Stock Price Correlation Coefficient Prediction with ARIMA-LSTM Hybrid Model

Hyeong Kyu Choi*

replicated by Shashank Dubey †

February 11, 2026

Abstract

Our replication study presents a hybrid ARIMA-LSTM model designed to predict correlations between stock prices, which are essential for portfolio management and risk assessment. The methodology begins with collecting fully imputed daily price data for 150 SP500 stocks, calculating correlation coefficients for each pair over a 100-day rolling window. Five different starting points for these windows are chosen to add diversity, resulting in $55,875$ time series datasets, each with 24 time steps. The data is then split into training, development, and test sets using a walk-forward optimization technique, allowing rigorous out-of-sample testing. The ARIMA model captures linear trends, while LSTM addresses non-linear patterns, with residuals from ARIMA fed into the LSTM. This hybrid approach outperforms traditional models, showcasing its robustness in forecasting stock correlations and providing a promising tool for financial applications in managing correlated assets.

*Department of Business Administration, Korea University

†Department of Management Studies, Indian Institute of Technology, Delhi

Introduction

The cornerstone of modern portfolio theory, the Markowitz model, relies on accurate estimates of asset returns and correlations to optimize investment portfolios. However, the assumption of constant correlation coefficients over time has been widely criticized. Empirical studies have shown that correlations tend to fluctuate, particularly during periods of market stress.

Various statistical and financial models have been proposed to forecast correlation coefficients to address this limitation. While these models have achieved some success, there is growing interest in leveraging the power of machine learning, specifically neural networks, for this task. With their ability to learn complex patterns, neural networks offer the potential to capture the dynamic nature of correlation coefficients more effectively. We replicate a hybrid model proposed by Choi (2018) that combines the strengths of ARIMA and LSTM models for correlation coefficient prediction. By combining ARIMA's linear modeling capabilities with LSTM's nonlinear pattern recognition, this hybrid approach aims to provide more accurate and robust forecasts.

This replication study contributes to the understanding of the proposed model and explores its potential applications in practical portfolio management. By examining the performance of the model on real-world data, we can assess its effectiveness in capturing the time-varying nature of correlation coefficients and its implications for portfolio optimization.

Current Correlation Models

Full Historical Model

The Full Historical model is the simplest method to implement for portfolio correlation estimation. This model adopts the past correlation value to forecast the future correlation coefficient. That is, the correlation of two assets for a certain future period is expected to be equal to the correlation value of a given past period.

$$\hat{\rho}_{ij}^{(t)} = \rho_{ij}^{(t-1)} \tag{1}$$

i, j: asset index in the correlation coefficient matrix

However, this model has encountered criticisms for its relatively inferior prediction quality compared to other equivalent models.

Constant Correlation Model

The Constant Correlation model assumes that the full historical model encompasses only the mean information of the mean correlation coefficient. Any deviation from the mean correlation coefficient of each pair is considered a random noise; it is sufficient to estimate the correlation of the mean correlation of all assets in a given portfolio. Therefore, applying the Constant Correlation of all pairs of assets in a single portfolio has the same correlation coefficient.

$$\hat{\rho}_{ij}^{(t)} = \frac{\sum_{i,j} \rho_{ij}^{(t-1)}}{n(n-1)/2} \tag{2}$$

i, j: asset index in the correlation coefficient matrix
n: number of assets in the portfolio

Single-Index Model

The Single-Index model presumes that asset returns move systematically with market return. To quantify the systematic movement to the market return, we need to specify the market return itself. The 'market model' relates the return of asset i with the market return at time t, represented by the equation:

$$R_{i,t} = \alpha_i + \beta_i R_{m,t} + \epsilon_{i,t} \tag{3}$$

$R_{i,t}$: return of asset i at time t
$R_{m,t}$: return of the market at time t
α_i: risk-adjusted excess return of asset i
β_i: sensitivity of asset i to the market
$\epsilon_{i,t}$: residual return; error term such that, $E(\epsilon_{i,t}) = 0$; $\mathrm{Var}(\epsilon_{i,t}) = \sigma_i^2$

Here, we use the sensitivity (β) of asset i and j to estimate the correlation coefficient.

$$\mathrm{Cov}(R_i, R_j) = \rho_{ij}\sigma_i\sigma_j = \beta_i\beta_j\sigma_m^2 \tag{4}$$

σ_i/σ_j: standard deviation of asset i's / j's return
σ_m: standard deviation of market return

The estimated correlation coefficient $\hat{\rho}_{ij}$ would be,

$$\hat{\rho}_{ij}^{(t)} = \frac{\beta_i\beta_j\sigma_m^2}{\sigma_i\sigma_j} \tag{5}$$

Multi-Group Model

The Multi-Group model takes the asset's industry sector into account. Under the assumption that assets in the same industry sector generally perform similarly, the model sets each correlation coefficient of asset pairs identical to the mean correlation of the industry sector pair's correlation value.
In other words, the Multi-Group model applies the Constant Correlation model to each pair of business sectors. For example, if company A and company B, each belonging to industry sectors α and β, their correlation coefficient would be the mean value of all the correlation coefficients of pairs of assets with the same combination of industry sectors (α, β).
The equation for the prediction is slightly different depending on whether the two industry sectors α and β are identical or not. The equation is as follows.

$$\hat{\rho}_{ij}^{(t)} = \begin{cases} \frac{1}{n_\alpha n_\beta} \sum_{i \in \alpha} \sum_{j \in \beta} \rho_{ij}^{(t-1)}, & \text{where } \alpha = \beta \\ \frac{1}{n_\alpha n_\beta} \sum_{i \in \alpha} \sum_{j \in \beta} \rho_{ij}^{(t-1)}, & \text{where } \alpha \neq \beta \end{cases} \tag{6}$$

α and β: industry sector notation
n_α / n_β: the number of assets in each industry sector

ARIMA

Autoregressive Integrated Moving Average (ARIMA) is a powerful statistical model for time series forecasting. They are beneficial when dealing with non-stationary time series data. An ARIMA model is characterized by three parameters: p, d, and q.

- p: The order of the autoregressive part, representing the number of lag observations included in the model.
- d: The degree of differencing, indicating the number of times the raw observations are differenced to achieve stationarity.
- q: The order of the moving average part, representing the size of the moving average window.

Model Equation

The general equation for an ARIMA(p,d,q) model is:

$$X_t = c + \phi_1 X_{t-1} + ... + \phi_p X_{t-p} + \theta_1 \epsilon_{t-1} + ... + \theta_q \epsilon_{t-q} + \epsilon_t \tag{7}$$

X_t: The value of the time series at time t.
c: A constant term.
$\phi_1, ..., \phi_p$: Autoregressive coefficients.
$\theta_1, ..., \theta_q$: Moving average coefficients.
ϵ_t: White noise error term.

Model Process

Stationarity: If the time series is not stationary, differencing is applied to make it stationary.

Model Identification: The appropriate values of p, d, and q are determined using techniques like the Autocorrelation Function (ACF) and Partial Autocorrelation Function (PACF).

Model Estimation: The model parameters are estimated using methods like maximum likelihood estimation.

Model Diagnostics: The model's residuals are checked for autocorrelation and normality to assess its fit.

Forecasting: Once the model is fitted, it can generate forecasts for future periods.

Long Short-Term Memory (LSTM) Networks

Long Short-Term Memory (LSTM) networks are a type of Recurrent Neural Network (RNN) specifically designed to address the vanishing gradient problem, which hinders the ability of traditional RNNs to learn long-term dependencies in sequential data.

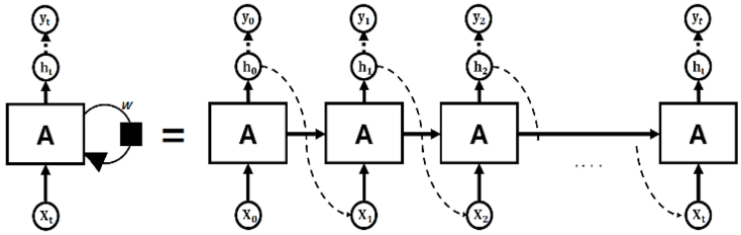

Figure 1: Structure of RNNs

Core Components of LSTM

An LSTM unit consists of four main components:

- **Cell State:** This is the long-term memory of the LSTM, which can store information over many time steps.
- **Input Gate:** This gate controls the flow of new information into the cell state.
- **Forget Gate:** This gate determines which information should be removed from the cell state.
- **Output Gate:** This gate decides which information from the cell state should be output.

Mathematical Formulation

Let h_t be the hidden state at time step t, c_t be the cell state at time step t, x_t be the input at time step t, and W and b be the weight and bias matrices, respectively. The core equations of an LSTM can be expressed as follows:

- **Forget Gate:**

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

- **Input Gate:**

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

- **Cell State Update:**

$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$
$$c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$$

- **Output Gate:**

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

- **Hidden State:**

$$h_t = o_t * \tanh(c_t)$$

σ: Sigmoid activation function.
tanh: Hyperbolic tangent activation function.

Applications in Financial Forecasting

LSTMs have shown promising results in various financial forecasting tasks, including:

- Stock Price Prediction: By analyzing historical stock price data, LSTMs can predict future price movements.
- Forex Trading: LSTMs can be used to predict exchange rate fluctuations.
- Portfolio Management: LSTMs can help in asset allocation and risk management by forecasting asset returns and correlations.
- Options Pricing: LSTMs can be employed to model the underlying asset's price dynamics and estimate option prices.

LSTMs excel in financial forecasting due to their ability to capture complex patterns, handle long-term dependencies, and adapt to changing market conditions. By leveraging the power of deep learning, LSTMs offer a valuable tool for financial analysts and investors.

Method and Data

Data Selection and Preprocessing

The data set provided by the author was utilized. The data set comprised adjusted closing prices of SP 500 stocks from 2008 to 2017. The data set was fully imputed using the last available price to fill in missing values. From this processed dataset, 150 stocks were randomly selected and their correlation coefficients were calculated over a rolling 100-day window, yielding time series data with 24-time steps per pair of stocks. This process produced a 55,875 correlation coefficient time series, each comprising five data sets for training, development, and testing purposes.

ARIMA Model

ARIMA is first applied to capture the linear components of the time-series data. While the paper tested for five different ARIMA configurations, I tested for three i.e., (1,1,0), (0,1,1), and (1,1,1). The model configuration with the lowest Akaike Information Criterion (AIC) score was selected for each dataset. The ARIMA model's residuals, which contain the non-linear aspects of the data, were passed to the LSTM model for further processing.

LSTM Model

The LSTM model—known for handling sequential data—processes the ARIMA residuals to account for non-linear trends in stock correlations. The model architecture consists of 10 LSTM units, with the final prediction layer activated by a scaled hyperbolic tangent function (double-tanh) to produce outputs in the range needed for correlation values. Overfitting is minimized using dropout, which randomly disables certain neurons during training, forcing the network to rely on different paths and improving generalization.

Evaluation Methodology

A walk-forward optimization approach is employed to assess model performance across rolling time intervals. This method requires training a new model for each interval and testing it on the next to ensure robustness. However, to reduce the computational cost, the study trained one model with the initial training window and applied it across subsequent test intervals. Evaluation metrics include Mean Squared Error (MSE), Mean Absolute Error (MAE), and Root Mean Squared Error (RMSE) for model comparison.

Comparison with Benchmark Models

The ARIMA-LSTM model's performance is contrasted with traditional models such as the Full Historical Model, Constant Correlation Model, Single Index Model and Multigroup Model. The hybrid model consistently outperforms these benchmarks, particularly the Constant Correlation model, suggesting significant predictive gains.

The hybrid ARIMA-LSTM model demonstrates superior predictive accuracy, validating its potential for use in portfolio optimization tasks. Choi concludes that incorporating both linear and nonlinear elements offers meaningful improvements in stock correlation prediction and could enhance risk management in financial portfolios.

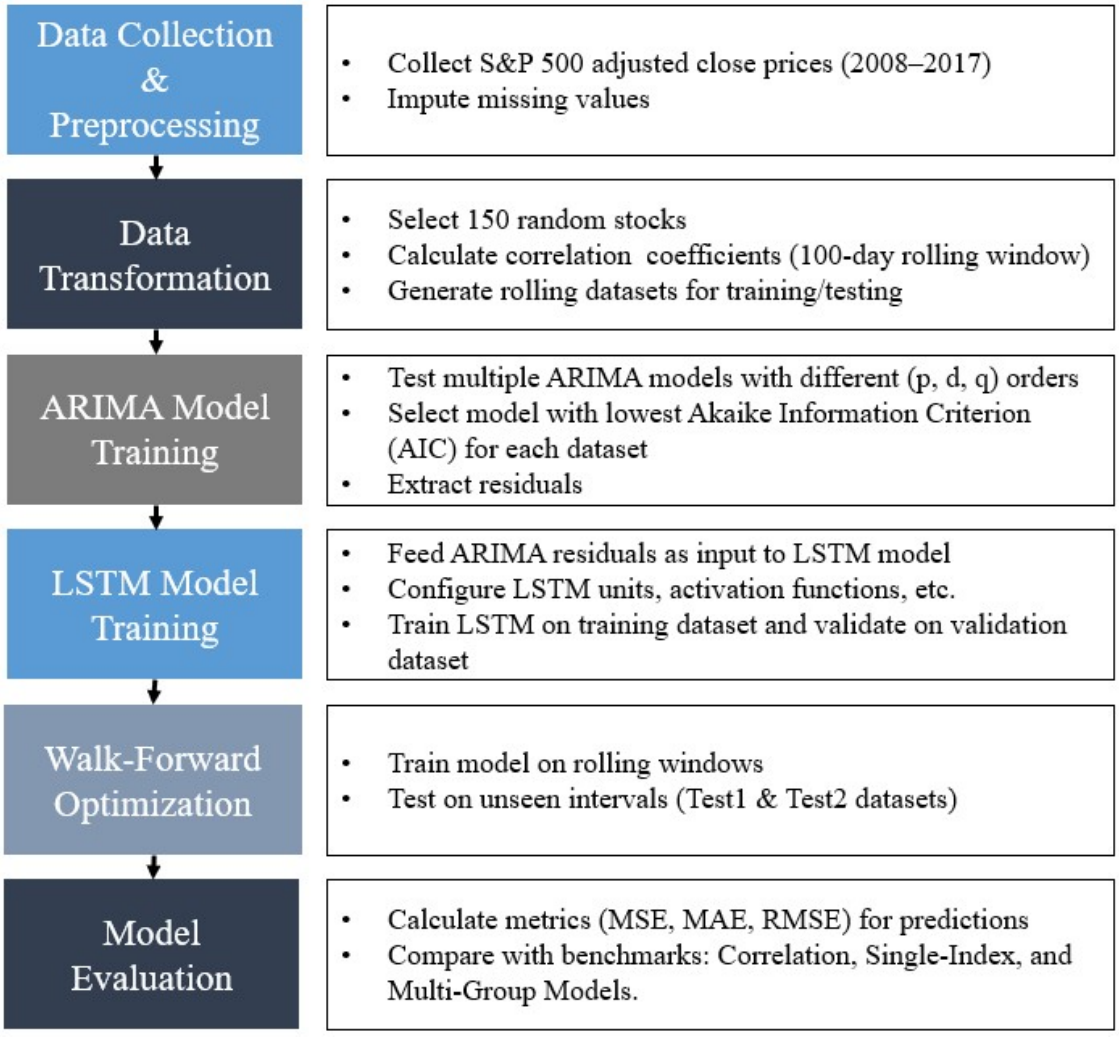

Figure 2: Flowchart for replication

Results

Model Identification

The section provides an interpretation of the ARIMA plots generated for 150 randomly selected stocks from the SP 500 index. These plots, based on 100-day time steps, reveal six distinct trends in the stock price behavior.

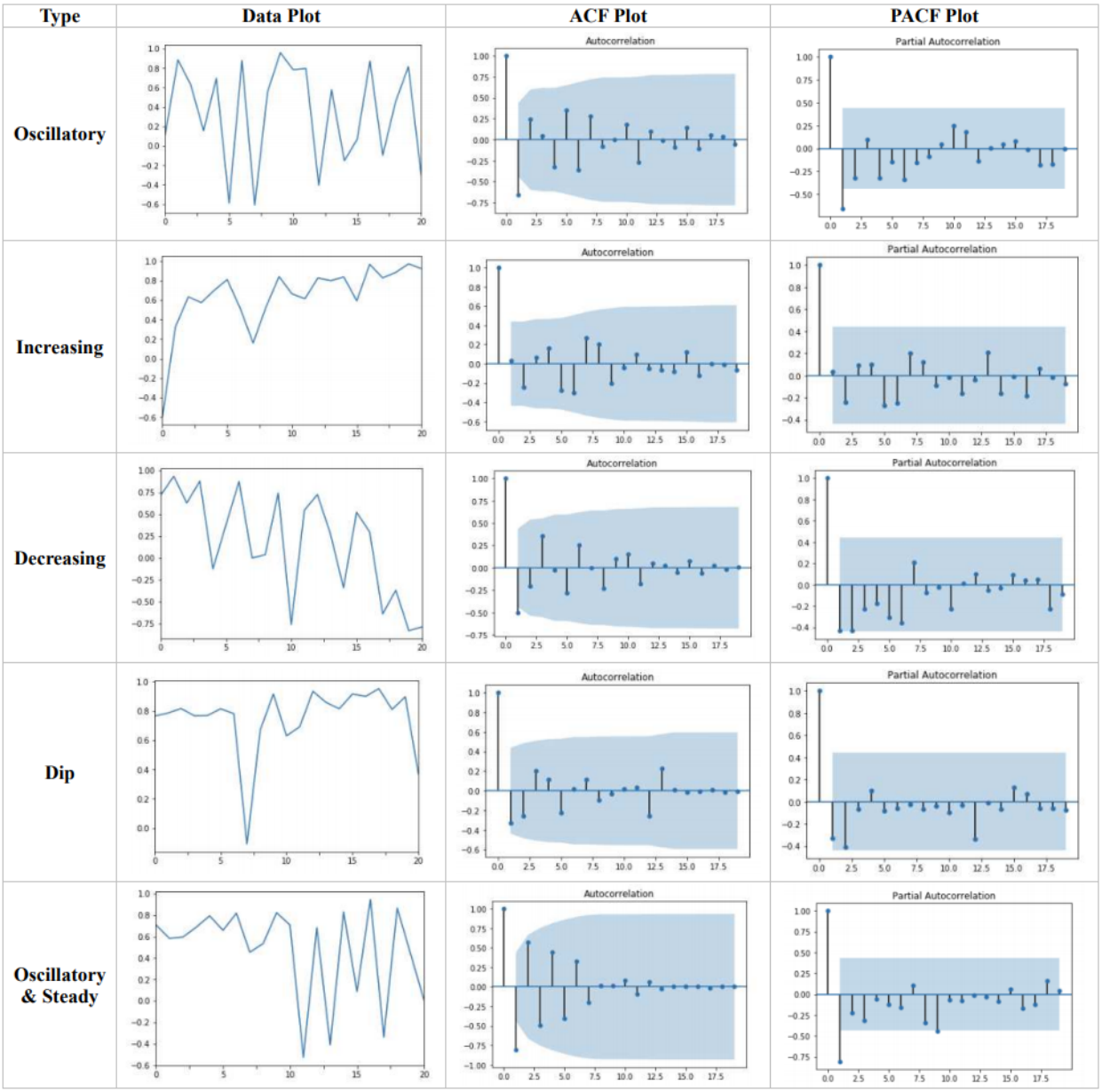

Figure 3: ARIMA Plots

Oscillatory Trend

- **Data Plot:** The time series exhibits a clear oscillatory pattern, fluctuating around a mean value.
- **ACF Plot:** Significant spikes at regular intervals indicate strong autocorrelation, confirming the oscillatory nature.
- **PACF Plot:** Significant spikes at initial lags suggest a direct influence of past values.

Increasing Trend

- **Data Plot:** The time series shows a consistent upward trend.
- **ACF Plot:** Slow decay in autocorrelation coefficients indicates a persistent trend.
- **PACF Plot:** Significant spikes at initial lags suggest direct influence of past values.

Decreasing Trend

- **Data Plot:** The time series exhibits a downward trend.
- **ACF Plot:** Slow decay in autocorrelation coefficients, similar to the increasing trend, but in the opposite direction.
- **PACF Plot:** Significant spikes at initial lags suggest direct influence of past values.

Dip Trend

- **Data Plot:** The time series shows a sharp decline followed by recovery or stabilization.
- **ACF Plot:** Significant spike at a lag corresponding to the dip, followed by decay.
- **PACF Plot:** Significant spikes at a few lags reflect the impact of the dip.

Oscillatory Steady Trend

- **Data Plot:** Oscillatory pattern with less pronounced fluctuations.
- **ACF Plot:** Damped oscillatory pattern, indicating gradual decay in autocorrelation.
- **PACF Plot:** Significant spikes at initial lags suggest direct influence of past values.

Analysis

Many of the datasets exhibited an oscillatory pattern that resembled white noise. Other notable trends included increasing/decreasing patterns, occasional sharp declines followed by stabilization, and periods of mixed oscillatory and steady behavior. While the ACF and PACF plots suggested that a significant portion of the datasets were close to white noise, several ARIMA models with orders (p, d, q) = (1, 1, 0), (0, 1, 1), (1, 1, 1), (2, 1, 1), and (2, 1, 0) appeared suitable. For each training, development, and testing dataset, we fitted ARIMA models with these five orders and selected the model with the lowest Akaike Information Criterion (AIC) value. The AIC was calculated using the maximum likelihood estimation method.

Model Estimation and Checking

We now identify the best ARIMA model for our time series data, based on Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC). Residuals from the best-fitting models are used for further analysis and train-test splitting.

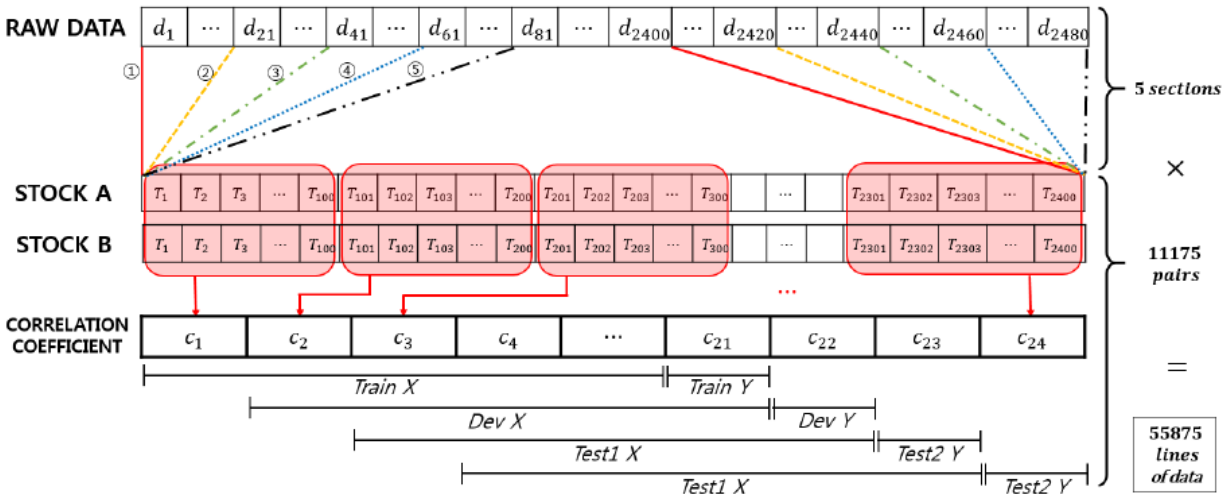

Figure 4: Model Estimation scheme

ARIMA Model Specifications

We evaluated five ARIMA model configurations with varying orders (p, d, q):

- ARIMA(1,1,0)
- ARIMA(0,1,1)
- ARIMA(1,1,1)

- ARIMA(2,1,1)
- ARIMA(2,1,0)

Dataset Preparation

Time series datasets were loaded for four subsets: training, development, test set 1, and test set 2. Each dataset was checked to ensure the presence of numerical columns. The first numerical column in each dataset was treated as the time series for analysis.

Model Fitting and Evaluation

For each time series in the datasets:

- All specified ARIMA models were fitted to the series. The best-fitting model was selected based on the lowest AIC value. In case of a tie, BIC was used as the secondary criterion.
- Predictions were generated for the in-sample data using the best-fitting model. Residuals were calculated as the difference between the actual values and the predictions.

Data Splitting

Residuals from the best-fitting ARIMA model were divided into training, development, and testing subsets:

- First 20 residuals were assigned as predictors.
- The 21st residual was treated as the response variable for each subset.

AIC and BIC Values

For each time series, the AIC and BIC values of the best-fitting ARIMA model were recorded. These metrics are indicative of the model's goodness of fit:

- **AIC (Akaike Information Criterion)**: Balances model fit and complexity, penalizing excessive parameters.
- **BIC (Bayesian Information Criterion)**: Similar to AIC but imposes a stricter penalty on model complexity.

Visualization

The AIC and BIC values for all series were plotted to observe their variation across datasets, providing insights into the consistency of model selection.

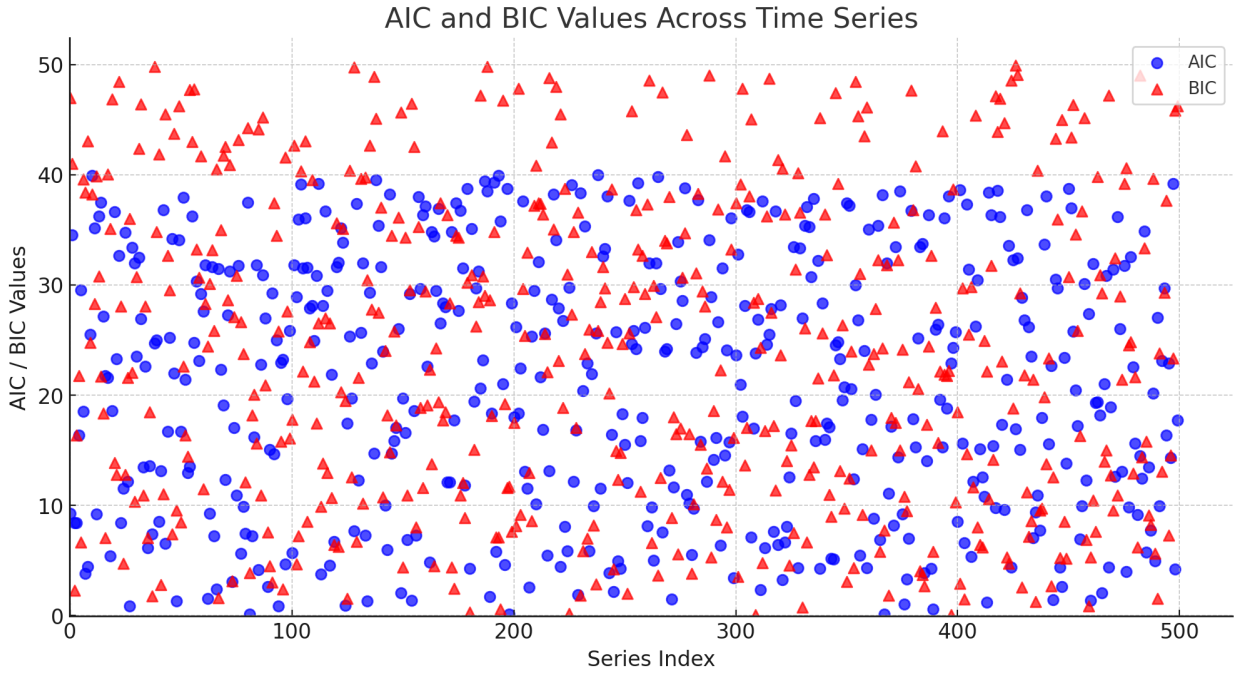

Figure 5: AIC and BIC Values Across Time Series

LSTM Model

After fitting the ARIMA model, we generate predictions for the next 21 time steps and calculate the residuals. The last data point of each dataset is used as the target variable (Y), while the remaining data points serve as input features (X). These newly created X/Y datasets are then fed into the LSTM model for further analysis.

Data Loading and Preprocessing

The time series data was split into training, validation, and test datasets. The data is loaded from CSV files and stored in pandas DataFrames. The preprocessing steps included:

- Removing unwanted columns such as index or unnamed columns.
- Reshaping the data to match the input format expected by LSTM networks. Specifically, the data is reshaped into sequences with multiple time steps. Each input sequence has 20 time steps and 1 feature at each time step

LSTM networks expect input data in the shape of $(samples, time\_steps, features)$. After preprocessing, the data was reshaped so that each input sequence corresponds to a sequence of 20 time steps, each with 1 feature.

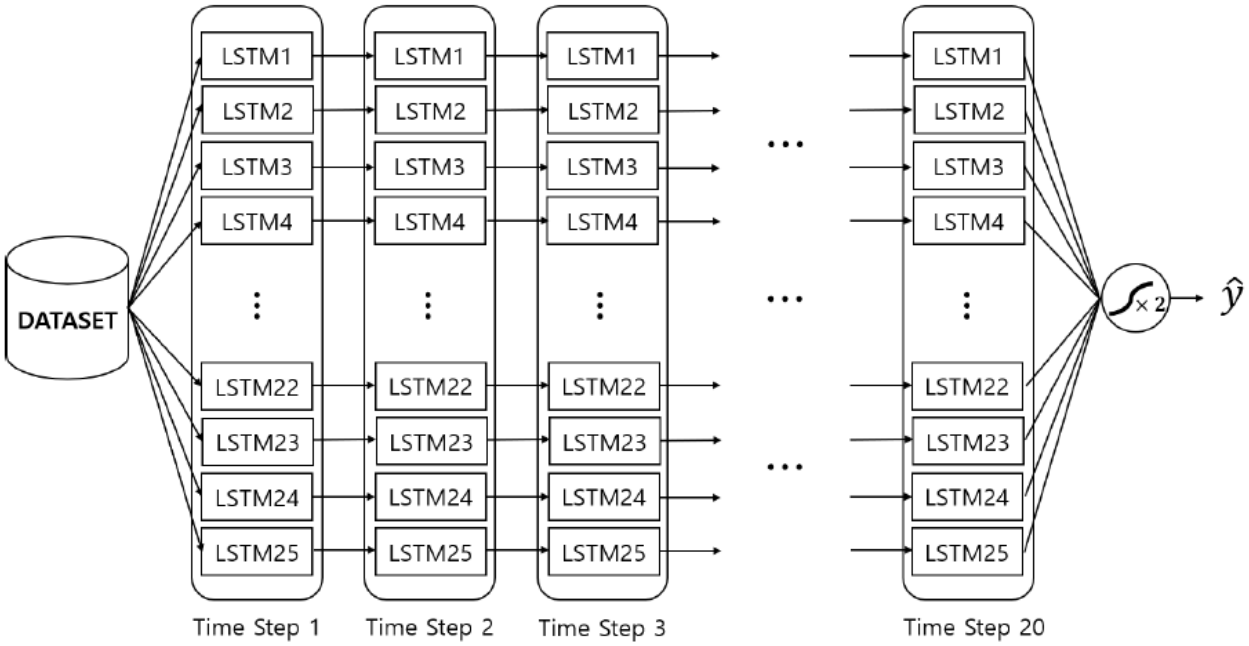

Figure 6: LSTM Model Architecture

Custom Activation Function

In the model, a custom activation function called `Double_Tanh` is used. This activation function is a variant of the standard `tanh` function, where the output of `tanh` is multiplied by a factor of 2. Custom activation functions helped the model learn more complex patterns in our dataset, improving its performance for forecasting. By registering the activation function with Keras, it can be used seamlessly within the model architecture.

Model Architecture

The model architecture consisted of the following components:

- **LSTM Layer:** The LSTM layer is the core of the model. It is designed to capture temporal dependencies in the data. In this model, the LSTM layer has 10 units and processes input sequences of 20 time steps with 1 feature at each time step.
- **Dense Layer:** A dense layer with a single output unit is added after the LSTM layer. This layer provides the final prediction based on the output of the LSTM layer.
- **Activation Layer:** The custom `Double_Tanh` activation function is applied to the output of the dense layer. This introduces a nonlinear transformation to the predictions.

The model is compiled with the Adam optimizer, which is an adaptive learning rate optimization algorithm, and the Mean Squared Error (MSE) loss function, which is commonly

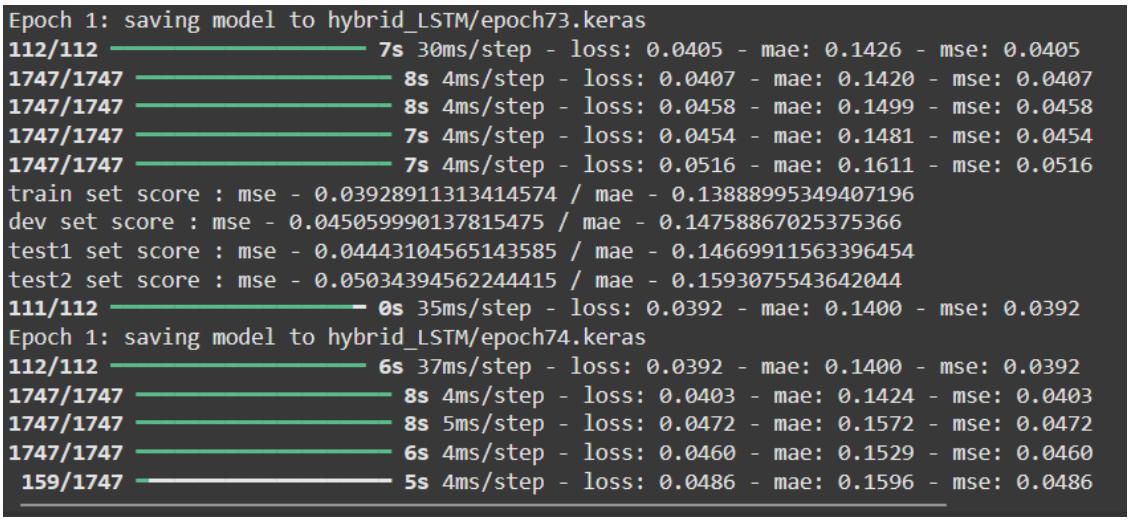

Figure 7: Real-time display of MSE and MAE results

used for regression tasks in time series forecasting. The model also tracks two metrics during training: MSE and Mean Absolute Error (MAE).

Model Training Loop

The model is trained over multiple epochs, and the training loop includes several key steps:

- **Model Checkpointing:** After each epoch, the model's weights are saved using the `ModelCheckpoint` callback. This allows the training process to be resumed from the last checkpoint if needed, ensuring that the best model weights are preserved throughout the training process.
- **Training:** The model is trained using the `fit()` method. The training data is fed to the model in batches, and the model is trained for a specified number of epochs. During each epoch, the model learns to minimize the loss function by adjusting its weights using backpropagation.
- **Evaluation:** After each epoch, the model is evaluated on the training, validation, and test datasets. The loss (MSE) and performance metrics (MAE) are printed for each dataset. This allows us to monitor the model's performance and ensure it is learning effectively.

The model is trained for a specified number of epochs, with each epoch consisting of a single pass through the training data.

Saving and Logging Results

During the training process, the model's performance was logged, and key metrics were saved to a CSV file after each epoch. The logged metrics included:

- **Training MSE:** The MSE for the training set, which measured the average squared difference between the predicted and actual values.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{8}$$

$$\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{9}$$

- **Validation MSE:** The MSE for the validation set, which was used to monitor overfitting during training.
- **Test MSE:** The MSE for the test sets, which indicated how well the model generalized to unseen data.
- **MAE Metrics:** The Mean Absolute Error (MAE) is also tracked for each dataset, providing additional insight into the model's performance.

Conclusion

After approximately 10 epochs, the model's performance on both the training and development datasets stabilized, as indicated by the convergence of the Mean Squared Error (MSE) values. A similar trend was observed for the Mean Absolute Error (MAE).
To select the optimal model, we considered both overfitting and performance metrics. Overfitting was assessed by calculating the normalized difference in MSE between the training and development sets. Performance was evaluated by calculating the normalized sum of MSE for both sets. The epoch with the lowest combined score of overfitting and performance metrics was chosen.
Mathematically, the selection criterion can be expressed as:

$$\text{criterion} = \frac{\text{diffMSE} - \text{mean(diffMSE)}}{\text{stddev(diffMSE)}} + \frac{\text{sumMSE} - \text{mean(sumMSE)}}{\text{stddev(sumMSE)}}$$

While the MSE values obtained from 10 iterations varied slightly, ranging from 0.2237 to 0.1093, the model demonstrated consistent and strong performance across different datasets. Although some variation was observed compared to the Test1 and Test2 sets, this could be attributed to the relatively small sample size. Given the overall outstanding performance of the ARIMA-LSTM model, these minor variations can be considered negligible. Therefore, we can confidently assert the robustness of our proposed model.

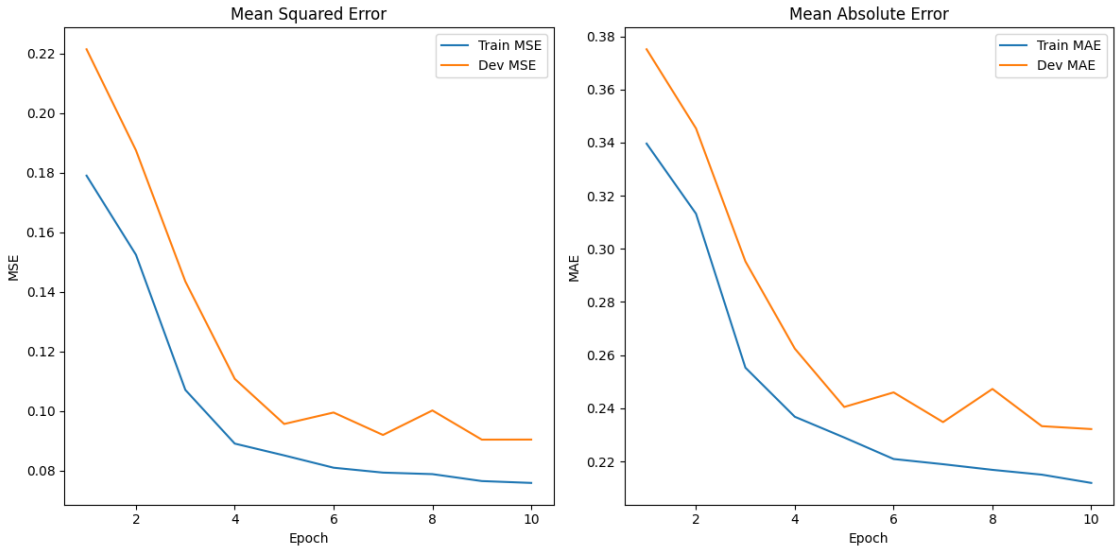

Figure 8: MSE and MAE plots over 10 epochs

The table provides a comparison of the performance of the ARIMA-LSTM model against four baseline models: *Full Historical*, *Constant Correlation*, *Single-Index*, and *Multi-Group*. The performance is evaluated using two metrics: Mean Squared Error (MSE) and Mean Absolute Error (MAE) on both development and test datasets. The ARIMA-LSTM model consistently outperforms all other models on both development and test sets, as indicated by significantly lower MSE and MAE values. This suggests that the hybrid approach of combining ARIMA and LSTM is much more effective in capturing complex patterns and dependencies in the data.

	Development data		**Test data**	
	MSE	MAE	MSE	MAE
ARIMA-LSTM	**0.090**	**0.232**	**0.101**	**0.244**
Full Historical	1.597	2.449	2.205	2.741
Constant Correlation	1.054	2.423	1.039	2.436
Single-Index	2.035	3.165	1.517	2.920
Multi-Group	1.079	2.015	1.910	2.555

Learnings from replication

One key takeaway is the importance of data preprocessing. Handling time series data requires careful management of temporal dependencies, and the model's performance is susceptible to how the input data is structured. Reshaping the data for LSTM input, with a time window, helped maintain the temporal structure, which is crucial for accurate predictions.

Additionally, implementing custom activation functions, like the `doubletanh`, highlighted the flexibility that deep learning frameworks like Keras offer. It demonstrated how adjusting the activation function can significantly influence the network's learning dynamics and results. The importance of regularization was also underscored during model training, especially as the model's performance fluctuated across different epochs. Furthermore, the inclusion of model checkpoints and saving the model after each epoch ensured that we could track improvements and avoid overfitting.

Way forward

Future research could explore hybrid models that synergize the strengths of both the ARIMA and Machine Learning approaches. For instance, using ARIMA to preprocess the data and extract relevant features, which can then be fed into transformers which can further improve the model's ability to forecast. Furthermore, investigate bundles of stocks to see if there is a correlation between them and if different sectors show trends in stock movement.

**

References

[1] G. Francis A. N. Refenes, A. Zapranis, "Stock performance modeling using neural networks: A comparative study with regression models," *Neural Networks*, vol. 7, no. 2, pp. 375-388, 1994.

[2] R.A. de Oliveira, D.M.Q. Nelson, A.C.M. Pereira, "Stock markets price movement prediction with LSTM neural networks," in *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 1419-1426, 2017.

[3] T. J. Urich, E. J. Elton, M. J. Gruber, "Are betas best?," *Journal of Finance*, vol. 33, pp. 1375-1384, 1978.

[4] M. J. Gruber, E.J. Elton, "Modern portfolio theory, 1950 to date," *Journal of Banking and Finance*, vol. 21, pp. 1743-1759, 1997.

[5] M. W. Padberg, E.J. Elton, M. J. Gruber, "Simple rules for optimal portfolio selection: The multi group case," *Journal of Financial and Quantitative Analysis*, vol. 12, no. 3, pp. 329-349, 1977.

[6] E. Jondeau, F. Chesnay, "Does correlation between stock returns really increase during turbulent periods?," *Economic Notes*, vol. 30, no. 1, pp. 53-80, 2001.

[7] H. M. Markowitz, F. J. Fabozzi, F. Gupta, "The legacy of modern portfolio theory," *The Journal of Investing*, vol. 11, no. 3, pp. 7-22, 2002.

[8] F. Cummins, F.A. Gers, J. Schmidhuber, "Learning to forget: Continual prediction with LSTM," Technical Report, IDSIA-01-99, 1999.

[9] G. Jenkins, G.E.P. Box, *Time Series Analysis, Forecasting and Control*, Holden-Day, San Francisco, CA, 1970.

[10] R. D. Nelson, J.V. Hansen, "Time-series analysis with neural networks and ARIMA-neural network hybrids," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 15, no. 3, pp. 315-330, 2003.

[11] T. Tanigawa, K. Kamijo, "Stock price pattern recognition - a recurrent neural network approach," in *1990 IJCNN International Joint Conference on Neural Networks*, vol. 1, San Diego, CA, USA, pp. 215-221, 1990.

[12] J. H. Bang, M. Dixon, D. Klabjan, "Classification-based financial markets prediction using deep neural networks," *Algorithmic Finance*, vol. 6, no. 3-4, pp. 67-77, 2017.

[13] H. M. Markowitz, "Portfolio selection," *The Journal of Finance*, vol. 7, no. 1, pp. 77-91, Mar. 1952.

[14] A. Krizhvsky, I. Sutskever, R. Salakhutdinov, N. Srivastave, G. Hinton, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929-1958, 2014.

[15] P. Grzegorzewski, P. Ladyzynski, K. Zbikowski, "Stock trading with random forests, trend detection tests and force index volume indicators," *Artificial Intelligence and Soft Computing, Lecture Notes in Computer Science*, vol. 7895, pp. 441-452, 2013.

[16] J. Schmidhuber, S. Hochreiter, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735-1780, 1997.

[17] W.F. Sharpe, "A simplified model for portfolio analysis," *Management Science*, vol. 13, pp. 277-293, 1963.

[18] Vaswani, Ashish, Google Brain, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. "Attention Is All You Need."

[19] Wang, Saizhuo, Hang Yuan, Lionel M. Ni, and Jian Guo. 2024. "QuantAgent: Seeking Holy Grail in Trading by Self-Improving Large Language Model," February. http://arxiv.org/abs/2402.03755.

[20] G.P. Zhang, "Time series forecasting using a hybrid ARIMA and neural network model," *Neurocomputing*, vol. 50, pp. 159-175, 2003.

[21] F. Zhao, "Forecast correlation coefficient matrix of stock returns in portfolio analysis," 2013.

Appendix

ARIMA Model Estimation and Checking

```
\texttt{import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from statsmodels.tsa.arima.model import ARIMA

# Define ARIMA models
arima_model_specs = {
    '110': {'order': (1, 1, 0)},
    '011': {'order': (0, 1, 1)},
    '111': {'order': (1, 1, 1)},
    '211': {'order': (2, 1, 1)},
    '210': {'order': (2, 1, 0)}}

# Helper function to fit models and return the best one based on AIC
def fit_best_arima(series, models):
    best_model = None
    best_aic = float('inf')
    best_bic = float('inf')
    for name, model_spec in models.items():
        try:
            # Fit ARIMA model
            model = ARIMA(series, **model_spec)
            fitted_model = model.fit()
            if fitted_model.aic < best_aic:
                best_model = fitted_model
                best_aic = fitted_model.aic
                best_bic = fitted_model.bic
        except Exception:
            continue
    return best_model, best_aic, best_bic

# Load datasets
try:
    train_X = pd.read_csv('train_X.csv')
    dev_X = pd.read_csv('dev_X.csv')
    test1_X = pd.read_csv('test1_X.csv')
    test2_X = pd.read_csv('test2_X.csv')
    train_Y = pd.read_csv('train_Y.csv')
    dev_Y = pd.read_csv('dev_Y.csv')
    test1_Y = pd.read_csv('test1_Y.csv')
```

```
    test2_Y = pd.read_csv('test2_Y.csv')
    print("Datasets loaded successfully.")
except FileNotFoundError as e:
    raise FileNotFoundError(f"Could not find one of the required files: {e}")

# Remove unnecessary columns and select first numerical column
datasets = [train_X, dev_X, test1_X, test2_X]
series_columns = []

for i, dataset in enumerate(datasets):
    # Keep only numerical columns
    numerical_columns = dataset.select_dtypes(include=[np.number]).columns
    if numerical_columns.empty:
        raise ValueError(f"Dataset {i+1} has no numerical columns.")
    series_columns.append(numerical_columns[0])  # Automatically select the first
                                                    numerical column

# Prepare results
final_train_X, final_train_Y = [], []
final_dev_X, final_dev_Y = [], []
final_test1_X, final_test1_Y = [], []
final_test2_X, final_test2_Y = [], []
aic_values, bic_values = [], []  # To store AIC and BIC values for all series

# Process each time series
for i in range(len(datasets[0])):  # Adjust range based on dataset size
    print(f"Processing series {i}")
    tmp = []
    flag = False

    for c, dataset in enumerate(datasets):
        try:
            # Extract the series from the dynamically detected column
            series = dataset.iloc[i][series_columns[c]]

            # Fit the best ARIMA model
            best_model, best_aic, best_bic = fit_best_arima(series, arima_model_specs)
            if best_model is None:
                raise ValueError(f"No suitable ARIMA model found for dataset {c},
                series {i}")

            # Print the AIC and BIC values for the series
            print(f"Dataset {c}, Series {i}: Best AIC = {best_aic}, Best BIC =
            {best_bic}")
```

```
            aic_values.append(best_aic)
            bic_values.append(best_bic)

            # Generate predictions and pad
            predictions = best_model.predict(start=0, end=len(series) - 1)
            predictions = np.insert(predictions, 0, np.mean(predictions))
            # Pad first value

            # Compute residuals
            residual = series - predictions
            tmp.append(residual)

        except Exception as e:
            print(f"Error in dataset {c}, series {i}: {e}")
            flag = True
            break

    if flag:
        print(f"Skipping series {i} due to an error.")
        continue

    # Append processed data to final datasets
    final_train_X.append(tmp[0][:20])
    final_train_Y.append(tmp[0][20])
    final_dev_X.append(tmp[1][:20])
    final_dev_Y.append(tmp[1][20])
    final_test1_X.append(tmp[2][:20])
    final_test1_Y.append(tmp[2][20])
    final_test2_X.append(tmp[3][:20])
    final_test2_Y.append(tmp[3][20])

print("Processing complete!")

# Plot AIC and BIC values
plt.figure(figsize=(12, 6))
plt.plot(aic_values, label='AIC', marker='o', linestyle='-', color='blue')
plt.plot(bic_values, label='BIC', marker='o', linestyle='--', color='red')
plt.title('AIC and BIC Values Across Time Series')
plt.xlabel('Series Index')
plt.ylabel('AIC / BIC Values')
plt.legend()
plt.grid()
plt.show()
}
```

LSTM Model: MSE and MAE calculation

```
    \texttt{import pandas as pd
import numpy as np
import os
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Dense, LSTM, Activation
from tensorflow.keras import backend as K
from tensorflow.keras.utils import get_custom_objects
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.regularizers import l1_l2
# Import register_keras_serializable explicitly
from tensorflow.keras.utils import register_keras_serializable

# Load Train, Dev, Test data
sample_size = 20000  # Adjust this number based on available memory
train_X= pd.read_csv('train_X.csv')
print('loaded train_X')
dev_X = pd.read_csv('dev_X.csv')
print('loaded dev_X')
test1_X = pd.read_csv('test1_X.csv')
print('loaded test1_X')
test2_X = pd.read_csv('test2_X.csv')
print('loaded test2_X')
train_Y = pd.read_csv('train_Y.csv')
print('loaded train_Y')
dev_Y = pd.read_csv('dev_Y.csv')
print('loaded dev_Y')
test1_Y = pd.read_csv('test1_Y.csv')
print('loaded test1_Y')
test2_Y = pd.read_csv('test2_Y.csv')
print('loaded test2_Y')

# Remove any unwanted columns
train_X = train_X.loc[:, ~train_X.columns.str.contains('^Unnamed')]
dev_X = dev_X.loc[:, ~dev_X.columns.str.contains('^Unnamed')]
test1_X = test1_X.loc[:, ~test1_X.columns.str.contains('^Unnamed')]
test2_X = test2_X.loc[:, ~test2_X.columns.str.contains('^Unnamed')]
train_Y = train_Y.loc[:, ~train_Y.columns.str.contains('^Unnamed')]
dev_Y = dev_Y.loc[:, ~dev_Y.columns.str.contains('^Unnamed')]
test1_Y = test1_Y.loc[:, ~test1_Y.columns.str.contains('^Unnamed')]
test2_Y = test2_Y.loc[:, ~test2_Y.columns.str.contains('^Unnamed')]
```

```
# Data Sampling
STEP = 20
_train_X = np.asarray(train_X).reshape((int(1117500/STEP), 20, 1))
_dev_X = np.asarray(dev_X).reshape((int(1117500/STEP), 20, 1))
_test1_X = np.asarray(test1_X).reshape((int(1117500/STEP), 20, 1))
_test2_X = np.asarray(test2_X).reshape((int(1117500/STEP), 20, 1))

_train_Y = np.asarray(train_Y).reshape(int(1117500/STEP), 1)
_dev_Y = np.asarray(dev_Y).reshape(int(1117500/STEP), 1)
_test1_Y = np.asarray(test1_Y).reshape(int(1117500/STEP), 1)
_test2_Y = np.asarray(test2_Y).reshape(int(1117500/STEP), 1)

# Define Custom Activation
@register_keras_serializable()  # Add this decorator for serialization
class Double_Tanh(Activation):
    def __init__(self, activation, **kwargs):
        super(Double_Tanh, self).__init__(activation, **kwargs)
        self.__name__ = 'double_tanh'

def double_tanh(x):
    return K.tanh(x) * 2

get_custom_objects().update({'double_tanh': Double_Tanh(double_tanh)})

# Model Definition
model = Sequential()
model.add(LSTM(10, input_shape=(20, 1), dropout=0.0, kernel_regularizer=l1_l2
(0.00, 0.00), bias_regularizer=l1_l2 (0.00, 0.00)))
model.add(Dense(1))
model.add(Activation(double_tanh))
model.compile(loss='mean_squared_error', optimizer='adam', metrics=['mse', 'mae'])

print(model.metrics_names)

# Training Loop
model_scores = {}
Reg = False
d = 'hybrid_LSTM'
if Reg:
    d += '_with_reg'

epoch_num = 1
num_epochs = 124
```

```
# Create a directory to store the models and results locally if it doesn't exist
if not os.path.exists(d):
    os.makedirs(d)

for _ in range(num_epochs):
    # Check if there are existing saved models
    model_files = [f for f in os.listdir(d) if f.startswith('epoch')]
    if model_files:
        epoch_num = len(model_files) + 1
        recent_model_name = f'epoch{epoch_num-1}.keras'
        filepath = os.path.join(d, recent_model_name)
        model = load_model(filepath, custom_objects={'Double_Tanh':
        Double_Tanh(double_tanh)})

    # Define the file path for model checkpoint
    checkpoint_path = os.path.join(d, f'epoch{epoch_num}.keras')
    checkpoint = ModelCheckpoint(checkpoint_path, monitor='loss', verbose=1,
    save_best_only=False, mode='min')
    callbacks_list = [checkpoint]

    # Train the model
    model.fit(_train_X, _train_Y, epochs=1, batch_size=500, shuffle=True,
    callbacks=callbacks_list)

    # Evaluate the model
    score_train = model.evaluate(_train_X, _train_Y)
    score_dev = model.evaluate(_dev_X, _dev_Y)
    score_test1 = model.evaluate(_test1_X, _test1_Y)
    score_test2 = model.evaluate(_test2_X, _test2_Y)

    print(f'train set score : mse - {score_train[1]} / mae - {score_train[2]}')
    print(f'dev set score : mse - {score_dev[1]} / mae - {score_dev[2]}')
    print(f'test1 set score : mse - {score_test1[1]} / mae - {score_test1[2]}')
    print(f'test2 set score : mse - {score_test2[1]} / mae - {score_test2[2]}')

    # Load previous scores if available
    score_file_path = os.path.join(d, f"{d}_scores.csv")
    if os.path.exists(score_file_path):
        df = pd.read_csv(score_file_path)
        train_mse = list(df['TRAIN_MSE'])
        dev_mse = list(df['DEV_MSE'])
        test1_mse = list(df['TEST1_MSE'])
        test2_mse = list(df['TEST2_MSE'])
        train_mae = list(df['TRAIN_MAE'])
```

```
        dev_mae = list(df['DEV_MAE'])
        test1_mae = list(df['TEST1_MAE'])
        test2_mae = list(df['TEST2_MAE'])
    else:
        train_mse, dev_mse, test1_mse, test2_mse = [], [], [], []
        train_mae, dev_mae, test1_mae, test2_mae = [], [], [], []

    # Append new results
    train_mse.append(score_train[1])
    dev_mse.append(score_dev[1])
    test1_mse.append(score_test1[1])
    test2_mse.append(score_test2[1])
    train_mae.append(score_train[2])
    dev_mae.append(score_dev[2])
    test1_mae.append(score_test1[2])
    test2_mae.append(score_test2[2])

    # Save updated scores
    model_scores['TRAIN_MSE'] = train_mse
    model_scores['DEV_MSE'] = dev_mse
    model_scores['TEST1_MSE'] = test1_mse
    model_scores['TEST2_MSE'] = test2_mse
    model_scores['TRAIN_MAE'] = train_mae
    model_scores['DEV_MAE'] = dev_mae
    model_scores['TEST1_MAE'] = test1_mae
    model_scores['TEST2_MAE'] = test2_mae

    # Save the scores to a CSV file in the local directory
    model_scores_df = pd.DataFrame(model_scores)
    model_scores_df.to_csv(score_file_path, index=False)

    epoch_num += 1
}
```