 4: Training Final Model

With the optimal hyperparameters determined, a final XGBoostClassifier is trained on the entire available dataset. No validation or testing step is included. This ensures the model has full visibility into the most recent historical structure.

The objective is to tightly fit the dataset such that the resulting decision boundary is highly reactive to recent shifts in feature-behavior dynamics. This practice, while incompatible with static deployment settings, is optimal in scenarios where the model lifespan is short (single prediction) and rapidly refreshed.

Step 5: Out-of-Sample Prediction

The final prediction is generated using the last row from the original, unshifted dataset, which corresponds to $X_{(t)}$ (e.g., Sunday), for which the label $Y_{(t+1)}$ (e.g., Monday) is unknown.

The model, having never seen this record during training, performs a forward pass to estimate the probability of the binary outcome occurring at $t + 1$.

This prediction is made outside the training distribution in a temporal sense, not in a statistical sense. This final step embodies the idea of contextualization: aligning the model with the most recent input state to make a precise, localized forecast just before the true label becomes available and the model is retrained.

Backtesting Sample Code

```python
import optuna
import xgboost as xgb
import pandas as pd
from sklearn.metrics import log_loss

######################################################
############ Step 1: Data Frame Set Up ############
######################################################

# Remove any NaN values.
df_meta = df_meta.dropna()

# Create a copy for out-of-sample prediction (without applying shift)
df_meta_original = df_meta.copy()

# Convert Y to binary:
df_meta['Y'] = (df_meta['Y'] > 0).astype(int)

# Apply shift to the target variable for alignment:
df_meta['Y'] = df_meta['Y'].shift(-1)
df_meta = df_meta.dropna().reset_index(drop=True)

# Prepare dataframe to store backtesting results
df_oots_predictions = pd.DataFrame(columns=['Index', 'Y_real', 'Y_pred'])

######################################################
########### Step 2: Time Machine Loop #############
######################################################

# Initial training size for expanding window
initial_size = 5000

# Loop to simulate the expanding window prediction
total_rows = len(df_meta)
for i in range(initial_size, total_rows):

    # Slice data up to index i (exclude current for prediction)
    df_slice = df_meta.iloc[:i]
    X_train = df_slice.drop(columns='Y')
    y_train = df_slice['Y']

#####################################################
######## Step 3: Optuna Optimization ###############
#####################################################

    def objective(trial):
        params = {
            'n_estimators': trial.suggest_int("n_estimators", 300, 1500),
            'max_depth': trial.suggest_int("max_depth", 10, 30),
            'learning_rate': trial.suggest_float("learning_rate", 1e-3, 0.5, log=True),
            'subsample': trial.suggest_float("subsample", 0.7, 1.0),
            'colsample_bytree': trial.suggest_float("colsample_bytree", 0.7, 1.0),
            'gamma': trial.suggest_float("gamma", 0, 10),
            'reg_alpha': trial.suggest_float("reg_alpha", 0, 20),
            'reg_lambda': trial.suggest_float("reg_lambda", 0, 20),
            'min_child_weight': trial.suggest_int("min_child_weight", 1, 20),
            'tree_method': trial.suggest_categorical("tree_method", ["auto", "exact",
                                                                      "hist", "gpu_hist"]),
            'objective': 'binary:logistic',
            'random_state': 42
        }

        model = xgb.XGBClassifier(**params)
        model.fit(X_train, y_train)
        preds = model.predict_proba(X_train)[:,1]
        return log_loss(y_train, preds)

    study = optuna.create_study(direction='minimize')
    study.optimize(objective, n_trials=10)
    best_params = study.best_params

####################################################
########### Step 4: Train Final Model #############
####################################################

    final_model = xgb.XGBClassifier(**best_params, objective='binary:logistic', random_state=42)
    final_model.fit(X_train, y_train)

####################################################
######## Step 5: Out-of-Sample Prediction #########
####################################################

    X_next = df_meta_original.drop(columns=['Y']).iloc[i:i+1]
    y_real = df_meta['Y'].iloc[i]
    y_pred = final_model.predict_proba(X_next)[0,1]

    df_oots_predictions = pd.concat([
        df_oots_predictions,
        pd.DataFrame({'Index': [i], 'Y_real': [y_real], 'Y_pred': [y_pred]})
    ], ignore_index=True)

# Final output of all out-of-sample predictions
print(df_oots_predictions)

```

To enable sequential evaluation in the absence of a traditional test set, a time-aware expanding window strategy is implemented. This approach initializes the model using a fixed subset of historical observations and subsequently retrains it iteratively by incorporating one additional data point per cycle.

At each step t, the model is trained exclusively on data available up to time t, and the prediction is performed for $Y_{(t+1)}$, representing the true out-of-sample target. The corresponding predictor $X_{(t)}$ is introduced only after training, enabling contextualization with the most recent market conditions without contaminating the training window.

This temporal simulation loop mimics a realistic forecasting environment, where future outcomes are never exposed during training. It ensures that each predicted value is generated using only information that would have been available at that point in time, thus preserving the causal structure and avoiding lookahead bias.

The loop iterates from the end of the initial training window to the penultimate observation, producing a sequence of true out-of-sample predictions that can be used for backtesting. Without this structure, only a single forecast would be available for $Y_{(t+1)}$, limiting the model's evaluability.

Modelling for Volatility

The approach models next-day volatility using a multi-stage process. Predicted future price trajectories in the previous section are used as input features for volatility estimation.

The framework integrates Hybrid Bidirectional Stacked LSTMs (xLSTMs), Temporal Fusion Transformers (TFTs), and a Meta Learners (MLs) to predict next-day volatility.

Input Features: Previously predicted Prices

Generated by previous predictive models, serving as inputs for volatility modeling:

- $\hat{S}_{(t+1,close,XLSTM)}, \hat{S}_{(t+1,high,XLSTM)}, \hat{S}_{(t+1,low,XLSTM)}$ → Price forecast from xLSTM.
- $\hat{S}_{(t+1,close,TFT)}, \hat{S}_{(t+1,high,TFT)}, \hat{S}_{(t+1,low,TFT)}$ → Price forecast from Temporal Fusion Transformer.
- $\hat{S}_{(t+1,close,ML)}, \hat{S}_{(t+1,high,ML)}, \hat{S}_{(t+1,low,ML)}$ → Price forecast from Meta Learner.
- $\hat{S}_{(t+1,close,XGB)}, \hat{S}_{(t+1,high,XGB)}, \hat{S}_{(t+1,low,XGB)}$ → Probability forecast from XGBoost.

Input Features: Observed data

- $S_{(t,open)}, S_{(t,low)}, S_{(t,high)}, S_{(t,low)}, S_{(t+1,open\, refference)}$ → Observed prices at t.
- $\sigma_{(t,at\ open)}, \sigma_{(t,at\ close)}, \sigma_{(t,max)}, \sigma_{(t,min)}, \sigma_{(t,mean)}, \sigma_{(t+1,open\ reference)}$ → Observed volatility measures at t.

Process

Since all model-predicted variables (except the observed ones) are arithmetic returns to open price, consistency in scale and information representation is necessary.

Observed variables contain absolute price levels and volatility measures. These require transformation to extract meaningful return-based features.

Absolute prices $S_{(t,k)}$ are transformed into log-returns for better distributional properties for volatility modeling:

$$r_{(t,open-close)} = \log\left(\frac{S_{(t,close)}}{S_{(t,open)}}\right)$$

$$r_{(t,open-high)} = \log\left(\frac{S_{(t,high)}}{S_{(t,open)}}\right)$$

$$r_{(t,open-low)} = \log\left(\frac{S_{(t,low)}}{S_{(t,open)}}\right)$$

- Logarithmic returns stabilizes variances over time, reduces heteroskedasticity and aligns statistical properties of the time series.

Since raw volatilities ($\sigma_{(t,at\ open)}, \sigma_{(t,at\ close)}, \sigma_{(t,max)}, \sigma_{(t,min)}, \sigma_{(t,mean)}, \sigma_{(t+1,open\ reference)}$) can be non-stationary, relative volatility scaling is applied:

$$r\sigma_{(t,at\ open)} = \frac{\sigma_{(t,at\ open)}}{S_{(t,open)}}$$

$$r\sigma_{(t,at\ close)} = \frac{\sigma_{(t,at\ close)}}{S_{(t,close)}}$$

$$r\sigma_{(t,max)} = \frac{\sigma_{(t,max)}}{S_{(t,high)}}$$

$$r\sigma_{(t,min)} = \frac{\sigma_{(t,min)}}{S_{(t,low)}}$$

$$r\sigma_{(t,mean)} = \frac{\sigma_{(t,mean)}}{\frac{S_{(t,high)}+S_{(t,low)}}{2}}$$

$$r\sigma_{(t+1,open\ reference)} = \frac{\sigma_{(t+1,open\ reference)}}{S_{(t+1,open\ reference)}}$$

- Relative volatility scaling ensures comparability at different price levels, reduces scale dependence, and captures relative price dispersion rather than absolute magnitude.

The Augmented Dickey-Fuller Test for stationarity, INFNA test for NaNs and infinities, and Mutual Information Elimination Test (detailed in the *Data Analytics & Feature Selection* section), are applied to each input feature.

Subsequently, the architectures of the Hybrid Bidirectional Stacked LSTM, Temporal Fusion Transformers (TFT), and Meta Learner (ML) as detailed in the preceding section, will be adapted for the volatility modelling.

For LSTM and TFT, the primary modifications will be in the input features and target variables, which now focus on forecasting the future maximum volatility, future minimum volatility, and future volatility at the close:

$$\hat{\sigma}_{(t+1,at\ close)}, \hat{\sigma}_{(t+1,max)}, \hat{\sigma}_{(t+1,min)}$$

To achieve this, three Hybrid Bidirectional Stacked LSTM models will be tailored to predict these volatility measures at the respective price levels. Similarly, three TFT models will be adapted for the same objectives. This results in the following outputs from the models:

- $\left(\hat{\sigma}_{(t+1,\text{at }close,XLSTM)}, \hat{\sigma}_{(t+1,max,XLSTM)}, \hat{\sigma}_{(t+1,min,XLSTM)}\right)$
- $\left(\hat{\sigma}_{(t+1,\text{at }close,TTF)}, \hat{\sigma}_{(t+1,max,TTF)}, \hat{\sigma}_{(t+1,min,TTF)}\right)$

Additionally, the Meta Learner receives the output volatilities of for the next day from the models as input features to predict these volatilities, resulting in the following outputs:

- $\hat{\sigma}_{(t+1,\text{at }close,ML)}, \hat{\sigma}_{(t+1,max,ML)}, \hat{\sigma}_{(t+1,min,ML)}$

Conclusion

The modeling framework operationalizes a multi-tiered forecasting system, wherein distinct neural and tree-based architectures are embedded within an expanding window protocol to generate daily-updated forecasts for both stock price movement and volatility.

Each base learner—*xLSTM*, *TFT*, and *XGBoost*—is deliberately overfitted to the most recent available data, leveraging complete historical records without partitioning for validation.

The absence of a test set is counterbalanced by the implementation of a time-indexed retraining loop, which ensures that only contemporaneously observable features inform predictions of the subsequent time step, $Y_{(t+1)}$. This structural design allows for causal fidelity and prevents information leakage.

The "time machine" mechanism, instantiated through an expanding window loop, constructs a synthetic temporal evaluation environment by simulating the model's historical deployment.

At each iteration, models are retrained with an additional observation and tasked with predicting the out-of-sample target for the next period, emulating a real-time pipeline.

This approach converts a static dataset into a dynamic sequence of expanding training and prediction events, enabling rigorous out-of-sample evaluation and high-frequency backtesting.

In the absence of a predefined test set, this iterative retraining structure becomes a necessary surrogate for validating short-term predictive accuracy.

Strategy

Strategic design remains undefined due to the absence of domain-specific execution expertise in high-frequency options trading. To address this, collaboration is being initiated with Dr. Wang, a recognized expert in algorithmic derivatives trading and a faculty member at the University of South Florida.

His research portfolio includes signal processing under latency constraints, delta-neutral hedging under transaction cost frictions, and stochastic control under market microstructure noise. His involvement would integrate a domain-specific decision-theoretic layer into the pipeline, optimizing model-generated signals for real-time deployment under dynamic conditions.

The objective of this collaboration is to inject expert priors into the strategy design layer, enabling efficient mapping from predictive outputs to executable trades. This includes selecting appropriate execution templates, contract filtering heuristics (e.g., delta and gamma thresholds), risk constraints (e.g., maximum notional exposure, volatility-adjusted position sizing), and temporal signal decay models.

Dr. Wang's expertise will also be leveraged for the derivation of utility-optimizing policy functions, ensuring that the strategy is optimal.

Backtesting

The backtesting framework integrates the predictive outputs of all proposed architectures with the execution logic of a trading strategy designed for American-style out-of-the-money (OTM) options.

The simulation environment replicates real-market dynamics by leveraging historical option chain data and precise market microstructure granularity. The simulation is executed via QuantConnect’s research environment, which incorporates full archival datasets, order book states, transaction fees, slippage models, tax implications, and liquidity constraints.

This infrastructure facilitates forward-testing under conditions that mirror live trading with near-tick-level fidelity.

The system ingests model-generated forecasts—whether directional, continuous, or probabilistic—and transforms them into actionable trading signals. Each signal triggers an order placement logic, which is executed under the constraints of market timing, available capital, and regulatory limits.

The strategy evaluates both long and short OTM options, selecting contracts based on moneyness thresholds, time-to-expiry filters, delta bounds, and implied volatility ranks.

Trade decisions are conditioned on an ensemble consensus derived from the outputs. The simulation employs a rolling window backtesting mechanism with temporal causality constraints. At each iteration t, the model is retrained using all information up to time t, and a forecast is generated for$t + 1$.

The simulation environment then executes the strategy using this forecast, logs the result, updates internal states, and iterates forward. This time-stepping framework prevents data leakage and lookahead bias, enforcing strict separation between training and evaluation windows.

Two primary performance metrics serve as the pillars of evaluation: binary efficiency per operation and average profit/loss per operation. Binary efficiency assigns a value of 1 if the net PnL of a transaction (post-costs, slippage, tax adjustments) is positive, and 0 otherwise. This metric quantifies the directional correctness of the decision-making pipeline.

However, directional accuracy alone is insufficient. High binary efficiency with marginal gains and infrequent but large losses can yield a net-negative return profile.

Therefore, the average profit/loss per operation is computed to capture the monetary magnitude of outcomes. This metric evaluates mean returns across all trades and acts as a stochastic expectation estimator for future deployments.

Additional performance indicators are integrated to support multi-dimensional diagnostics. Some of them are the following:

- Cumulative Return: Aggregates all realized PnLs over the backtesting horizon.
- Average ROI per Trade: Measures relative return against margin requirements or capital exposure per trade.
- Cumulative ROI: Provides normalized total return over the capital base.
- Max Drawdown: Quantifies peak-to-trough equity declines, capturing tail-risk exposure.
- Sortino Ratio: Assesses risk-adjusted return penalizing only downside volatility.
- Sharpe Ratio: Computes return-to-volatility tradeoff across all trades.
- Volatility of PnL: Captures standard deviation of outcomes to evaluate consistency.
- Skewness and Kurtosis of PnL: Measures asymmetrical and tail-fatness to detect deviation from Gaussian return profiles.
- Hit Rate by Contract Maturity and Delta: Segments binary efficiency by options characteristics to uncover behavioral micro-patterns.
- PnL Attribution by Model Source: Decomposes returns based on which model (LSTM, TFT, Meta Learner) originated the signal.
- Autocorrelation of PnL Series: Quantifies serial dependency in trade outcomes to detect regime persistence or mean-reverting patterns.
- Hurst Exponent of Equity Curve: Measures long-term memory in the cumulative return series, indicating trending vs. mean-reverting dynamics.
- Rolling Beta to Market Index: Estimates dynamic systematic exposure using time-varying regression against benchmark indices.
- PnL Cross-Correlation Matrix Across Models: Evaluates temporal dependency and redundancy between predictive sources.
- Time-to-Mean-Reversion After Loss: Quantifies average number of trades or time units required to recover from drawdowns.
- Conditional Value at Risk (CVaR): Captures expected tail loss conditional on breaching a specified quantile of the return distribution.
- Z-score of Trade PnLs: Normalizes trade outcomes to assess extremity relative to empirical distribution.
- Jensen’s Alpha: Computes strategy-specific alpha generation net of market risk, based on CAPM assumptions.
- Rolling Information Ratio: Evaluates consistency of excess returns over a benchmark across sliding windows.
- Trade Duration Distribution Analysis: Models entry-to-exit holding periods and their influence on profitability via kernel density estimation.
- Execution Delay Sensitivity: Measures PnL erosion due to simulated latency and delayed fills.
- Entropy of Trade Outcome Distribution: Measures the uncertainty and information content in the distribution of trade results, indicating system predictability.

- Kullback–Leibler Divergence from Normality: Quantifies how much the empirical PnL distribution deviates from a Gaussian reference, detecting structural asymmetries.
- Anderson-Darling Test Statistic on PnL: Assesses the goodness-of-fit of trade outcomes against theoretical distributions to validate modeling assumptions.
- Stationarity Test (ADF) on Cumulative Returns: Evaluates the presence of unit roots in the equity curve, essential for assessing trend sustainability and reversion dynamics.
- Cross-Validated Predictive Log-Likelihood: Aggregates log-likelihoods across temporal folds to measure the statistical fit and generalization capacity of the model.

Calibration diagnostics compare predicted probabilities (from classification-based XGBoost outputs) to empirical frequencies using Brier scores, calibration curves, and expected calibration errors (ECE). Under-calibrated or overconfident outputs signal the need for temperature scaling or Platt calibration post-training.

Feature ablation studies remove subsets of inputs (e.g., volatility predictors, macroeconomic indicators, technical signals) to quantify marginal utility. This process iteratively estimates Shapley values, offering model-agnostic feature importance under causal inference constraints.

Model selection criteria incorporate Akaike Information Criterion (AIC), Bayesian Information Criterion (BIC), and Minimum Description Length (MDL) under complexity-penalized scoring functions. These criteria serve to compare model variants within the same architecture family (e.g., shallow LSTM vs. stacked LSTM) without relying on out-of-sample validation.

The ensemble strategy is evaluated under adversarial scenarios: *(1) low liquidity shocks*, *(2) volatility clustering anomalies*, *(3) unannounced earnings events*, and *(4) structural breaks (e.g., COVID-type regime shifts)*. Stress-testing under these conditions evaluates survivability rather than profitability.

Execution diagnostics include fill rate statistics, partial fill analysis, slippage drift tracking, and bid-ask spread sensitivity. The simulation logs on whether market orders execute within the specified temporal horizon and under what market depth conditions.

Model synchronization frequency is benchmarked. Recalibration intervals (daily) are cross validated against predictive decay curves to determine the optimal retraining cadence, balancing signal freshness against computational overhead.

Profitability is further decomposed by:

- Option type (Call vs. Put)
- Strike distance from spot
- Implied volatility percentile bucket
- Holding time post-entry
- Forecast confidence decile

Stop-loss and take-profit boundaries are simulated via synthetic price trajectories reconstructed from intra-day granularity. This adds realism to the position of lifecycle beyond simple open-close deltas.

Market impact simulation is conducted using Kyle's lambda approximation and volume-weighted depth-of-book data, modeling slippage as a function of aggressiveness and notional size.

Transaction cost modeling includes SEC fees, exchange levies, broker-specific commissions, and SEC Rule 605/606-based fill quality impact estimates. These are parameterized via empirical cost curves calibrated to historic execution logs.

Decision attribution maps each prediction to its originating model, input configuration, and hyperparameter state, enabling root-cause diagnosis during post-trade analysis. Each trade record is stored as a high-dimensional vector in an indexed analytical database.

Threshold optimization is implemented via ROC-AUC maximization, profit-maximizing threshold sweep (PMTS), and differentiable F-beta optimization to identify the trade-off frontier between precision and recall in signal execution.

Together, this comprehensive framework not only enables high-fidelity simulation of OTM options trading under real-world constraints but also acts as a rigorous validation pipeline.

Each component is stress-tested, calibrated, and decomposed to allow performance auditing at every architectural and strategic layer. This ensures the transition from research to live deployment is not only profitable but robust under adversarial market conditions and operational constraints.

Real-Time Deployment

Real-time deployment initiates with a snapshot request triggered precisely at 9:27 AM Eastern Time via Bloomberg's Python API. This request retrieves a dense matrix of market features and instrument metrics, returning a structured JSON response optimized for high-frequency ingestion.

Bloomberg's server-side delivery infrastructure ensures that the time from request to receipt remains under one second. Upon arrival, the data is transmitted over a secure HTTPS channel to a high-performance AWS EC2 instance, outfitted with an NVIDIA A100 GPU and hosted in the US-East region.

This instance has been specifically optimized for ultra-low latency workloads, with all internal containers governed by lightweight orchestration tools that auto-restart on fault and guarantee memory consistency across parallel workers.

Within this inference environment, the incoming raw data is immediately passed through a real-time filtration module built in accordance with the *Data Analytics & Feature Selection* section.

This pipeline consistently distills the input down to 15 features dynamically selected according to the *Arattone Theorem for High Dimensionality*, for their predictive stability that day. These features are then propagated into the model hierarchy using a shared memory system, enabling parallel dispatch without incurring IO bottlenecks or serialization costs.

The modeling stack follows a strict hierarchy but is parallelized wherever architectural independence permits. Hybrid Bidirectional Stacked LSTM and Temporal Fusion Transformer (TFT) models are the first to activate, generating open-relative return forecasts for high, low, and close prices.

Their outputs are subsequently processed by an XGBoost-based Meta Learner, which consolidates the diverse architectural signals into a refined continuous-value forecast. A final XGBoost classifier—distinct from the regression-based Meta Learner—then aggregates these outputs with the original feature set to produce a probability estimate of directional return.

Following price modeling, a new inference cycle is initiated for volatility forecasting. Here, the original filtered features are combined with the full output layer of the price prediction ensemble.

Separate LSTM and TFT models are then run to forecast three volatility measures: close-to-close, maximum, and minimum. These outputs are subsequently integrated by a volatility-specific Meta Learner, trained exclusively on volatility outputs to preserve abstraction and avoid data leakage from the price prediction task.

All forecasts—nineteen in total—are combined into a unified signal tensor, which is then transformed into a JSON payload through a custom schema conversion utility. This converter ensures full compatibility with QuantConnect's Lean ingestion protocol, accounting for key-value mapping, type strictness, and structural expectations, while also providing segmentation secrecy.

The resulting package is published to a secure endpoint monitored continuously by QuantConnect's Lean engine.

Inside Lean, a persistent polling mechanism checks the signal endpoint every 500 milliseconds. Once a new signal is detected, order preparation begins immediately. This includes risk-layer validation, compliance checks, and dynamic order-type selection based on signal strength and market state.

Trades are routed through QuantConnect's native brokerage integrations, with Charles Schwab selected as the execution partner for this implementation.

The integration leverages Schwab's direct API connectivity through Lean, enabling the submission of complex order types with millisecond precision.

Execution receipts, including partial fills and latency feedback, are streamed back to the EC2 instance in real time and stored in an indexed transaction ledger for reconciliation and auditability.

Orchestration of this end-to-end system is designed around a hard three-minute SLA. Every component—from data acquisition and feature selection to model inference, payload formatting, and trade execution—is clocked independently and benchmarked for performance.

To maintain this SLA, compute tasks are segmented across multiple cores and synchronized using a multiprocessing event-driven framework. Data movement occurs through in-memory pipelines rather than disk IO, and fault-tolerant retry logic is built into each service node to prevent cascading delays.

The final system architecture thus forms a tightly coupled, latency-optimized decision pipeline. It unifies diverse subsystems—data vendors, inference hardware, model abstraction, and brokerage APIs—into a single computational moment. This moment begins at 9:27 AM with a snapshot of the world and concludes, within three minutes, with a live market order engineered for maximal predictive power, computational efficiency, and operational robustness.

Next Steps

The forthcoming phases will focus on the development of a robust strategic framework in collaboration with Dr. Wang to optimize the utility of algorithmic resources.

This initiative will encompass the refinement of quantitative methodologies to ensure maximal efficiency in data-driven decision-making. Advanced derivatives techniques will be leveraged to enhance predictive accuracy, risk assessment, and market adaptability.

In parallel, a comprehensive approach will be established alongside Dr. DiGiovanni for portfolio management and financial modeling.

This will involve the integration of sophisticated asset allocation models, leveraging stochastic processes and optimization algorithms to enhance risk-adjusted returns. The objective is to construct an adaptive framework capable of dynamically responding to market fluctuations and macroeconomic shifts.

The implementation of this initiative will necessitate access to Bloomberg terminals equipped with API functionality. These terminals will serve as a vital conduit for real-time data acquisition, facilitating the seamless integration of financial datasets into predictive models.

The API capabilities will enable automated data retrieval and processing, ensuring that all analytical outputs are based on the most up-to-date market conditions.

Moreover, high-performance computing (HPC) infrastructure will be indispensable for handling vast volumes of financial data and executing complex algorithms with minimal latency.

The deployment of parallel computing architectures and GPU acceleration will significantly reduce computational time, allowing for the efficient backtesting and validation of intricate financial models.

The synergy between algorithmic development and financial modeling will be reinforced through interdisciplinary collaboration. The convergence of expertise in machine and deep learning, statistical modeling, and quantitative finance will yield a comprehensive framework that aligns theoretical advancements with practical implementation.

Rigorous validation procedures will be employed to ensure the robustness and reliability of all derived insights.

A structured evaluation mechanism will be implemented to assess the efficacy of the proposed methodologies. Key performance indicators (KPIs) will be established to benchmark algorithmic performance, predictive accuracy, and portfolio optimization effectiveness.

This will enable continuous refinement of the strategic approach and facilitate iterative enhancements based on empirical findings.

The initiative will also emphasize the importance of risk management through the application of advanced econometric techniques and stochastic modeling.

Tail risk, volatility clustering, and regime-switching behaviors will be incorporated into the modeling process to ensure a comprehensive understanding of potential downside risks and market contingencies.

Furthermore, scenario analysis and stress testing methodologies will be integrated into the portfolio management framework. These techniques will enable proactive identification of vulnerabilities under extreme market conditions, thereby strengthening risk mitigation strategies and enhancing capital preservation measures.

Given the complexity of the proposed framework, the availability of institutional resources from USF, including access to expert counsel and cutting-edge technological infrastructure, will be instrumental in ensuring successful implementation.

The alignment of these resources with the strategic objectives will facilitate the seamless execution of all model development and portfolio management activities.

Ultimately, the synthesis of high-caliber computational resources, advanced financial modeling, and expert collaboration will establish a pioneering approach to algorithmic trading and portfolio optimization.

The integration of real-time data, rigorous analytical methodologies, and institutional expertise will ensure that the proposed framework delivers sustained financial performance and strategic resilience in dynamic market environments.

Conclusion

This study presents a comprehensive and rigorously engineered framework for real-time intraday trading of American out-of-the-money (OTM) options, grounded in an ensemble of nineteen predictive models and unified through a strategic orchestration system that emphasizes adaptability, precision, and theoretical innovation.

By integrating models that span deep learning (hybrid xLSTM and vanilla Temporal Fusion Transformers), machine learning (Meta Learner using XGBoost Regression), and probabilistic classification (XGBoost Classifier), the framework creates a multidimensional predictive landscape that captures both directional price movements and the stochastic nature of volatility, offering traders a set of insights into market microstructure dynamics.

What sets this approach apart is not simply the layering of diverse model architectures but the philosophical and methodological reconfiguration of how financial forecasting can be performed in volatile, high-frequency environments.

The focus is not solely on the price action of the underlying asset, but on its evolving volatility signature—a component often ignored or superficially modeled in traditional systems.

The bifurcation of forecasting into stock return prediction and volatility estimation allows for a dual-axis decision mechanism that recognizes the interdependence of expected returns and risk metrics in the valuation of options.

This is particularly significant in the context of OTM options, where implied volatility changes can exert disproportionate influence on pricing.

Central to the efficacy of this architecture is the Expanding Window retraining methodology. Traditional financial forecasting models suffer from obsolescence and concept drift, their parameters frozen within the temporal limits of a static training set.

The Expanding Window approach counters this by recalibrating all models daily with every new market observation. This ensures that the ensemble remains contextually relevant, capable of absorbing new market conditions and reflecting shifts in investor behavior and volatility regimes.

Crucially, the Expanding Window does not rely on a conventional testing set; rather, it derives validation intrinsically through its forward-moving recalibration logic.

This daily retraining paradigm transforms the classic limitations of overfitting into an advantage, allowing models to mirror contemporary market nuances and achieve sharper forecast granularity.

The three novel theorems proposed in this study—*Expanding Window*, *Sae-Mayo for Observational Efficiency*, and *Arattone for High Dimensionality*—serve as the foundational scaffolding for the system's design and operation.

The Expanding Window Theorem provides the epistemological basis for dynamic model updating, ensuring continuous alignment with the market's informational state.

The Sae-Mayo Theorem empirically establishes a threshold of 5,000 observations as the optimal training size for capturing stable predictive behavior without incurring diminishing returns.

The Arattone Theorem, meanwhile, formalizes the non-linear relationship between model dimensionality and forecast precision, ensuring that the feature space remains expressive but not overburdened by redundancy.

Together, these theorems enable a rare combination of theoretical clarity and practical efficiency, empowering the ensemble to operate with both agility and robustness.

Equally significant is the framework's attention to data preprocessing and feature integrity. Unlike many systems that treat feature engineering as a peripheral concern, this framework places data quality, interpretability, and statistical soundness at the forefront.

Through rigorous INFNA protocols, the dataset is purged of non-interpretable values (NaNs, ∞, $-\infty$), and these transformations are tested via distributional and trend validation to ensure that imputation does not distort the underlying statistical structure.

Outliers are deliberately retained rather than suppressed, honoring the heavy-tailed, non-Gaussian nature of financial return distributions. In an environment where rare events often drive regime shifts, preserving these anomalies becomes essential for realistic modeling.

Further, this system goes beyond univariate variable testing and adopts a sophisticated mutual information-based elimination protocol to account for multicollinearity.

By computing mutual information scores not only between each feature and the target variable, but also between each pair of features, the system ensures that only uniquely informative variables are retained. This double-entropy filter prevents both redundancy and spurious correlation, ensuring that each input contributes novel, actionable signal.

Importantly, this process also integrates stationarity testing using the Augmented Dickey-Fuller test, discarding variables that display persistent unit roots and thus resist temporal modeling.

In terms of operational execution, the system achieves real-time implementation under strict latency constraints. Each trading day at exactly 9:27 AM Eastern Time, a full market snapshot is retrieved via Bloomberg and processed through a cloud-based inference node.

Within three minutes, this node engages the entire ensemble—deep learning pipelines, XGBoost regressors, and classification layers—and outputs a structured payload of predictions. These signals are routed through a customized API to QuantConnect and executed via a direct brokerage integration with Charles Schwab.

The entire cycle is monitored via a real-time feedback loop that measures performance, latency, and fill quality, ensuring continuous coherence across forecasting, execution, and response.

The framework’s separation of signal generation from option-specific pricing complexities further enhances its modularity. By isolating the prediction of the underlying stock's return and volatility characteristics and incorporating these forecasts into a derivative-aware execution layer, the system maintains conceptual clarity while retaining operational flexibility.

This allows for broad generalization across different underlying’s, strike prices, or expiration timelines without requiring retraining at the option level.

From a systems design perspective, this framework offers not just a model, but a full-stack intelligent infrastructure. Its components are tightly integrated yet independently functional, and its design philosophy marries statistical rigor with engineering scalability.

The predictive ensemble, feature pipeline, preprocessing architecture, and execution system form a self-correcting ecosystem capable of continuous learning and adaptive signal generation.

In conclusion, this paper contributes a scalable, theoretically grounded, and empirically validated framework for intraday options trading. It provides a clear departure from static predictive architectures and simplistic signal generation strategies, introducing instead a real-time adaptive system capable of responding to the complexities of modern markets.

By harmonizing deep learning, ensemble machine learning, rigorous data handling, and precise execution mechanics, the framework exemplifies a new perspective of financial modeling—one that is as theoretically rigorous as it is operationally effective.

It offers proof of concept for systems that can evolve with the markets they aim to understand, and a blueprint for building architectures that do not merely forecast the future—they grow with it.